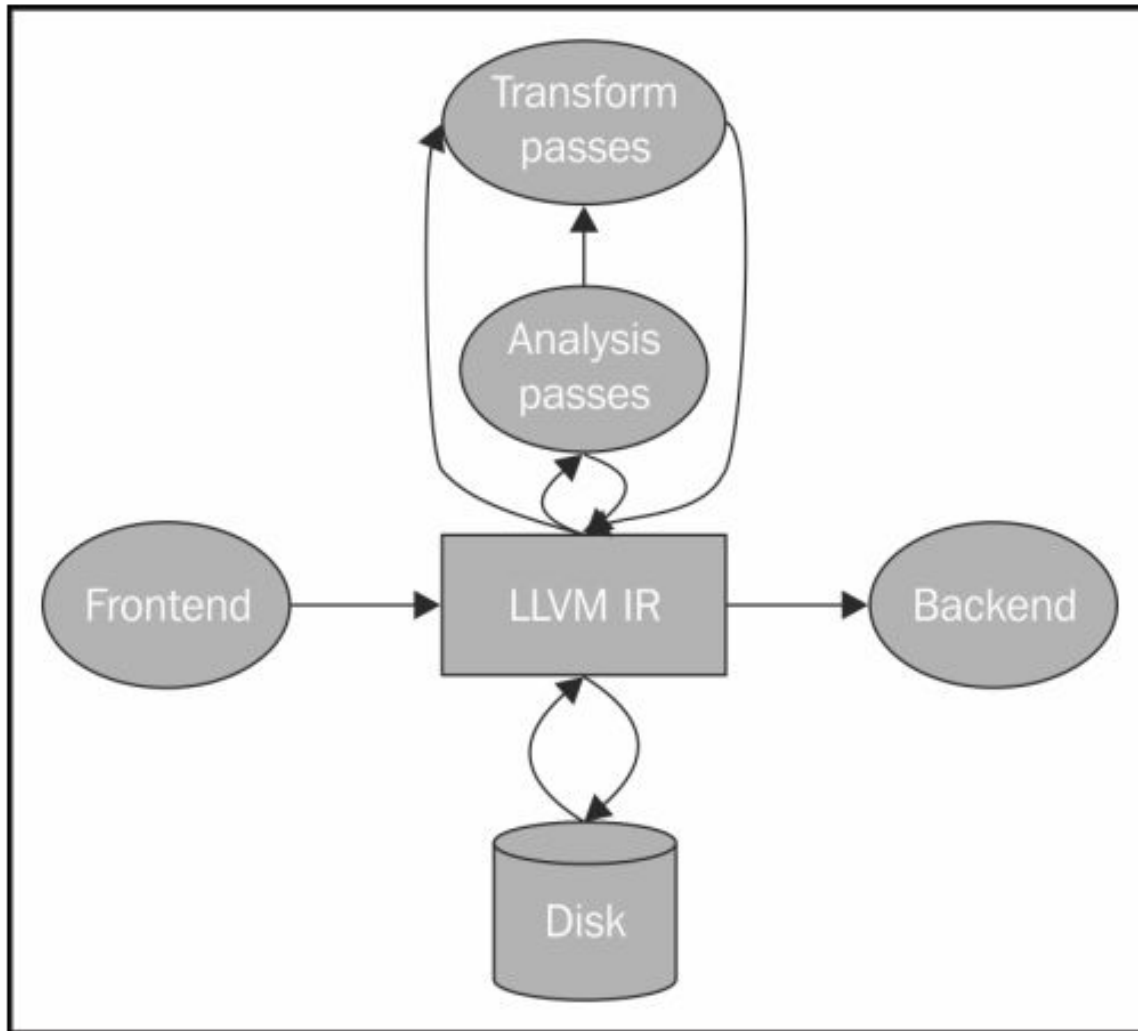


LLVM Pass e Core Libraries

Projeto 3

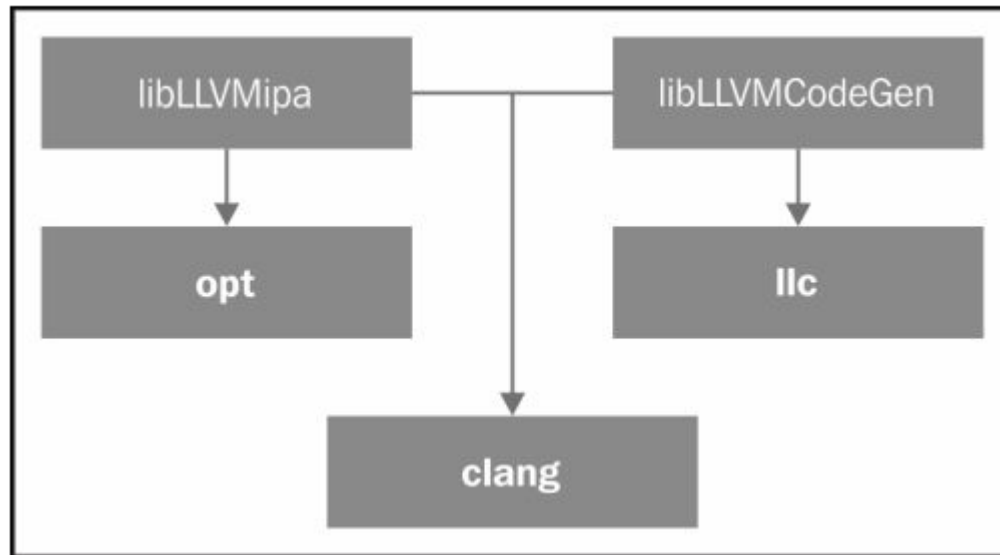
Relembrando...



Ferramentas LLVM

- Várias ferramentas "*standalone*"
 - llc, lli, clang, opt, prof, tblgen, clang-check, clang-modernizer, clang-format, ...
- Ferramentas incorporam bibliotecas modulares reutilizáveis
- Diferentes ferramentas compartilham a mesma biblioteca
 - Resultados consistentes
 - Facilidade de desenvolvimento e depuração

Por que o Clang usa essas libs?

