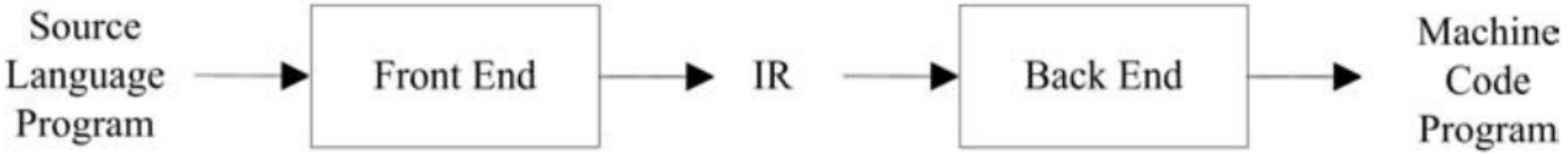
Geração de Código Intermediário

ÉFREN L. SOUZA

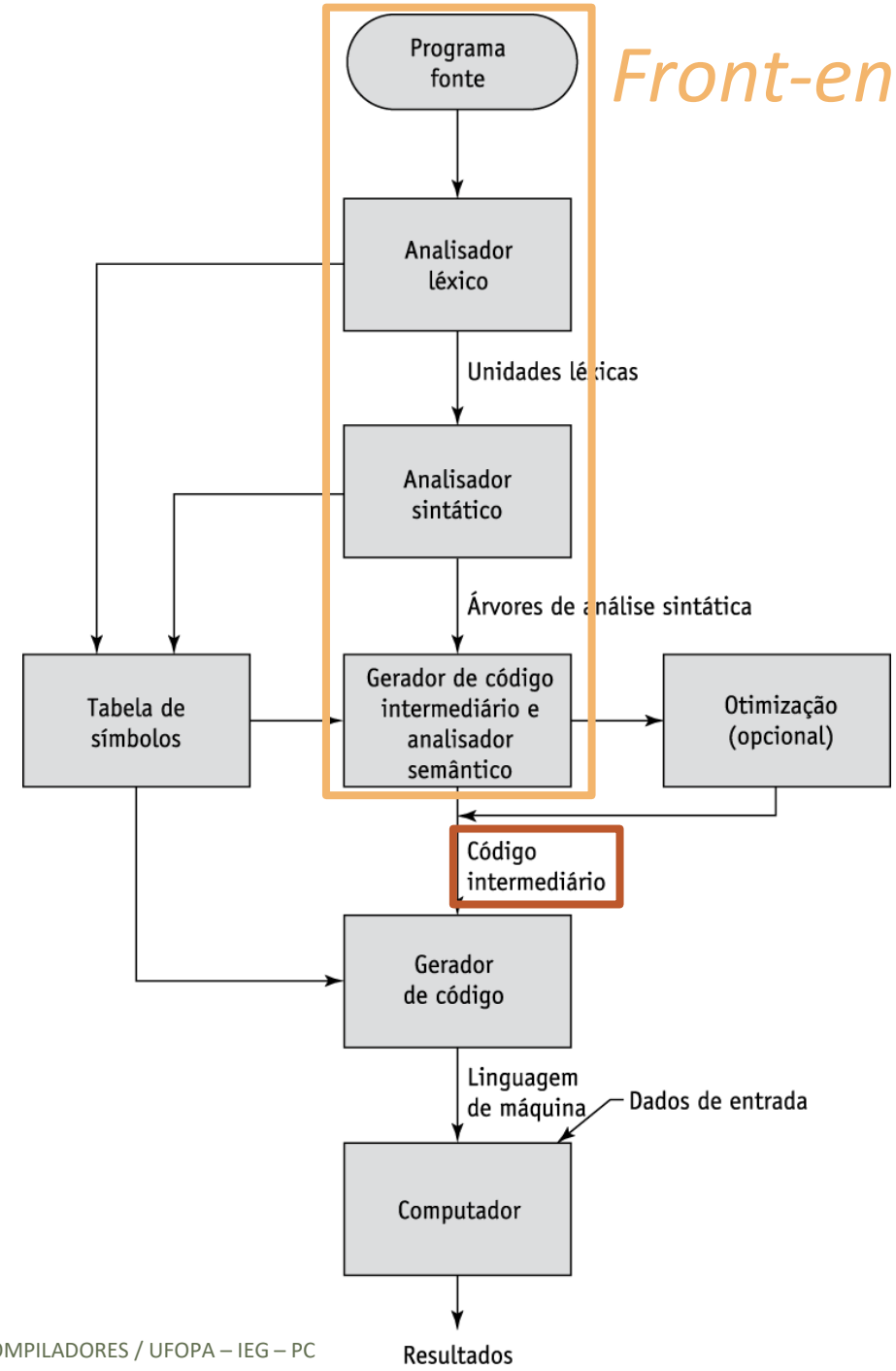
Front-End

O objetivo do *front-end* é produzir uma representação intermediária do programa fonte



