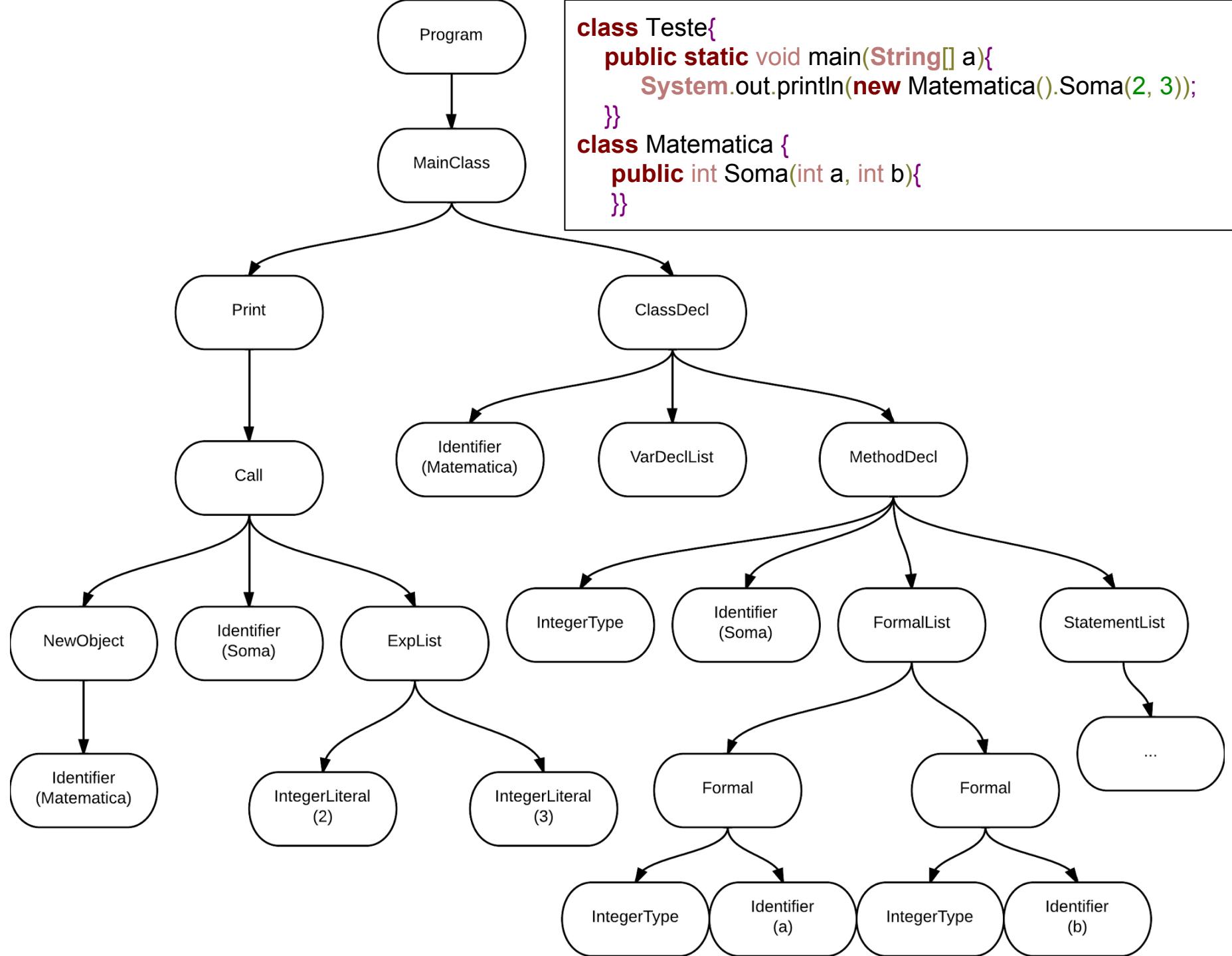


MiniJava/LLVM

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*
 - ...