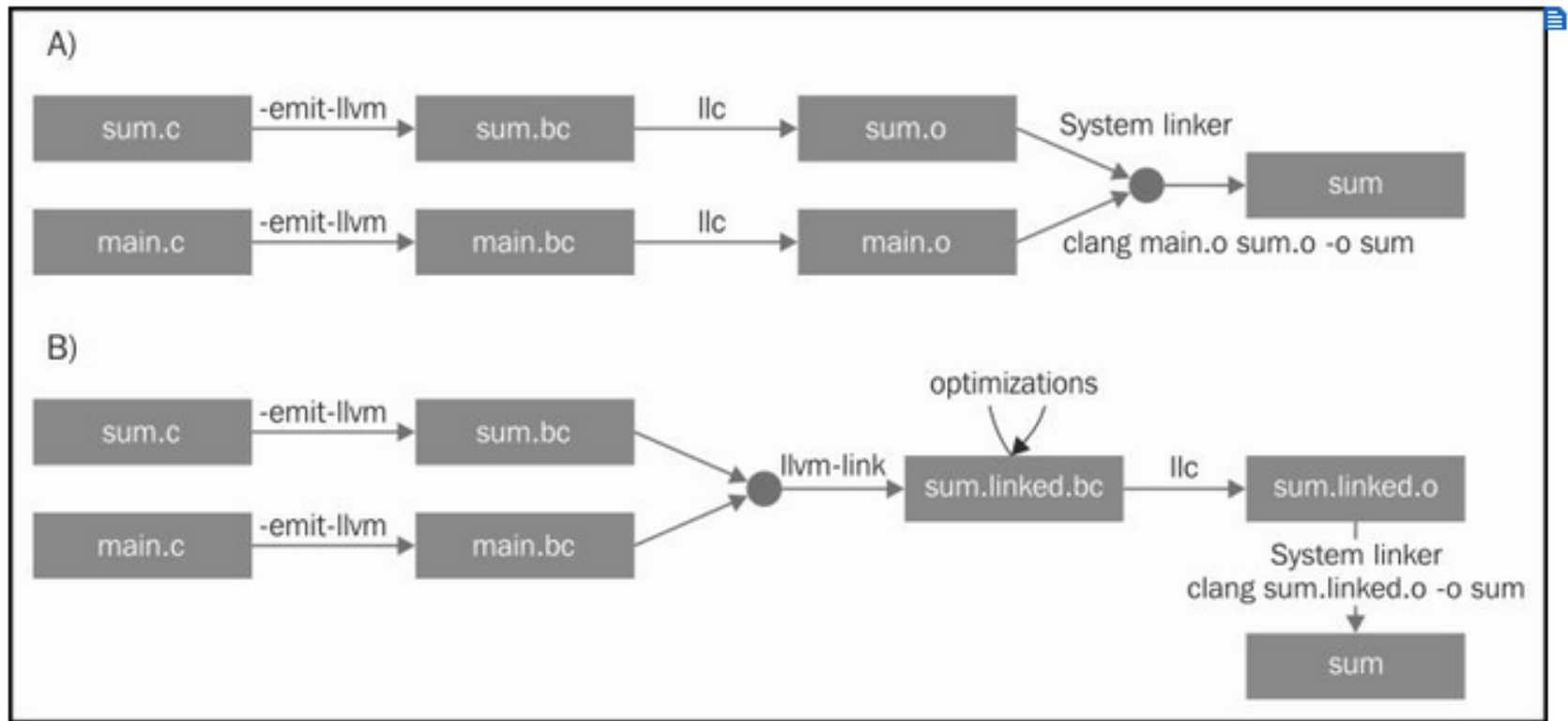


- O **Clang** respeita as mesmas *flags* do **gcc**

clang main.c sum.c -o sum

- Além de *frontend*, **clang** pode ser um *compiler driver*
 - Conduz todas as etapas de compilação

Usando algumas ferramentas "standalone"



- A) compila os bitcodes para objetos da arquitetura alvo com o **llc**, e usa o *linker* do sistema para gerar o executável
- B) Liga os dois bitcodes em um bitcode final com o **llvm-link**, otimiza com o **opt**, e depois usa o **llc** e o *linker* do sistema para gerar o executável

Usando algumas ferramentas "standalone"

- A)