Processo de compilação

- O *front-end* produz a representação intermediária
- O *back-end* produz o programa objeto



Representações Intermediárias

- No processo de traduzir um programa fonte, o compilador pode produzir uma ou mais representações intermediárias
  - Representações Hierárquicas
  - Representações Lineares

Árvores Sintáticas

- Usadas na análise sintática e semântica
- Os nós dessa árvore representam construções do programa

$$WHILE \longrightarrow \textbf{while}\ (EXPR)\ STMT$$

Código de Três Endereços

- Sequência de etapas elementares do programa
  - Pode imaginar como um programa para uma máquina abstrata
- A razão deste nome vem do formato $x = y \; \boldsymbol{op} \; z$
  - Dois endereços para os operandos y e z
  - E um para o resultado x
- Deve ter duas propriedades importantes
  - Ser fácil de produzir a partir do programa fonte
  - Ser fácil de traduzir para o programa alvo

Código de Três Endereços

$$p = i + r * 60$$

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

C como Linguagem Intermediária

- O C é uma linguagem de alto nível (com recursos de baixo nível)
- Seu compilador é popular e flexível
- Ela é muito utilizada como linguagem intermediária
  - O *front-end* do C++ gera código C, usando o compilador do C como *back-end*

LLVM

LLVM (https://llvm.org/)

- LLVM não é um acrônimo, é o nome completo do projeto

- É um conjunto de ferramentas para compiladores
  - *LLVM intermediate representation* (LLVM-IR)
  - Gera código otimizado para diversas arquiteturas
  - Usa o Clang para traduzir C/C++ para LLVM-IR
  - Muitos outros recursos

LLVM-IR

- Busca ser a representação de código intermediário universal

- Pode ser usada de três formas:
  - Como uma IR na memória
  - Como uma IR em *bitcodes* no disco
  - Uma representação de *assembly* legível para humanos

Tipos

INTEIRO

`i1`	a single-bit integer.
`i32`	a 32-bit integer.
`i1942652`	a really big integer of over 1 million bits.

PONTO FLUTUANTE

`half`	16-bit floating point value
`float`	32-bit floating point value
`double`	64-bit floating point value
`fp128`	128-bit floating point value (112-bit mantissa)
`x86_fp80`	80-bit floating point value (X87)
`ppc_fp128`	128-bit floating point value (two 64-bits)

Literais

Tipos	Literais
`i1`	***`true`***, ***`false`***
`i32`	Padrão inteiro (e.g. `4`, `12`, `-25`)
`double`	Notação decimal (e.g. `123.421`) Notação exponencial (e.g. `1.23421e+2`)

Identificadores

- Uma cadeia de caracteres com o prefixo %
  - `%foo`
  - `%DivisionByZero`
  - `%a.really.long.identifier`

- O prefixo é usado para evitar conflitos de nomes entre palavras reservadas e identificadores

Identificadores Não Nomeados

- Um valor numérico com o prefixo %
  - %1
  - %2
  - %3
- Devem ser usados em sequência
- São usados pelo compilador para criar temporários sem conflitos com os nomes de variáveis
  - São como registradores virtuais

Algumas Operações Binárias

Operação	Inteiro	Ponto Flutuante
Soma	`add`	`fadd`
Subtração	`sub`	`fsub`
Multiplicação	`mul`	`fmul`
Divisão	`sdiv`	`fdiv`

$$result = op\ type\ op1, op2$$

$$\%2 = add\ i32\ 15, \%1$$

Acesso à Memória

- Para representar dados em memória, precisamos alocar (`alloca`) o espaço antes de usar
- Uma vez alocado, esse espaço pode ser alterado (`store`) e lido (`load`)

```
 %a = alloca i32                ;alocando memória para inteiro
store i32 2, i32* %a            ;colocando 2 no espaço alocado
%1 = load i32, i32* %a          ;lendo o valor da memória
```

