Um pouco de LLVM

Após o front-end

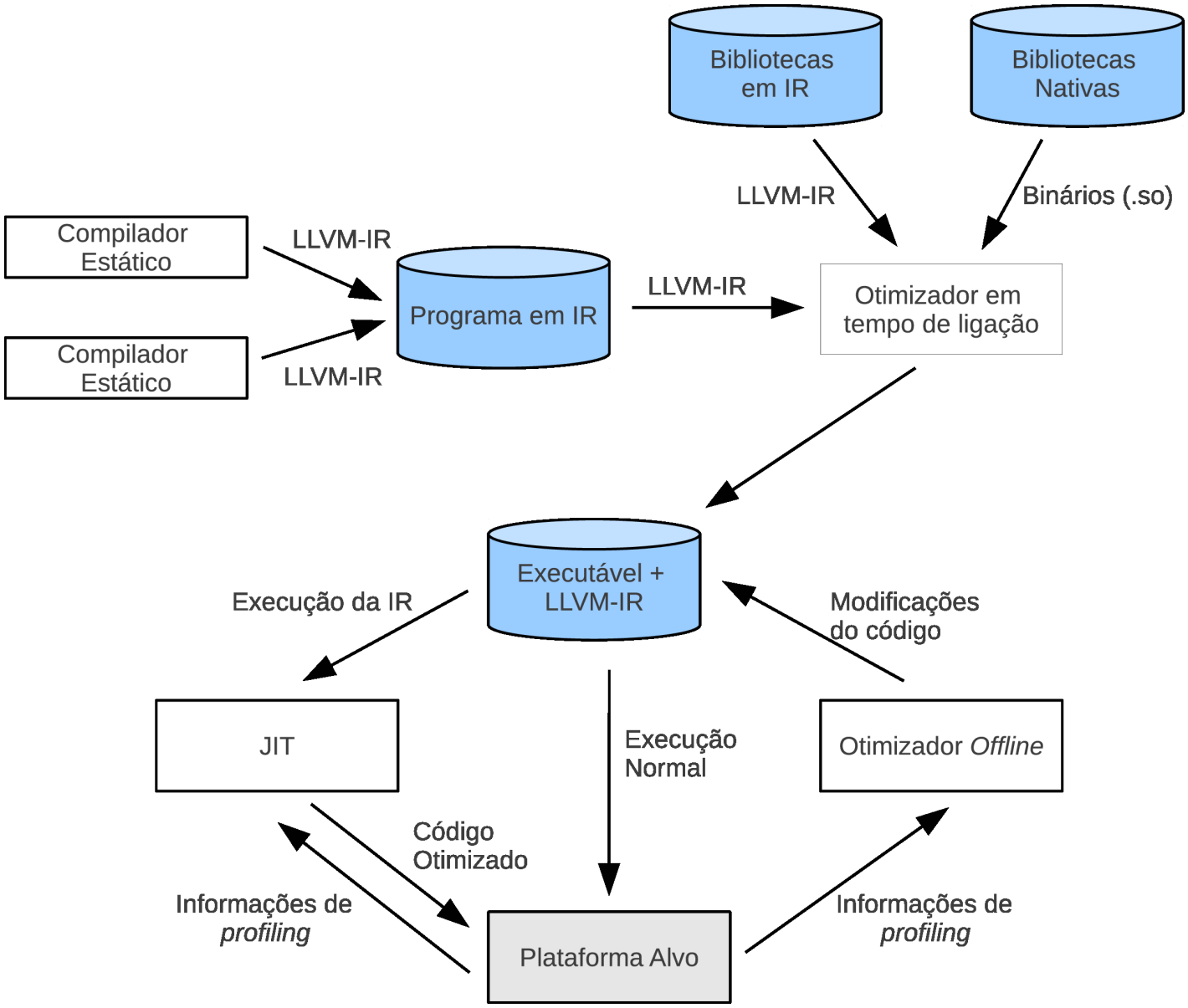
- Depois do léxico e do sintático, o compilador:
  - sabe que o texto do programa está OK
  - constrói a *abstract syntax tree (AST)*
- O que falta para gerar o executável x86?

Representação Intermediária (IR)

- Linguagem independente
- Deve ser facilmente construída a partir da AST
- Não pode conter construções de alto nível
- Utilizada para realizar otimizações

LLVM - Low Level Virtual Machine

- Atua no *back-end* da compilação
  - Da IR até o executável
  - CLANG é o *front-end* padrão
- Várias otimizações em tempo de compilação e execução
- www.llvm.org


Bibliotecas em IR
Bibliotecas Nativas
LLVM-IR
Binários (.so)
Compilador Estático
LLVM-IR
Programa em IR
LLVM-IR
Otimizador em tempo de ligação
Compilador Estático
LLVM-IR
Executável + LLVM-IR
Execução da IR
Modificações do código
JIT
Execução Normal
Otimizador Offline
Código Otimizado
Informações de profiling
Plataforma Alvo
Informações de profiling

Nesta fase do curso...

