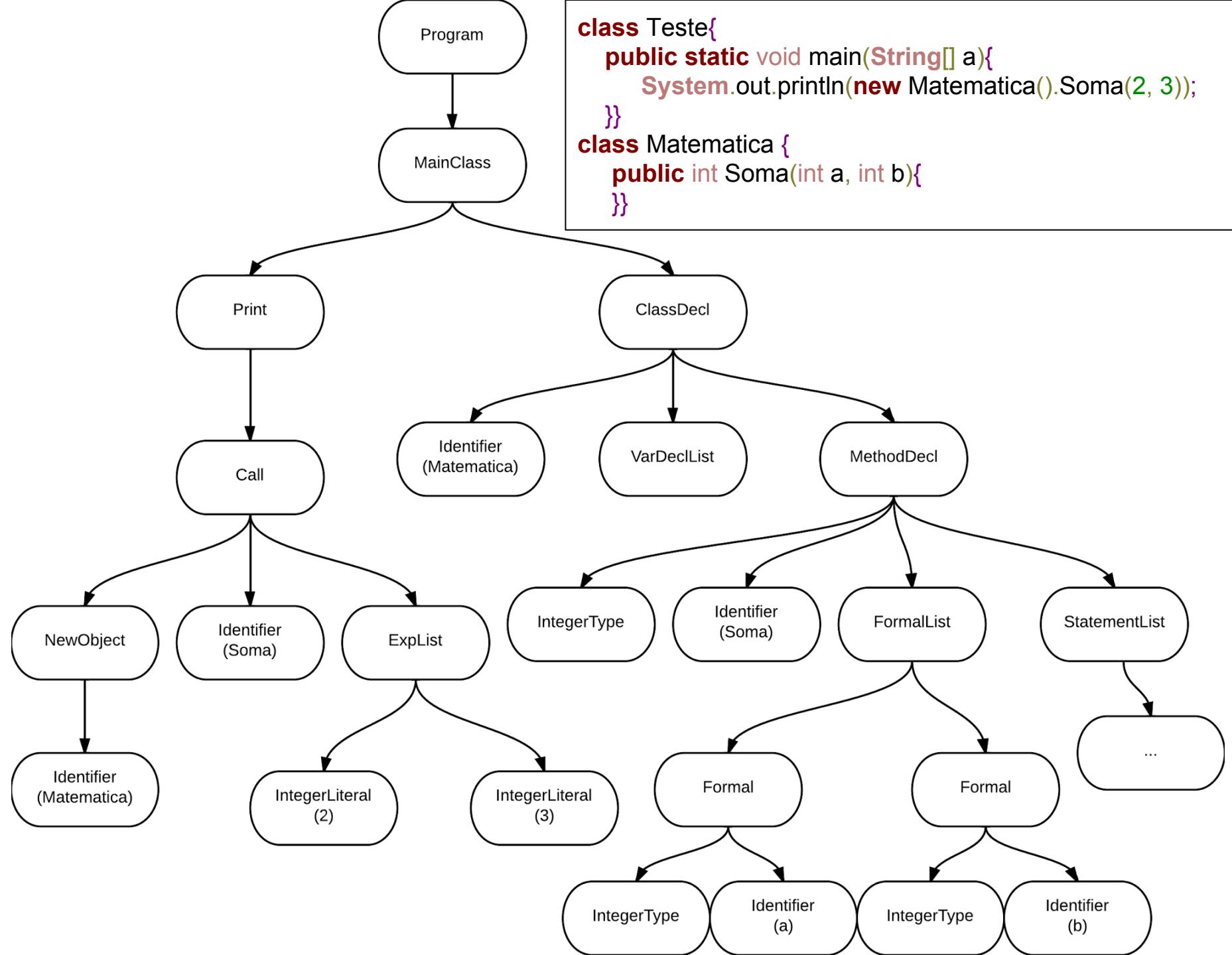


MiniJava/LLVM

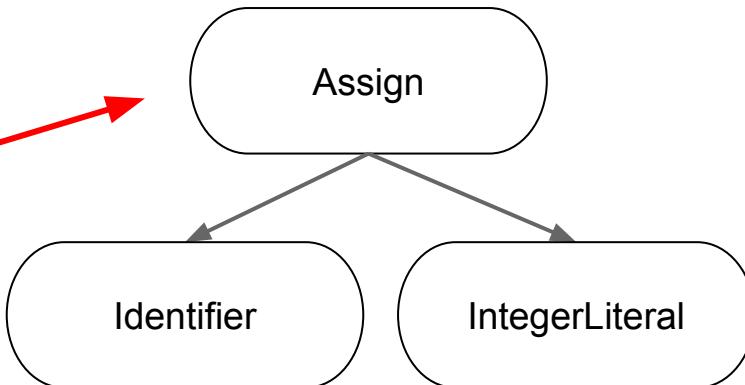
Após o front-end

- Depois do léxico e do sintático, o compilador:
 - sabe que o texto do programa está OK
 - constrói a ***abstract syntax tree (AST)***
 - constrói a **tabela de símbolos**
- Tipos da AST:
 - *Program*,
 - *MainClass*,
 - *ClassDeclSimple*,
 - *MethodDecl*,
 - *Formal*,
 - *Block*,
 - *If*,
 - *While*
 - ...



Já posso emitir código a partir da AST?

```
class Matematica {  
    int x;  
    public int Soma(int a, int b){  
        int y;  