Exemplo

PROGRAMA EM C

```
int main() {
  int a, b, c, d;
  a = 2; b = 4; c = 6;
  d = b*b - 4*a*c;
  return 0;
}
```

PROGRAMA EM LLVM-IR

```
define i32 @main() #0 {
  %a = alloca i32
  %b = alloca i32
  %c = alloca i32
  %d = alloca i32
  store i32 2, i32* %a
  store i32 4, i32* %b
  store i32 6, i32* %c
  %1 = load i32, i32* %b
  %2 = load i32, i32* %b
  %3 = mul i32 %1, %2
  %4 = load i32, i32* %a
  %5 = mul i32 4, %4
  %6 = load i32, i32* %c
  %7 = mul i32 %5, %6
  %8 = sub i32 %3, %7
  store i32 %8, i32* %d
  ret i32 0
}
```

Exercício

Escreva um programa em LLVM-IR que dado a aresta (l) de um cubo, calcule a sua área (A)

$$A = 6 \times l^2$$

Use valores do tipo *double*

```
define i32 @main() #0 {
  %L = alloca double
  store double 10.0, double* %L
  %1 = load double, double* %L
  %2 = fmul double 6.0, %1
  %3 = fmul double %2, %1
  %a = alloca double
  store double %3, double* %a
  ret i32 0
}
```

Instalando LLVM & Clang

```
$sudo apt-get install llvm-6.0
$sudo apt-get install clang-6.0
```

Executando um Código LLVM-IR

- O código LLVM-IR pode ser interpretado

```
$lli prog.ll
```

- Ele também pode ser compilador

```
$llc prog.ll -o prog.s
$gcc -s prog.s -o prog
```

Clang

- Trata-se de um *front-end* alternativo para C/C++
- Ele traduz programas dessas linguagens para LLVM-IR

```
$clang-6.0 prog.c -S -emit-llvm
```

DICA

Quando você não souber como implementar alguma coisa em LLVM-IR, implemente em C e depois traduza para LLVM-IR usando o Clang.

Iniciando a Geração de Código Intermediário em DL

Impontando o Projeto para o Eclipse

1. Clique no menu `File -> Import`
2. Selecione a opção `General -> Existing Projects into Workspace`
3. Marque a opção `Select archive file`
4. Selecione o arquivo do projeto `dl_short_3.zip`
5. Clique em `Finish`

Estado atual do projeto

- Faz a análise léxica
- Faz a análise sintática
  - Verifica sintaxe
  - Cria e imprime a árvore sintática
- Faz a análise semântica
  - Verifica a declaração de variáveis
  - Verifica os tipos

Usando a Árvore Sintática

- É possível emitir (produzir) o código intermediário durante a análise sintática, sem construir a árvore explicitamente

- É mais comum utilizar os dados da árvore sintática para isso
  - Os nós têm os atributos para as análises sintática e semântica
  - Dessa forma, os dados para emitir o CI estão disponíveis
  - Basta percorrer a árvore e gerar o código equivalente para cada construção

Emitter.java

- Classe do pacote *inter* que possui um atributo `code` que armazena todo o código intermediário
- Cada método dessa classe gera algum trecho de código em LLVM
- Atualmente, a classe possui apenas os métodos para gerar o bloco principal do programa

Node.java

- O CI é comum a todos os nós
- Portanto há um atributo estático que guarda o mesmo código para todos eles
- Também há um método para obter esse código

```
public abstract class Node {
    private LinkedList<Node> children =
            new LinkedList<Node>();
    protected static Emitter code =
            new Emitter();

    public static String code() {
        return code.toString();
    }
```

Stmt.java

- Todos os comandos precisam de um método para gerar o código intermediário
- Então vamos incluir um método abstrato em `Stmt` que afetará todos os comandos
- Sobrescreva esse método em todas as classes herdeiras
  - Deixe o método vazio

```
public abstract class Stmt
extends Node {
    public abstract void gen();
}
```

Program.java

- Por agora, vamos incluir apenas o código para o bloco principal do programa
- Apenas o início e o fim do programa

```
@Override
public void gen() {
        code.emitHead(id);
        block.gen();
        code.emitFoot();
}
```