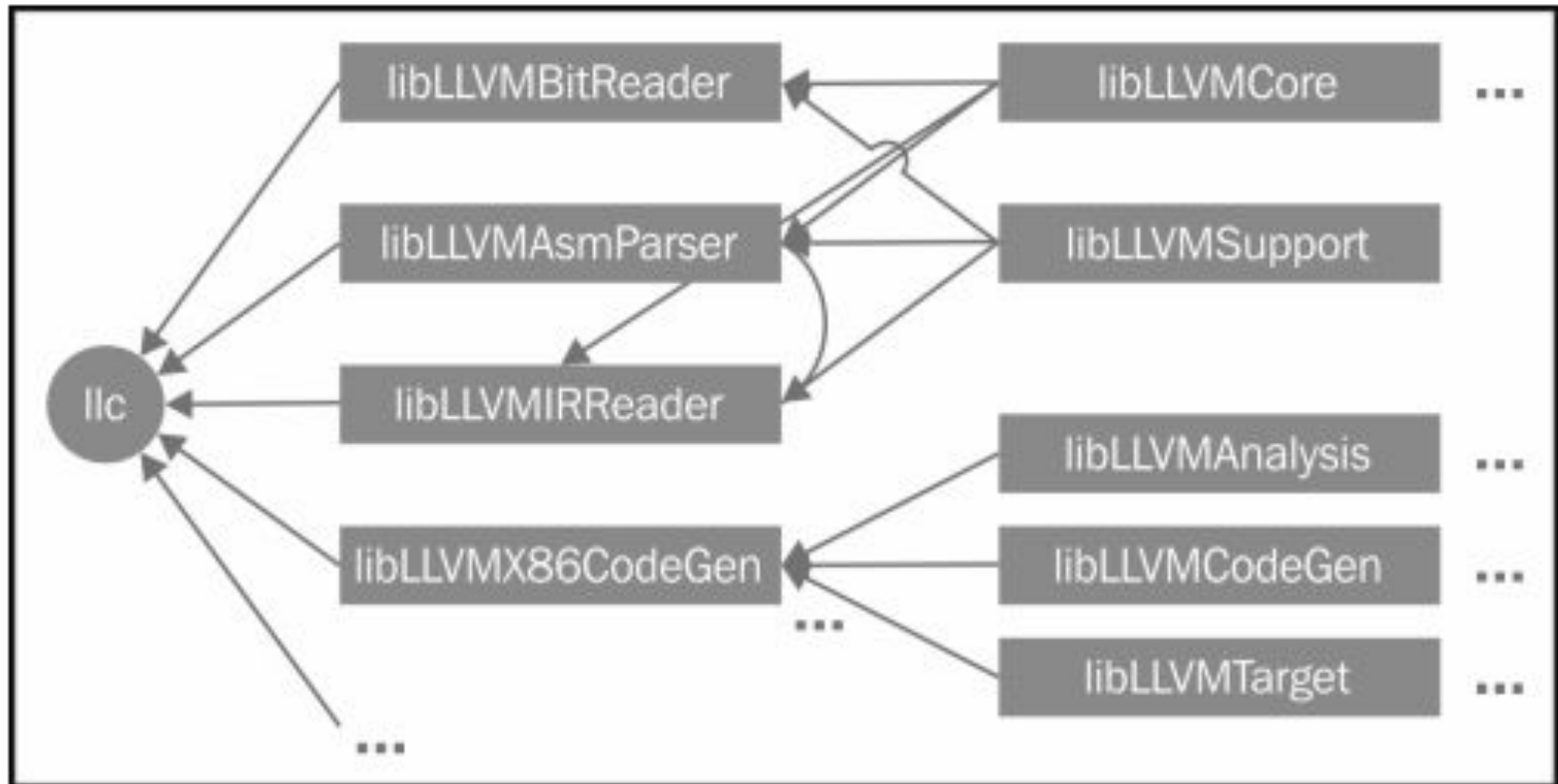
```
$ llc -filetype=obj main.bc -o main.o  
$ llc -filetype=obj sum.bc -o sum.o  
$ clang main.o sum.o -o sum
```
- B)

```
$ llvm-link main.bc sum.bc -o sum.linked.bc  
$ llc -filetype=obj sum.linked.bc -o sum.linked.o  
$ clang sum.linked.o -o sum
```