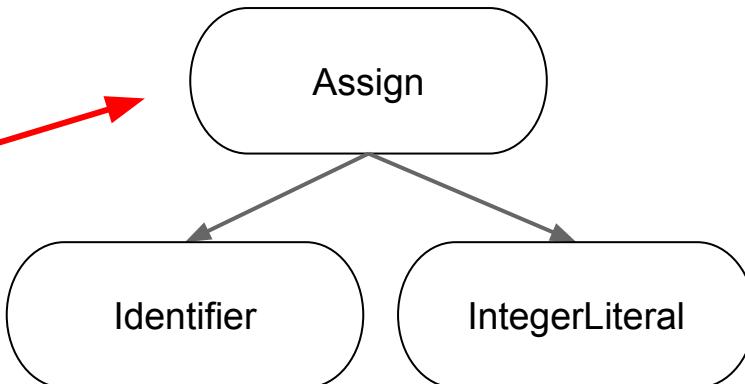


Minijava para LLVM

- A classe **Ilvm/Codegen.java** percorre a AST do programa gerando LLVM-IR
- Alguns métodos já estão implementados como exemplo
- Há comentários na classe Codegen detalhando o funcionamento e a implementação do pacote

Já posso emitir código a partir da AST?

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

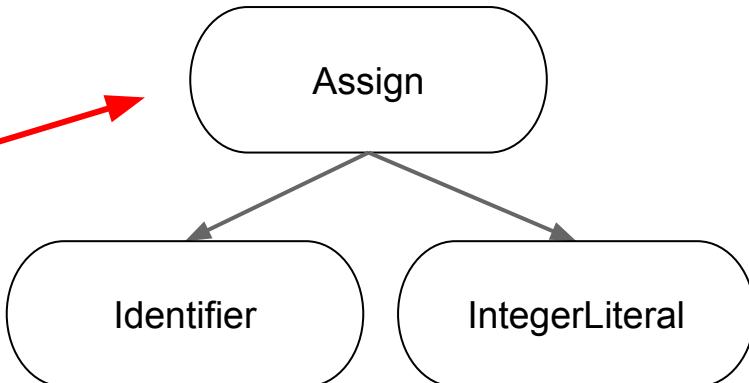


%class.Matematica = type { i32 } ; estrutura da classe

```
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;  
    }  
}
```

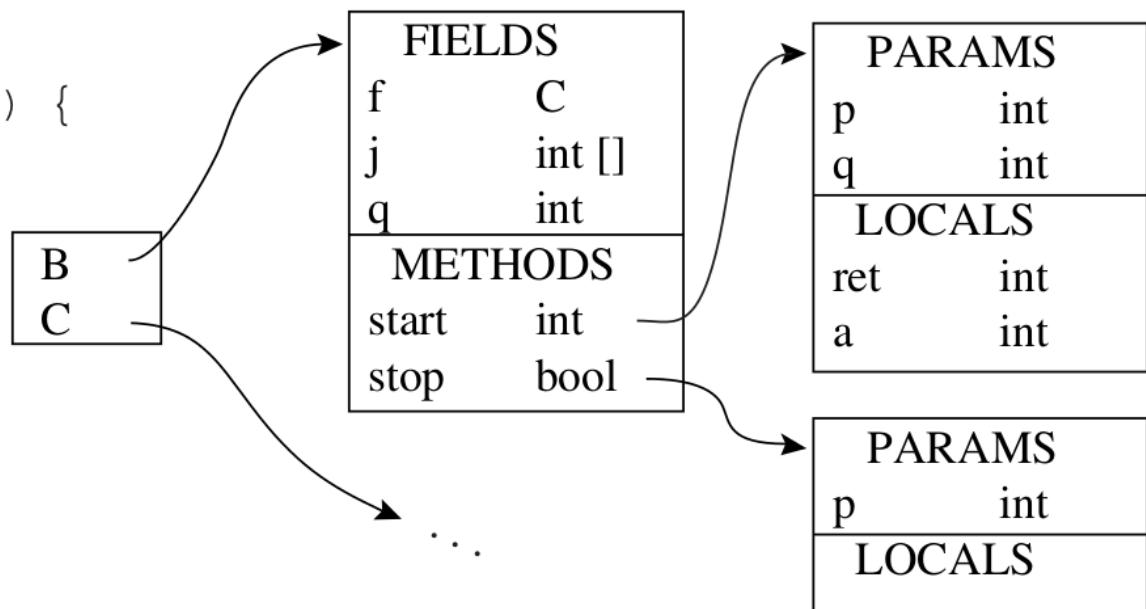


%class.Matematica = type { i32 } ; estrutura da classe

```
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;
    }
}
```