Block.java

- O código de um bloco é composto pelos códigos dos comandos que ele contém

```
@Override
public void gen() {
    for( Node s: children )
        ((Stmt)s).gen();
}
```

Parser.java

- Após a análise, o código intermediário já estará pronto
- Apenas por conveniência, vamos acessar o código a partir do *parser*

```
public void parse() {
    Stmt p = program();
    root = p;
    p.gen();
}

public String code() {
    return Node.code();
}
```

DL.java

- Agora para ver o código, precisamos apenas imprimi-lo
- Apenas o bloco principal será gerado

- **Teste o código!**

```
//Imprimindo a árvore sintática e CI
System.out.println(p.parseTree());
System.out.println(p.code());
System.out.println("finalizado");
```

Declaração

Tradução de declaração

DECLARAÇÃO EM DL

```
inteiro a;
real b;
booleano c;
```

TRADUÇÃO PARA LLVM-IR

```
%a = alloca i32
store i32 0, i32* %a
%b = alloca double
store double 0.0, double* %b
%c = alloca i1
store i1 0, i1* %c
```

Para uma declaração precisamos de

- Um comando para **alocar memória** (`alloca`)
- Um comando para inicializar o **valor da variável** (`store`)
- Uma sequência de **temporários** que estarão vinculados aos **identificadores**

Emitter.java `alloca`

- Um método para emitir o comando `alloca`
- Ele usa o comando `codeType()` que traduz os tipos da DL para os tipos da LLVM-IR

```
%a = alloca i32
```

```
public void emitAlloca(Expr var) {
    emit( var + " = alloca "
            + codeType(var.type()));
}

public static String codeType(Tag type) {
    switch (type) {
    case BOOL: return "i1";
    case INT: return "i32";
    case REAL: return "double";
    default: return "";
    }
}
```

Emitter.java `store`

- Um método para emitir o comando `store`
- Também usa o método `codeType()` definido anteriormente
- Para usar esses métodos, precisamos definir o código para as expressões

`store i32 0, i32* %a`

```
public void emitStore(Expr dest,
        Expr value) {
    emit( "store "
            + codeType(dest.type())
            + " " + value + ", "
            + codeType(dest.type())
            + "* " + dest);
}
```

Emitter.java

- Declare esses atributos constantes na classe `Emitter`
- Essas constantes serão usadas para inicializar as variáveis após a declaração

```
public static final Literal LIT_ZERO_INT =
    new Literal( new Token(
        Tag.LIT_INT, "0"), Tag.INT);
public static final Literal LIT_ZERO_REAL =
    new Literal( new Token(
        Tag.LIT_REAL, "0.0"), Tag.REAL);
```

Decl.java

- Tendo como modelo os comandos acima, fica fácil entender como o código fica
- Já podemos declarar variáveis
- **Teste o código!**

```
%a = alloca i32
store i32 0, i32* %a
%b = alloca double
store double 0.0, double* %b
%c = alloca i1
store i1 0, i1* %c
```

```
@Override
public void gen() {
    Literal init = (id.type()==Tag.REAL
            ? Emitter.LIT_ZERO_REAL
            : Emitter.LIT_ZERO_INT);
    code.emitAlloca(id);
    code.emitStore(id, init);
}
```

Comando Escreva

Tradução de declaração

ESCREVA EM DL

```
inteiro a;
escreva(a);
```

TRADUÇÃO PARA LLVM-IR

```
%a = alloca i32
store i32 0, i32* %a
%1 = load i32, i32* %a
%2 = call i32 (i8*, ...) @printf(i8*
getelementptr inbounds([4 x i8], [4 x
i8]* @str_print_int, i32 0, i32 0),
i32 %1)
```

Emitter.java
load

- Antes de escrever o valor de uma variável, é preciso lê-lo da memória
- Na verdade isso acontece sempre que o valor de uma variável vai ser usado
- A leitura da memória é feito com o comando `load`

```
%1 = load i32, i32* %id
```

```
public void emitLoad(
        Expr dest, Expr value) {
    emit( dest + " = load "
        + codeType(dest.type()) + ", "
        + codeType(dest.type()) + "* "
        + value);
}
```

Expr.java

- Assim como os comandos, todas as expressões também geram código
- A função ao lado afeta todas as expressões
- Implemente este métodos em todas as classes herdeiras

```
public abstract Expr gen();
```