Bibliotecas básicas

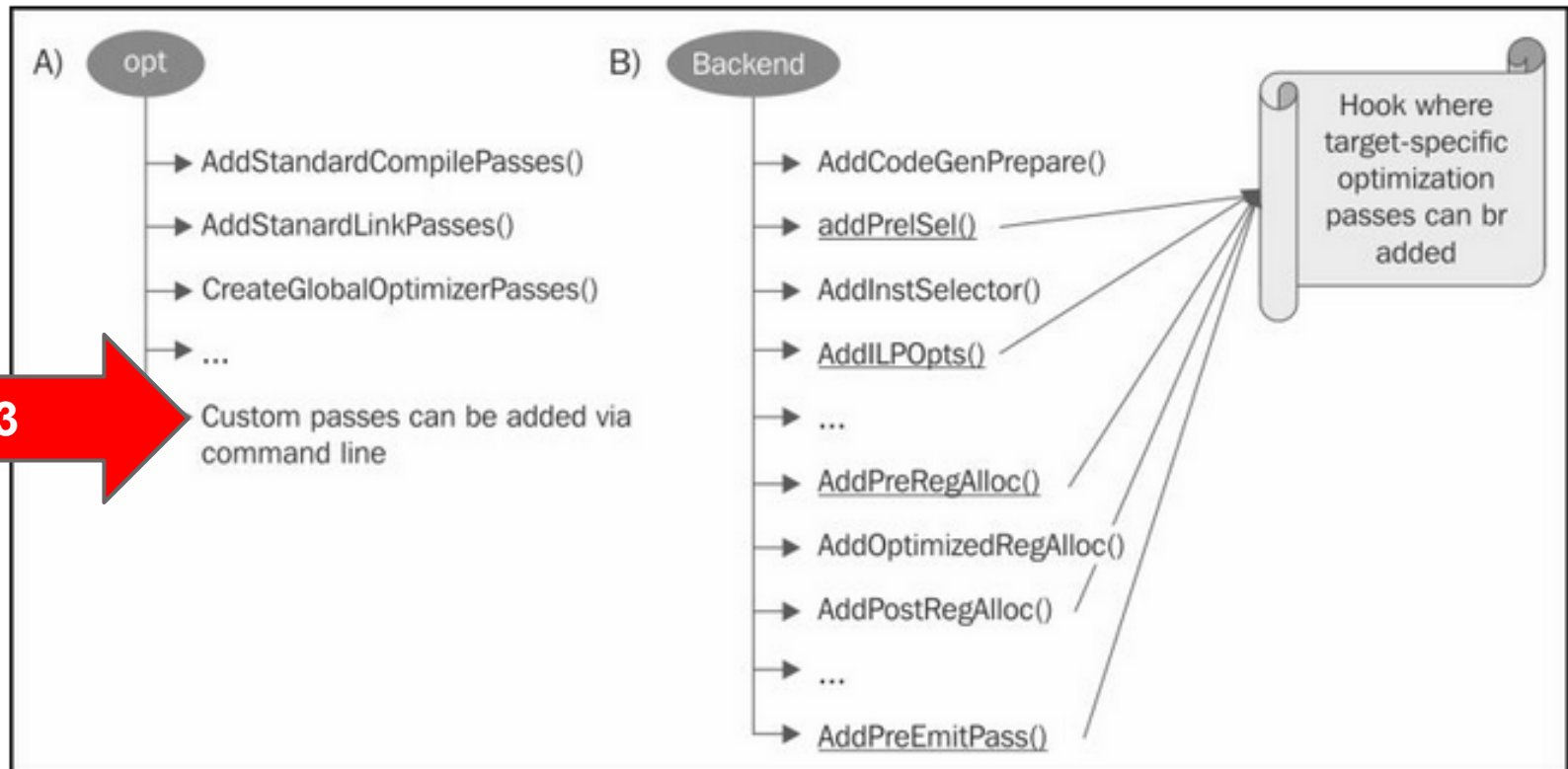
- **libLLVMCore:** contém a lógica do LLVM-IR
- **libLLVMAnalysis:** diversas análises. Algumas serão estudadas neste curso
- **libLLVMX86CodeGen:** backend x86
- **libclang:** funcionalidades de *frontend*, como *AST* e *code completion*
- **libclangDriver:** classes que preparam e organizam os passos de compilação
- **libLLVMSupport:** utilitários diversos, como *parsing* de linha de comando, depuração, manipulação de string, dentre outros.

Bibliotecas básicas



Otimizações (Passes) em LLVM

- Podem ser inseridas em diferentes fases na compilação de um programa
 - Otimização de *frontend* (**clang**)
 - Otimização de Linker (**opt**)
 - Otimização de Backend dependente de arquitetura



Usando o "opt"

- Pode-se utilizar os argumentos do GCC
opt -O3 sum.bc -o sum-opt.bc
- Compile-time optimizations:
opt -std-compile-opts sum.bc -o sum-opt.bc
- Link-time optimizations:
llvm-link file1.bc file2.bc file3.bc -o all.bc
opt -std-link-opts all.bc -o all-opt.bc
- Individual Pass
opt sum.bc -mem2reg -o sum-opt.bc -time-passes

Pass API

- Uma otimização em LLVM normalmente é composta de:
 - *Analysis Pass*
 - *Transform Pass*
- É possível criar dependência entre passo e análise
- Ao implementar uma Otimização, deve-se escolher a granularidade:
 - ModulePass
 - FunctionPass
 - BasicBlockPass
 - LoopPass
 - ...