        y = 10;  
    }  
}
```

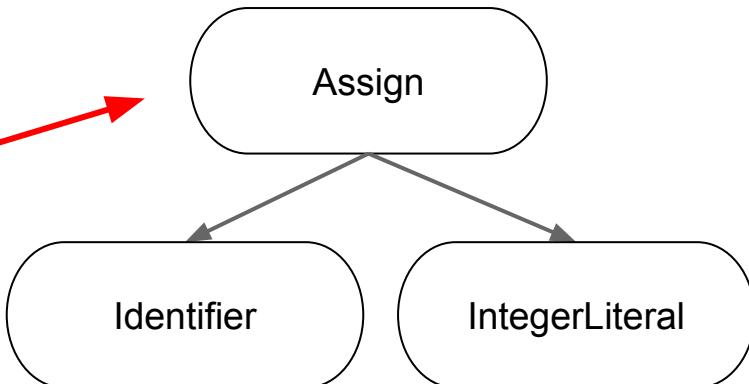


%class.Matematica = type { i32 } ; estrutura da classe

```
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {  
entry0:  
    %a_addr = alloca i32  
    store i32 %a, i32 * %a_addr  
    %b_addr = alloca i32  
    store i32 %b, i32 * %b_addr  
    %y = alloca i32  
    store i32 10, i32 * %y  
    ret i32 1  
}
```

Já posso emitir código a partir da AST?

```
class Matematica {  
    int x;  
    public int Soma(int a, int b){  
        int y;  
        x = 10;  
    }  
}
```