Temp.java

- Crie essa classe no pacote `inter.expr`
- Essa classe será responsável por criar a sequência de temporários
- Cada novo temporário terá seu número incrementado em relação ao anterior

```
public class Temp extends Expr {
    private static int count = 1;
    private int number;
    private static Token TOKEN_TEMP =
            new Token(Tag.TEMP, "");

    public Temp(Tag type) {
        super(TOKEN_TEMP, type);
        number = count++;
    }

    public int getNumber() {
        return number;
    }
```

Temp.java

- O método `toString()` gera a forma %n para o CI
- O temporário é um nó terminal, então seu código é ele mesmo

```
    @Override
    public Expr gen() {
        return this;
    }

    @Override
    public String toString() {
        return "%" + number;
    }
}
```

Id.java

- Antes de escrever o valor de uma variável, é preciso lê-lo da memória
- Então o código para uma variável é simplesmente lê-lo da memória
- O valor é jogado para um temporário (registrador virtual)
  - Esse temporário é retornado pela função

```
@Override
public Expr gen() {
    Temp d = new Temp(type);
    code.emitLoad(d, this);
    return d;
}
```

emitWrite

- Os comandos de entrada e saída chamam funções do C
- Então é necessário declarar o método antes de usar
- Essa declaração já está feita no cabeçalho do programa
- Descomente o comando `emitWrite()` da classe `Emitter` e o analise

Write.java

- O código da variável é emitido antes
- Seu valor é jogado para um temporário
- O comando escreva escreve esse temporário

```
@Override
public void gen() {
    Expr e = id.gen();
    code.emitWrite(e);
}
```

DL.java

- Vamos gerar um arquivo com o código LLVM-IR
- Acrescente o bloco `try` ao lado antes de sinalizar que o programa terminou
- Use o terminal, vá até a pasta do projeto e execute
  - `$lli prog.ll`

```
try {
    PrintWriter pw =
        new PrintWriter("prog.ll");
    pw.write(p.code());
    pw.close();
} catch (Exception e) {
    e.printStackTrace();
}

System.out.println("finalizado");
```

Atribuição

Tradução de uma Atribuição

ESCREVA EM DL

```
a = b * 2;
```

TRADUÇÃO PARA LLVM-IR

```
%1 = load i32, i32* %b
%2 = mul i32 %1, 2
store i32 %2, i32* %a
```

Primeiro precisamos...

- Antes de atribuirmos uma expressão, precisamos gerar o código da expressão

Emitter.java

- Precisamos selecionar a instrução adequada
- Se for uma multiplicação do tipo real, deve selecionar a instrução `fmul`, por exemplo

```
public static String codeOperation(
        Tag op, Tag type) {
    if ( type.isReal() ) {
        switch( op ) {
        case SUM: return "fadd";
        case SUB: return "fsub";
        case MUL: return "fmul";
        default:  return null;
        }
    } else {
        switch( op ) {
        case SUM: return "add";
        case SUB: return "sub";
        case MUL: return "mul";
        default: return null;
        }
    }
}
```

Emitter.java

- Para executar uma instrução precisamos de três endereços
- Os operadores podem ser variáveis ou literais

```
%4 = mul i32 %3, 2
```

```
public void emitOperation(Expr dest,
        Expr op1, Expr op2, Tag op) {
    emit( dest + " = "
            + codeOperation(op, op1.type())
            + " " + codeType(op1.type())
            + " " + op1 + ", " + op2 );
}
```

Literal.java

- Um literal não precisa ser resolvido
- Seu código é o próprio literal

```
@Override
public Expr gen() {
    return this;
}
```

Bin.java

- Precisamos resolver as duas subexpressões antes da expressão atual
- O endereço de destino é um novo temporário

```
@Override
public Expr gen() {
      Expr e1 = expr1.gen();
      Expr e2 = expr2.gen();
      Temp d = new Temp(type);
      code.emitOperation(d, e1,
                  e2, op.tag());
      return d;
}
```

Assign.java

- Na atribuição basta resolver a expressão, jogando o resultando em um temporário
- Depois atribui esse temporário à variável de destino
- **Teste o programa!**
  - **O que ocorre se atribuir um inteiro a uma real?!?!**

```
%3 = mul i32 %2, 2
store i32 %3, i32* %a
```