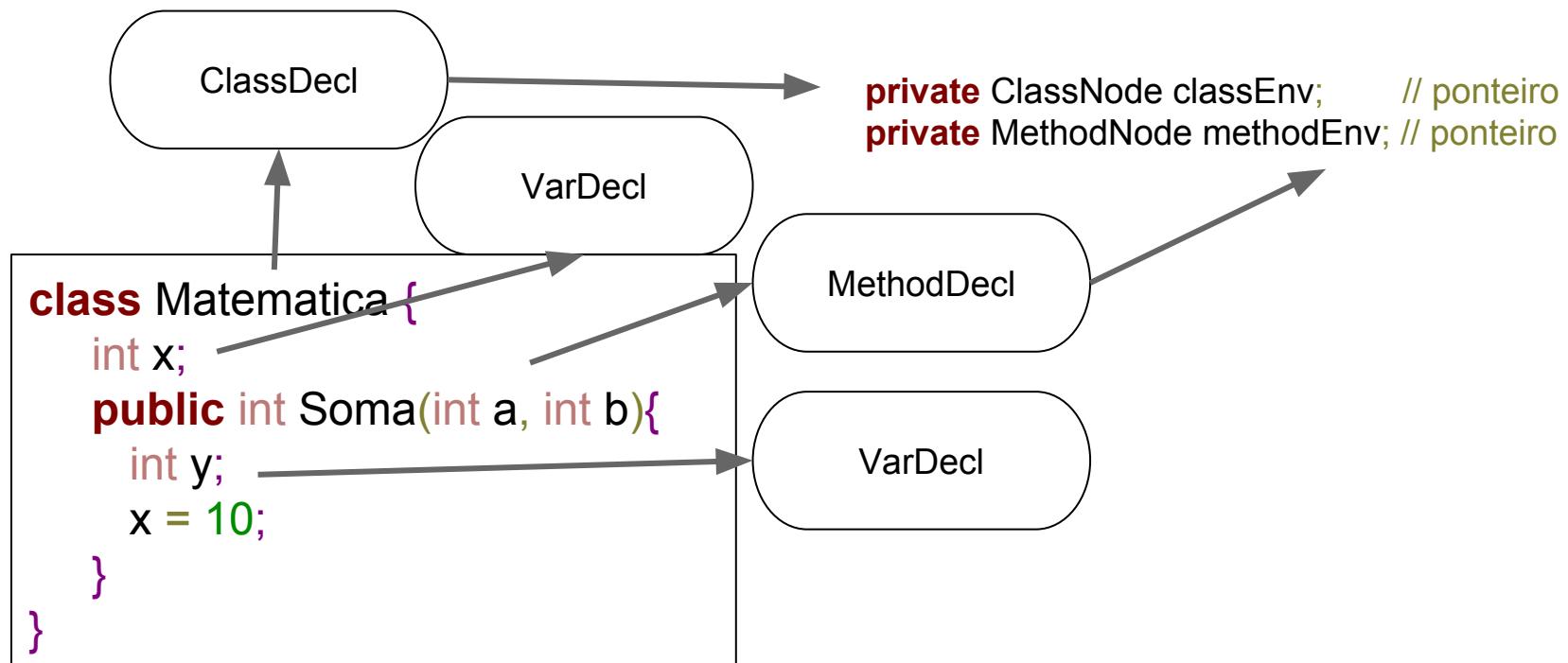
Essa tabela é suficiente?

- 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
    }
}
```