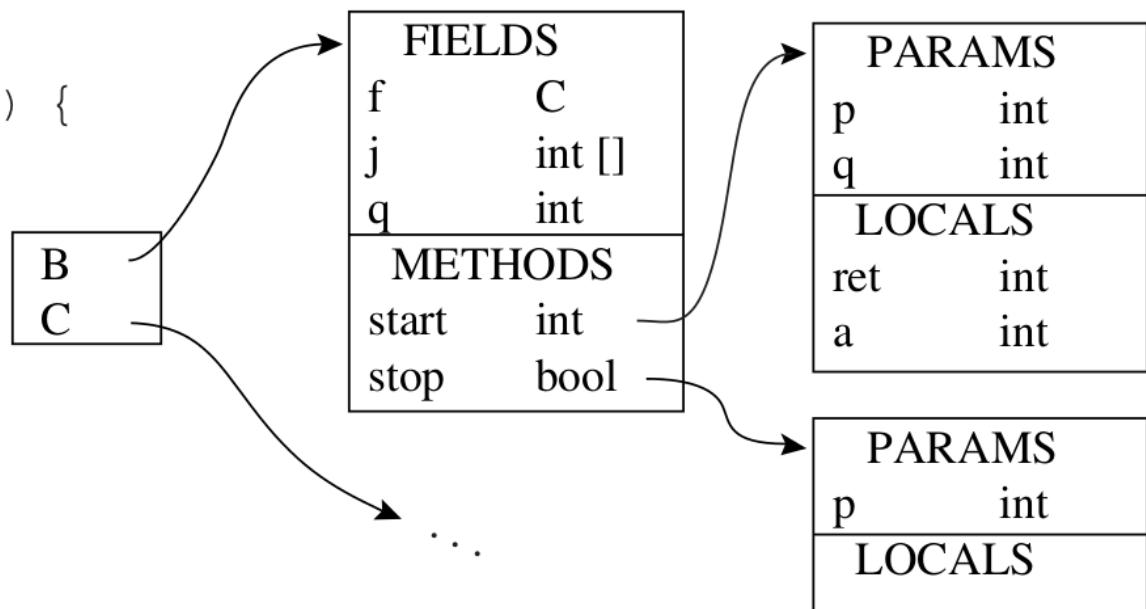


%class.Matematica = type { i32 } ; estrutura da classe

```
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {  
entry0:  
    %a_addr = alloca i32  
    store i32 %a, i32 * %a_addr  
    %b_addr = alloca i32  
    store i32 %b, i32 * %b_addr  
    %y = alloca i32  
    %tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0  
    store i32 10, i32 * %tmp0  
    ret i32 1  
}
```

Tabela de Símbolo

```
class B {  
    C f;  int [] j;  int q;  
    public int start(int p, int q) {  
        int ret;  int a;  
        /* ... */  
        return ret;  
    }  
    public boolean stop(int p) {  
        /* ... */  
        return false;  
    }  
}  
  
class C {  
    /* ... */  
}
```



Essa tabela já existe

```
public class Codegen extends VisitorAdapter{
    private List<LlvmInstruction> assembler;
    private Env env;
    private ClassInfo classEnv;      // aponta para a classe em uso
    private MethodInfo methodEnv; // aponta para o método em uso

    public Codegen(){
    }

    public String translate(Program p, Env env ){
        ...
        this.env = env;
    }
}
```

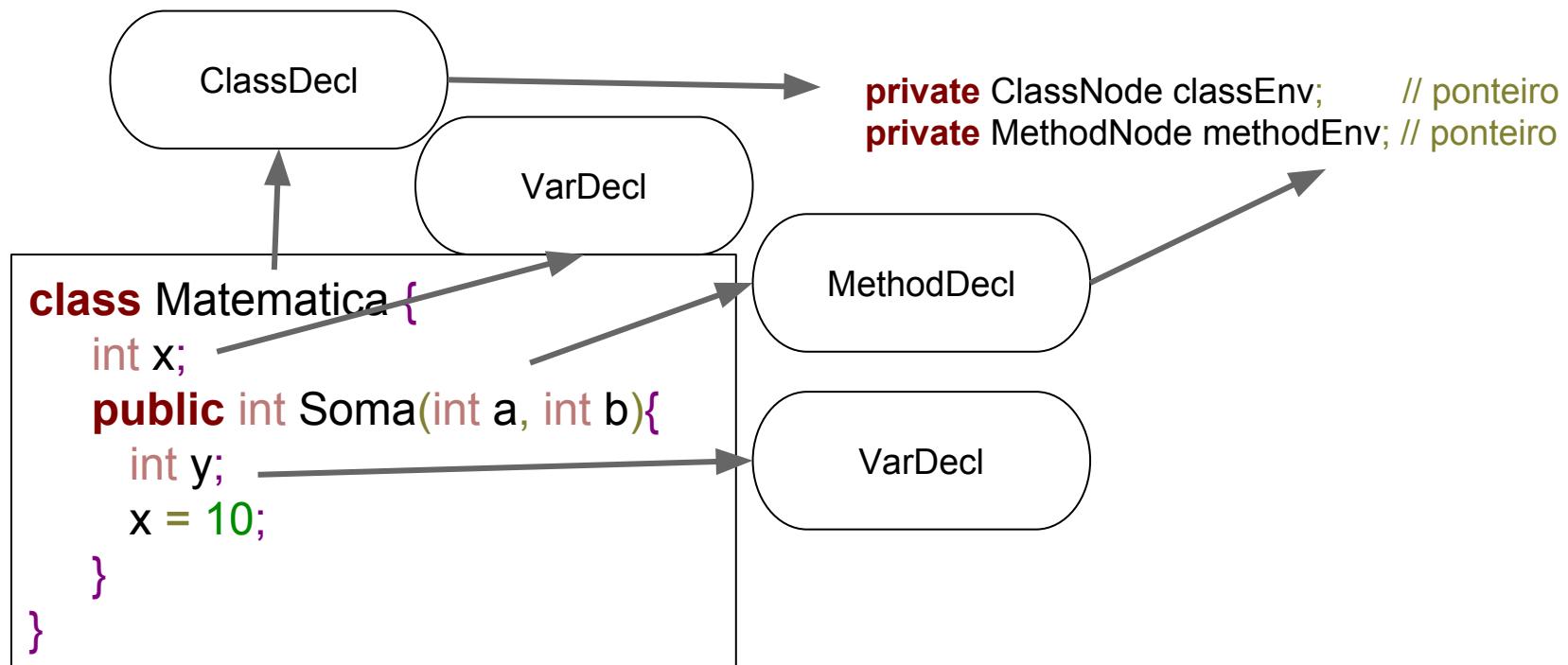
Essa tabela é suficiente?