Projeto 3

Códigos de Apoio

Projeto:

- Implementar a Liveness Analysis
- Implementar duas otimizações DCE:
 - Usando seu Liveness (Appel, Ed. 2, Pag. 360)
 - Usando Def-Use Chain do LLVM (Appel, Ed. 2, Pag 417)
- **Material:**
 - [dce-pass-p3](#)
 - Um passo que imprime o nome das funções (-hello)
 - Um passo que emite o CFG do programa (-printCFG)
 - Meu Liveness iniciado (o .h está completo)
 - Testado com LLVM 3.5 e 3.6
 - [dce-tests](#)
 - Conjunto de testes

Estrutura do pacote dce-pass

- p3:
 - Diretório dos fontes das suas otimizações
 - Cada otimização/análise deve ter seu próprio arquivo .cpp
- Release:
 - Diretório que contém o objeto **P3.so** gerado após a compilação
 - **P3.so** contém todas as suas otimizações
- Makefile:
 - Compila suas otimizações

Comandos úteis

Para compilar suas otimizações presentes em "p3"

make

Para limpar o projeto

make clean

Compilando arquivos .c para LLVM-IR

clang -emit-llvm -S input.c -o output.ll

Para utilizar a otimização que está em

"p3/Hello.cpp" no código "dce-tests/minijava/BubbleSort.ll"

**opt -S -load Release/P3.so -hello \
../dce-tests/minijava/BubbleSort.ll > BubbleSort.opt.ll**

Primeiros passos

- Conferir se existe o programa 'llvm-config' na sua máquina

```
$ llvm-config --version
```

- Baixar e descompactar o '[dce-pass-p3](#)'

- Compilar os passos

```
$ make
```

- Aplicar o passo no programa BubbleSort

```
$ opt -S -load Release/P3.so -hello \  
  ../dce-tests/minijava/BubbleSort.ll -o BubbleSort.opt.ll
```

- Executar o programa otimizado

```
$ lli BubbleSort.opt
```


Dicas de Implementação

- Ler a documentação da API C++ do LLVM - Parte 1
 - [Core Classes](#)
 - [Value class](#)
 - [User class](#)
 - [Function class](#)
 - [Instruction class](#)
 - [BasicBlock class](#)
 - [Iterando Instructions e BBs dentro de Functions](#)
- **BasicBlock** (labels) é subclasse de **Value**
- **Instructions** e **Functions** são subclasses de **User**
- **User** é subclasse de **Value** e contém uma [lista de operandos](#)
- **SSA**: Única definição
 - Um operando de uma instrução é:
 - um apontador para a sua instrução de definição; ou
 - um argumento

```
%tmp0 = load i32* %a
%tmp1 = load i32* %b
%tmp2 = add i32 %tmp0, %tmp1
```

Instrução %tmp2

Instruction *I = /* instrução de um Basic Block */

```
errs() << "INS:" << *i << '\n';
```

```
for (Instruction::op_iterator o = i->op_begin(), oe = i->op_end(); o != oe; ++o) {
```

```
    Value *v = *o;
```

```
    if (isa<Instruction>(*v) || isa<Argument>(*v)){
```

```
        errs() << "OPE:" << *v << '\n';
```

```
    }
```

```
}
```

Testa se o operando é ponteiro para uma instrução ou para um argumento

Output:

```
INS: add i32 %tmp0, %tmp1
OPE: %tmp0 = load i32* %a
OPE: %tmp1 = load i32* %b
```