1. Apresentar a IR do LLVM
2. Estudar algumas IR's geradas com o Clang
3. Apresentar as classes em Java responsáveis pelo Front-End (até a AST) da linguagem minijava.
4. **Projeto 2: Gerar LLVM-IR para programas em minijava**
5. **Projeto 3: Implementar otimizações na LLVM-IR**

Instruções

```
; Results in a poison value.
%var1 = sub i32 2, %var0
%cmp = icmp eq i32 %D, 0
br i1 %cmp, label %true, label %end

true:
%var2 = load i32* %i

end:
store i32 %var1, i32* %result
```

Vamos codificar...

Comandos úteis

```
# Usando o Clang para gerar executável x86
$ clang main.c -o main

# Clang front-end emitindo arquivo texto LLVM-IR
$ clang main.c -S -emit-llvm -o main.s

# Monta o arquivo texto LLVM-IR (main.s) em um arquivo
# bytecode LLVM-IR (main.bc)
$ llvm-as -f main.s -o main.bc

# Interpreta o bytecode
$ lli main.bc
```

Comandos úteis

```
# A partir do bytecode LLVM-IR (main.bc) é possível gerar
# assembler x86 (main_x86.s)
$ llc main.bc -o main_x86.s

# Gerando código objeto com o GAS
$ as main.s -o main.o

# Linkando as libs no objeto e gerando executável x86
$ gcc main.o -o main

# Executando
$ ./main
```

SSA - Static Single Assignment

- Em LLVM-IR, um registrador não pode ser assinalado mais que uma vez:

Um novo registrador para cada resultado!

- Essa restrição facilita as otimizações

```
define i32* @foo(%struct.ST* %s) {
  %t1 = getelementptr %struct.ST* %s, i32 1
  %t2 = getelementptr %struct.ST* %t1, i32 0, i32 2
  %t3 = getelementptr %struct.RT* %t2, i32 0, i32 1
  %t4 = getelementptr [10 x [20 x i32]]* %t3, i32 0, i32 5
  %t5 = getelementptr [20 x i32]* %t4, i32 0, i32 13
  ret i32* %t5
}
```

Tipos em LLVM

- LLVM é fortemente tipado: todas as instruções e todos os valores são tipados.

- Primitivos:
  - **i1, i2, … i8, ... i16, ... i32**
  - **label, void**
  - **float e double** (não utilizado em nosso projeto)

- Derivados:
  - Array: **[40 x i32]**
  - Pointers: **[4 x i8]***
  - Structures: **{ i32, (i32)*, i1 }**
  - Function - <tipo_retorno> (<parametro_list>): **i32 (i32)**

`getelementptr`

Detalhes em: http://llvm.org/docs/GetElementPtr.html

- <result> = **getelementptr** <pty>* <ptrval> {, <ty> <idx>}*
- O primeiro argumento sempre é um ponteiro
- Construção mais complexa do LLVM-IR
- Serve para acessar estruturas a partir do primeiro operando

Código em C

```
struct T {
  int f1;
  int f2;
};
void foo(struct T *PT) {
  PT[0].f1 = PT[1].f1 + PT[2].f2;
}
```

foo em LLVM-IR

```
void %foo(%struct.T* %PT) {
entry:
  %tmp = getelementptr %struct.T* %PT, i32 1, i32 0
  %tmp5 = load i32* %tmp
  %tmp6 = getelementptr %struct.T* %PT, i32 2, i32 1
  %tmp7 = load i32* %tmp6
  %tmp8 = add i32 %tmp7, %tmp5
  %tmp9 = getelementptr %struct.T* %PT, i32 0, i32 0
  store i32 %tmp8, i32* %tmp9
  ret void
}
```

Código em C

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

foo em LLVM-IR

```
define i32* @foo(%struct.ST* %s) {
%tmp = getelementptr %struct.ST* %s, i32 1, i32 2, i32 1,
       i32 5, i32 13
ret i32 $tmp
}
```

Código em C

```
struct RT {
  char A;
  int B[10][20];
  char C;
};
struct ST {
  int X;
  double Y;
  struct RT Z;
};
int *foo(struct ST *s) {
  return &s[1].Z.B[5][13];
}
```

foo em LLVM-IR

```
define i32* @foo(%struct.ST* %s) {
  %t1 = getelementptr %struct.ST* %s, i32 1
  %t2 = getelementptr %struct.ST* %t1, i32 0, i32 2
  %t3 = getelementptr %struct.RT* %t2, i32 0, i32 1
  %t4 = getelementptr [10 x [20 x i32]]* %t3, i32 0, i32 5
  %t5 = getelementptr [20 x i32]* %t4, i32 0, i32 13
  ret i32* %t5
}
```

Minijava para LLVM

- A classe **llvm/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