- Os tipos armazenados são aqueles encontrados nos objetos da AST
- Não podemos fazer alterações (engessada)
- **Podemos fazer a nossa Tabela de Símbolos**

```
LlvmType  
%class.Matematica = type { i32 } ; estrutura da classe  
  
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {  
entry0:  
...  
%tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0  
...  
}
```

Construir a SymTab junto com a Emissão?

- Em MiniJava, toda variável deve ser declarada no começo da classe ou do método.
- Deste modo, é possível ir emitindo código e construindo a Tabela de Símbolos?



Problemas:

- Isso funciona quando não há herança!
- MiniJava possui herança

```
class Matematica extends Base {  
    public int Soma(int a, int b){  
        int y;  
        x = 20;  
    }  
}
```

```
class Base {  
    int x;  
    public int Init(int a, int b){  
        int y;  
        x = 10;  
    }  
}
```

Precisa conhecer a estrutura
de **Base** pra emitir código
em **Matemática**

E agora, José?

- Vamos visitar os nós necessários para construir a Tabela de Símbolos antes de começar emitir código

```
public class Codegen extends VisitorAdapter{

    private SymTab mySymTab;
    private ClassNode classEnv;      // Aponta para a classe atualmente em uso em symTab
    private MethodNode methodEnv;   // Aponta para o metodo atualmente em uso em symTab

    // Metodo de entrada do Codegen
    public String translate(Program p, Env env ){

        codeGenerator = new Codegen();
        codeGenerator.symTab.FillSymTab(p);
    }
}
```

```
class SymTab extends VisitorAdapter{
    public Map<String, ClassNode> classes;
    private ClassNode classEnv; //aponta para a classe em uso

    public LlvmValue FillTabSymbol(Program n){
        n.accept(this);
        return null;
    }

    public LlvmValue visit(Program n){
        n.mainClass.accept(this);

        for (util.List<ClassDecl> c = n.classList; c != null; c = c.tail)
            c.head.accept(this);

        return null;
    }

    public LlvmValue visit(MainClass n){
        classes.put(n.className.s, new ClassNode(n.className.s, null, null));
        return null;
    }

    public LlvmValue visit(ClassDeclSimple n){
        // Constroi TypeList com os tipos das variáveis da Classe (vai formar a Struct da classe)
        // Constroi VarList com as Variáveis da Classe
        classes.put(n.name.s, new ClassNode(n.name.s,
            new LlvmStructure(TypeList),
            VarList)
        );
        // Percorre n.methodList visitando cada método
    }
}
```