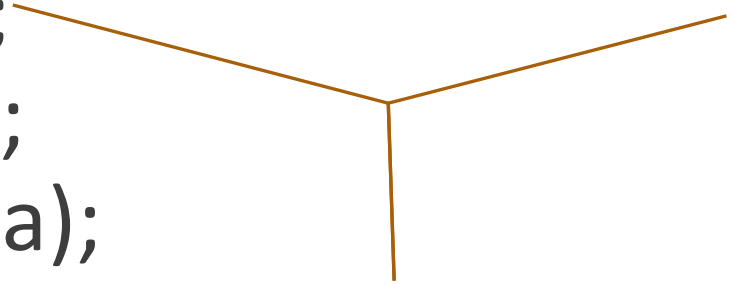
```
@Override
public void gen() {
    Expr e = expr.gen();
    code.emitStore(id, e);
}
```

Coerção

O que acontece com a tradução destes códigos?

```
programa teste inicio
  real a; real b;
  b = 2 - 4;
  a = b * 4;
  escreva(a);
fim.
```

```
programa teste inicio
  real a; real b;
  b = 2 - 4.0;
  a = b * 4;
  escreva(a);
fim.
```



Não pode atribuir uma expressão de um tipo diferente do tipo da variável

Conversão de Tipos

- As representações de números inteiros e de ponto flutuante são diferentes
- Diferentes instruções de máquina são usadas para diferentes tipos
- Se as regras da linguagem fonte ditam que tipos diferentes podem ser operados entre si, então deve haver uma conversão implícita (coerção)

Conversão na Atribuição

CÓDIGO EM DL

real a;
a = 2 – 4;

TRADUÇÃO PARA LLVM-IR

%a = **alloca double**
%1 = **sub i32** 2, 4
%2 = **sitofp i32** %1 **to double**
store double %2, **double*** %a

Emitter.java

- A instrução `sitofp` converte um valor inteiro para real

```
%4 = sitofp i32 %3 to double
```

```
public void emitConvert(Expr dest,
        Expr op) {
    emit( dest + " = "
            + "sitofp i32 "
            + op + " to double" );
}
```

Assign.java

- No construtor de `Assign` já são feitas as verificações de tipo
- O único caso de tipos diferente aceitável é tentar atribuir um inteiro a um real
- Caso isso aconteça, ocorre uma conversão

```
%a = alloca double
%1 = sub i32 2, 4
%2 = sitofp i32 %3 to double
store double %2, double* %a
```

```
@Override
public void gen() {
    Expr e = expr.gen();
    if ( id.type() == e.type() )
        code.emitStore(id, e);
    else {
        Temp t = new Temp(id.type());
        code.emitConvert(t, e);
        code.emitStore(id, t);
    }
}
```