Dicas de Implementação

- Ler a documentação da API C++ do LLVM - Parte 2
 - [isa, cast e dyn_cast](#)
 - [mayHaveSideEffects](#)
 - [eraseFromParent](#)
 - [def-use e use-def chains](#)
- Leiam o manual sobre [implementação de passos](#)
- Use [C++ STL](#) para melhorar sua produtividade em C++
 - Contém **sets**, **maps**, **lists** e **vectors**
- O padrão [C++ 11](#) é necessário para compilar o LLVM 3.5 e também pode ser utilizado em seu passo

Dead Code Elimination

- Uma instrução é trivialmente viva:
 - se seu comportamento gerar efeitos colaterais (*mayHaveSideEffects*)
 - se for um terminador (*TerminatorInst*)
 - se for instrução de debugging (*DbgInfoIntrinsic*)
 - se for instrução de exceção (*LandingPadInst*)
 - se for usada por outra instrução que está viva
- Códigos úteis:
 - Meu Liveness iniciado que está no pacote (LLVM-3.6)
 - Passo [Liveness](#) **incompleto** (**não** funciona no LLVM-3.6)
 - Passo [CFG Analysis](#) **incompleto** (**não** funciona no LLVM-3.6)

Analise Externa ao Passo

- Divisão de tarefa entre vários passos/análises
- Deixa o código mais organizado
- Mantém seu código dentro do padrão LLVM

DCE.cpp

```
#include "Liveness.h"

using namespace llvm;

namespace {

    struct DCE : public FunctionPass {

        DCE() : FunctionPass(ID) {}

        virtual bool runOnFunction(Function &F) {
            Liveness &L = getAnalysis<Liveness>();
            ...
        }

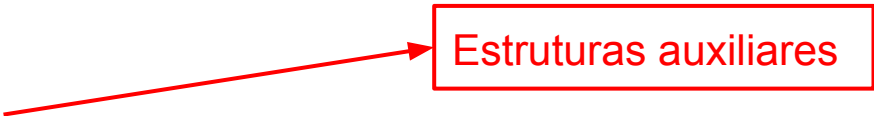
        virtual void getAnalysisUsage(AnalysisUsage &AU) const {
            AU.addRequired<Liveness>();
        }
    }
}
```

L pode chamar métodos de Liveness

Chama o método Run de Liveness

Liveness.h

```
namespace llvm {  
    struct LivenessInfo {  
        std::set<const Value *> use;  
        std::set<const Value *> def;  
        std::set<const Value *> in;  
        std::set<const Value *> out;  
    };  
    class Liveness : public FunctionPass {  
    private:  
        DenseMap<const Instruction*, LivenessInfo> iLivenessMap;  
        DenseMap<const BasicBlock*, LivenessInfo> bbLivenessMap;  
        DenseMap<const Instruction*, int> instMap;  
    public:  
        static char ID;  
        Liveness() : FunctionPass(ID) {}  
        virtual bool runOnFunction(Function &F); // implementação no Liveness.cpp  
        void computeBBDefUse(Function &F);  
        void computeBBInOut(Function &F);  
        ...  
    };  
}
```



Estruturas auxiliares

Liveness.cpp

```
void Liveness::computeBBInOut(Function &F)
{
    ...
}
```

```
bool Liveness::runOnFunction(Function &F)
{
    computeBBDefUse(F);
    computeBBInOut(F);
    computeIInOut(F);
    return false;
}
```

```
char Liveness::ID = 0;
RegisterPass<Liveness> X("liveness", "Live vars analysis", false, false);
```



Necessário estar no .cpp

Para saber mais

- <http://llvm.org/docs/ProgrammersManual.html>
- Figuras retiradas do Livro:
 - Bruno Cardoso, Rafael Auler. *"Getting Started with LLVM Core Libraries"*, 2014
- Baixem os Slides aqui: <http://www.ic.unicamp.br/~maxiwell/cursos/mc911>