Construções em LLVM

Classes

- Classes são estruturas contendo suas variáveis
- A classe **LlvmStructure** é utilizada para construir essa estrutura

```
class Teste {
    public static void main(String[] a){
        System.out.println(
            new Matematica().Soma(1,2));
    }
}

class Matematica {
    int x;
    int[] v;
    Complex C;
    public int Soma(int a, int b){
        int y;
        y = 10;
        return 1;
    }
}
class Complex {
    int i;
}
```

```

@.formatting.string = private constant [4 x i8] c"%d\0A\00"
%class.Teste = type {}

%class.Matematica = type { i32, i32 *, %class.Complex * }

%class.Complex = type { i32 }

define i32 @main() {
entry0:
    %tmp0 = alloca i32
    store i32 0, i32 * %tmp0
    %tmp2 = mul i32 20, 1
    %tmp3 = call i8* @malloc( i32 %tmp2)
    %tmp1 = bitcast i8* %tmp3 to %class.Matematica*
    %tmp4 = call i32 @_Soma_Matematica(%class.Matematica * %tmp1, i32 1, i32 2)
    %tmp5 = getelementptr [4 x i8] * @.formatting.string, i32 0, i32 0
    %tmp6 = call i32 (i8 *, ...)* @printf(i8 * %tmp5, i32 %tmp4)
    %tmp7 = load i32 * %tmp0
    ret i32 %tmp7
}

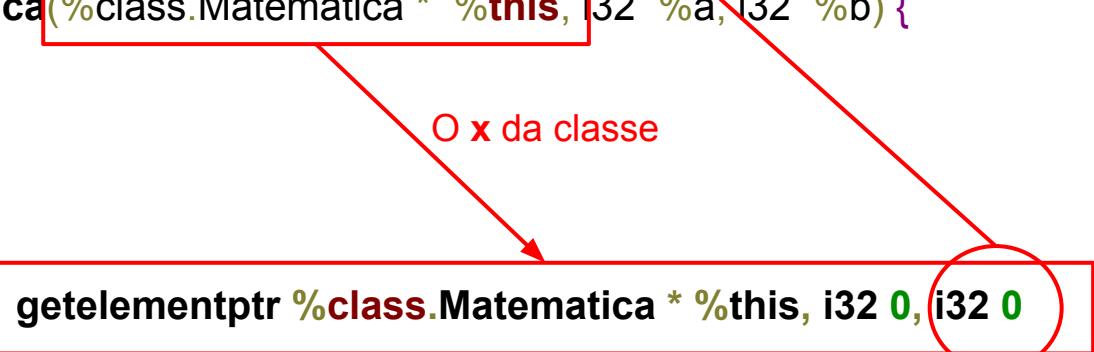
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
    %a_tmp = alloca i32
    store i32 %a, i32 * %a_tmp
    %b_tmp = alloca i32
    store i32 %b, i32 * %b_tmp
    %y = alloca i32
    store i32 10, i32 * %y
    ret i32 1
}

declare i32 @printf(i8 *, ...)
declare i8 * @malloc(i32)

```

Tamanho de %class.Matematica em **x86_64**
inteiro 4 bytes, ptr 8 bytes

```
@.formatting.string = private constant [4 x i8] c"%d\0A\00"
%class.Teste = type { }
%class.Matematica = type { i32, i32 *, %class.Complex * }
%class.Complex = type { i32 }
define i32 @main() {
entry0:
%tmp0 = alloca i32
store i32 0, i32 * %tmp0
%tmp2 = mul i32 20, 1
%tmp3 = call i8* @malloc( i32 %tmp2)
%tmp1 = bitcast i8* %tmp3 to %class.Matematica*
%tmp4 = call i32 @_Soma_Matematica(%class.Matematica * %tmp1, i32 1, i32 2)
%tmp5 = getelementptr [4 x i8] * @.formatting.string, i32 0, i32 0
%tmp6 = call i32 (i8 *, ...)* @printf(i8 * %tmp5, i32 %tmp4)
%tmp7 = load i32 * %tmp0
ret i32 %tmp7
}
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
%a_tmp = alloca i32
store i32 %a, i32 * %a_tmp
%b_tmp = alloca i32
store i32 %b, i32 * %b_tmp
%y = alloca i32
store i32 10, i32 * %y
ret i32 1
}
declare i32 @printf (i8 *, ...)
declare i8 * @malloc (i32)
```