Conversão nas Operações

CÓDIGO EM DL

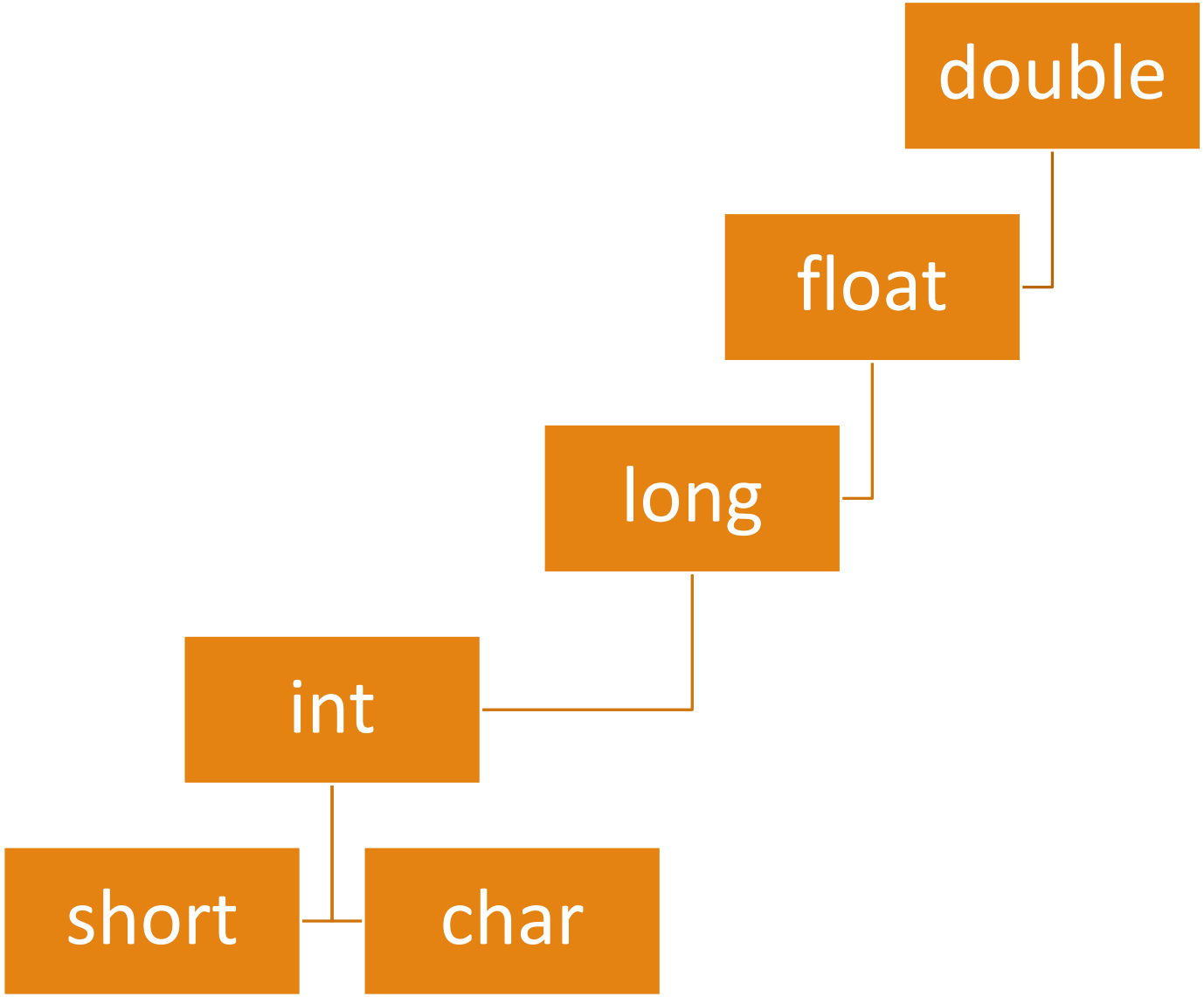
real a;
a = 2 – 4.0;

TRADUÇÃO PARA LLVM-IR

%a = **alloca double**
%1 = **sitofp i32** 2 **to double**
%2 = **fsub double** %2, 4.0
store double %2, **double*** %a

Conversão de Alargamento

- Preservam as informações
- Qualquer tipo mais abaixo na hierarquia pode ser convertido para um tipo mais acima
- Um `char` pode ser alargado para um `int` ou para um `float`, por exemplo
- Mas um `char` não pode ser alargado para um `short`
- E um `double` não pode ser alargado para um `int`



Expr.java

- Esse método faz a conversão de alargamento
- Se o tipo da expressão for “menor” que o tipo usado na comparação, então a expressão é convertida

```
public static Expr widen(Expr e,
        Tag type) {
    if ( e.type == type ||
            e.type().isReal() )
        return e;
    else if ( e.type().isInt() ) {
        Temp t = new Temp(Tag.REAL);
        code.emitConvert(t, e);
        return t;
    }
    error("Tipos incompatíveis");
    return null;
}
```

Bin.java

- Antes de emitir o código, há a possibilidade de ocorrer conversão de tipo de um dos operandos
- **Teste o código!**

```
@Override
public Expr gen() {
    Expr e1 = expr1.gen();
    Expr e2 = expr2.gen();
    Expr op1 = widen(e1, e2.type());
    Expr op2 = widen(e2, e1.type());
    Temp d = new Temp(type);
    code.emitOperation(d, op1,
            op2, op.tag());
    return d;
}
```

Bibliografia

- AHO, A. V.; SETHI, R.; ULLMAN, J. D. Compiladores: princípios, técnicas e ferramentas. Rio de Janeiro: LTC, 1995.
- CAMPBELL, B.; LYER, S.; AKBAL-DELIBAS, B. Introduction to Compiler Construction in a Java World. CRC Press, 2013.
- APPEL, A. W. Modern compiler implementation in C. Cambridge. Cambridge University Press, 1998.