Herança: acesso aos atributos da classe pai

- Primeiro elemento da classe é sua classe pai

```
class Teste {  
    public static void main(String[] a){  
        System.out.println(new Matematica().Soma(1,2));  
    }  
}  
  
class Matematica extends Complex {  
    int x;  
    public int Soma(int a, int b){  
        i = 10;  
        return 10;  
    }  
}  
  
class Complex {  
    int i;  
}
```

```
@.formatting.string = private constant [4 x i8] c"%d\0A\00"
%class.Matematica = type { %class.Complex, i32 }
%class.Teste = type { }
%class.Complex = type { i32 }
define i32 @main() {
entry0:
...
%tmp3 = call i8* @malloc ( i32 %tmp2 )
%tmp1 = bitcast i8* %tmp3 to %class.Matematica*
%tmp4 = call i32 @_Soma_Matematica(%class.Matematica * %tmp1, i32 1, i32 2)
%tmp5 = getelementptr [4 x i8] * @.formatting.string, i32 0, i32 0
%tmp6 = call i32 (i8 *, ...)* @printf(i8 * %tmp5, i32 %tmp4)
%tmp7 = load i32 * %tmp0
ret i32 %tmp7
}
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
%a_tmp = alloca i32
store i32 %a, i32 * %a_tmp
%b_tmp = alloca i32
store i32 %b, i32 * %b_tmp
%tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0
%tmp1 = getelementptr %class.Complex * %tmp0, i32 0, i32 0
store i32 10, i32 * %tmp1
ret i32 10
}
```

Não é ponteiro!

```
@.formatting.string = private constant [4 x i8] c"%d\\0A\\00"
%class.Matematica = type { %class.Complex, i32 }
%class.Teste = type { }
%class.Complex = type { i32 }
define i32 @main() {
entry0:
...
}
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
    %a_tmp = alloca i32
    store i32 %a, i32 * %a_tmp
    %b_tmp = alloca i32
    store i32 %b, i32 * %b_tmp
    %tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0
    %tmp1 = getelementptr %class.Complex * %tmp0, i32 0, i32 0
    store i32 10, i32 * %tmp1
    ret i32 10
}
declare i32 @printf (i8 *, ...)
declare i8 * @malloc (i32)
```

Herança: acesso aos métodos da classe pai

- Todo método é uma função global em LLVM-IR
- Encontrar qual classe contém o método na SymTab

```
class Teste {  
    public static void main(String[] a){  
        System.out.println(new Matematica().Soma(1,2));  
    }  
}  
  
class Matematica extends Complex {  
    int x;  
    public int Soma(int a, int b){  
        return this.Pow(1,2);  
    }  
}  
  
class Complex {  
    int i;  
    public int Pow(int n, int x){  
        return 0;  
    }  
}
```

```
@.formatting.string = private constant [4 x i8] c"%d\0A\00"
%class.Matematica = type { %class.Complex, i32 }
%class.Teste = type { }
%class.Complex = type { i32 }
define i32 @main() {
entry0:
...
}
define i32 @_Soma_Matematica(%class.Matematica * %this, i32 %a, i32 %b) {
entry0:
%a_tmp = alloca i32
store i32 %a, i32 * %a_tmp
%b_tmp = alloca i32
store i32 %b, i32 * %b_tmp
%tmp0 = getelementptr %class.Matematica * %this, i32 0, i32 0
%tmp1 = call i32 @_Pow_Complex(%class.Complex * %tmp0, i32 1, i32 2)
ret i32 %tmp1
}
define i32 @_Pow_Complex(%class.Complex * %this, i32 %n, i32 %x) {
entry0:
%n_tmp = alloca i32
store i32 %n, i32 * %n_tmp
%x_tmp = alloca i32
store i32 %x, i32 * %x_tmp
ret i32 0
}
declare i32 @printf (i8 *, ...)
declare i8 * @malloc (i32)
```

Herança: o que acontece neste código?

```
class Teste{
    public static void main(String[] a){
        System.out.println(new Zoo().Start());
    }
}
class Zoo {
    public int Start(){
        Tiger t1;
        t1 = new Tiger();
        return this.getInfo(t1);
    }
    public int getInfo(Animal c){
        return c.getWeight();
    }
}
class Tiger extends Animal{
    public int getWeight(){
        return 20;
    }
}
class Animal{
    public int getWeight(){
        return 0;
    }
}
```

Imprime 0 ou 20?

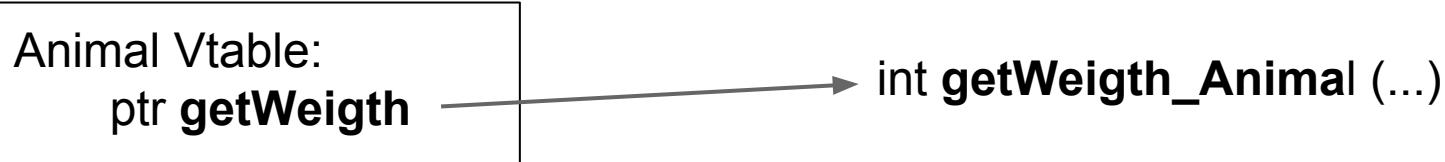
E em C++?

Herança: métodos virtuais

- Em Java: 20
- Em C++: 0
- Em C++ mudando o método **getWeigth** em **Animal** para virtual: 20
- Apesar de Java não ter métodos virtuais, quando há métodos repetidos em classes parentes, Java se comporta como C++ com métodos virtuais.
- **Como resolver isso em LLVM-IR?**

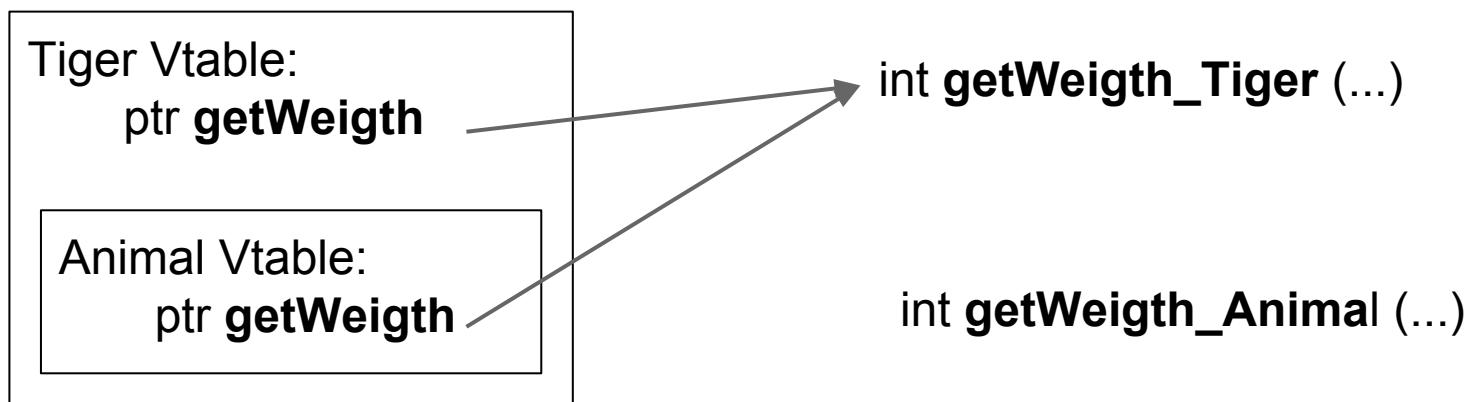
Virtual Method Table: Vtable

- Objetos possuem uma tabela com ponteiros para as funções



Virtual Method Table: Vtable em LLVM-IR

- **Lembrem:** Primeiro campo de uma classe filha é um objeto da classe pai
- O construtor do Objeto liga corretamente os ponteiros as funções



Virtual Method Table: Vtable

- LLVM-IR não tem suporte à classe, mas é possível implementar, como já vimos.
- LLVM-IR também não suporta Vtable, mas também é possível implementar.
- C também não suporta classe nem Vtable, mas é possível implementar ambos.
 - Exemplos interessantes: <http://milotshala.wordpress.com/2012/02/21/virtual-functions-in-c/>