

## Lecture 3.3

# 线性IR和解释执行

徐 辉

xuh@fudan.edu.cn



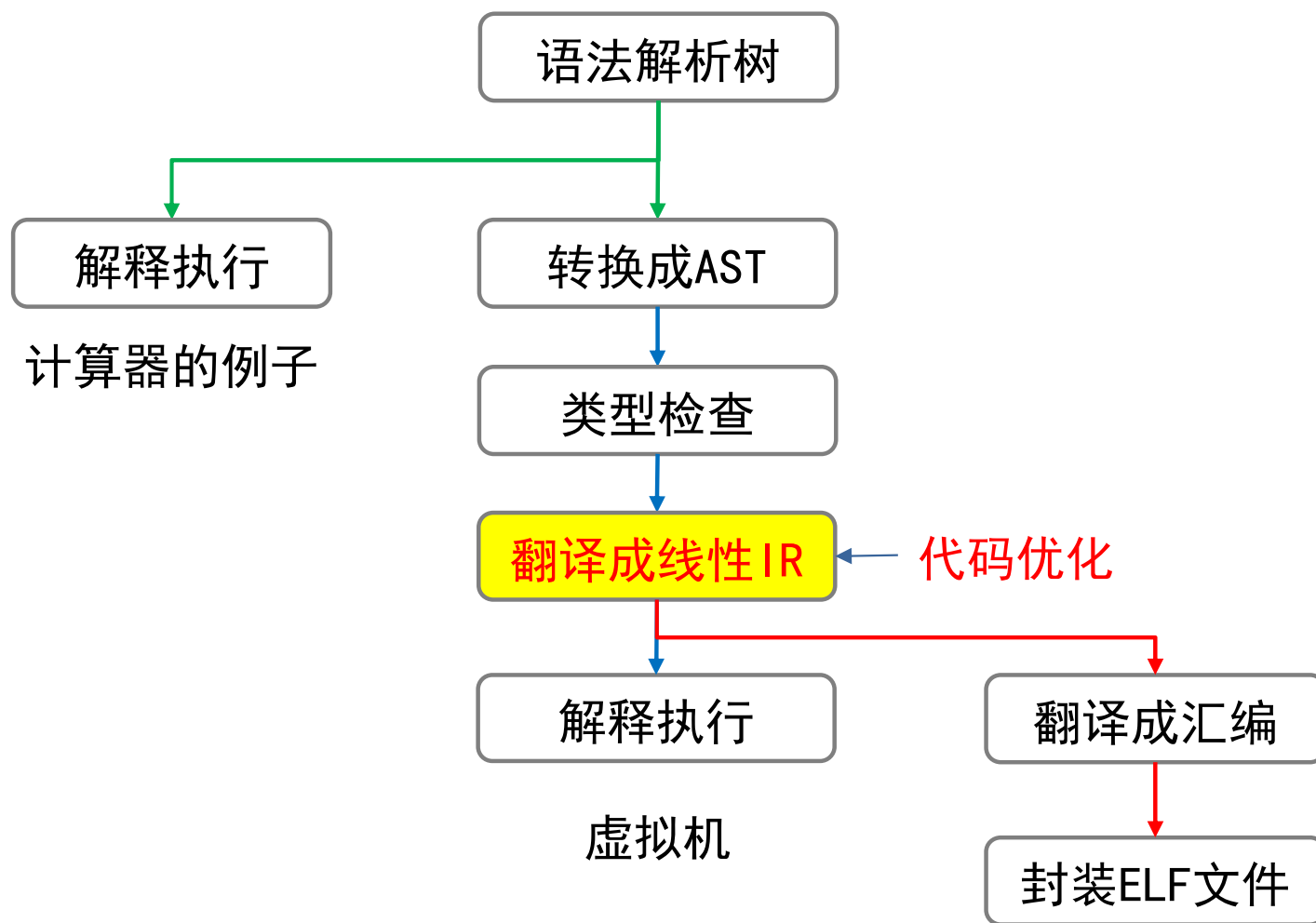
# 大纲

一、设计线性IR

二、翻译线性IR

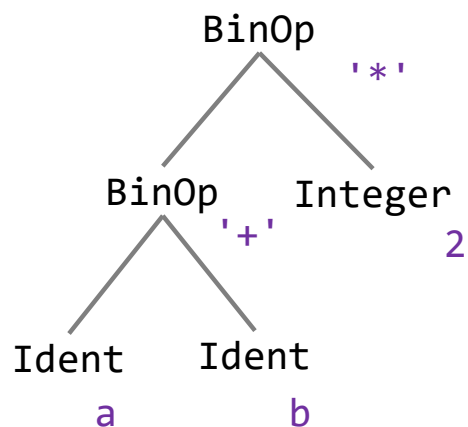
三、解释执行和虚拟机

# 展望

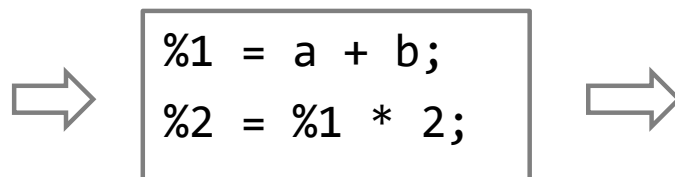


# 线性IR的基本形式

- 三地址代码，由指令和地址组成；
  - 地址：变量名、常量、编译器生成的临时变量或存储单元



抽象语法树



三地址线性IR

op	arg1	arg2	res
+	a	b	%1
*	%1	2	%2

四元式



```
mov  -4(%rbp), %eax
add  -8(%rbp), %eax
shl  $1, %eax
```

实际汇编代码



op	arg1	arg2
mov	a	r1
add	b	r1
mul	2	r1

两地址形式



op	arg1	arg2
+	a	b
*	(1)	2

三元式

# IR定义1：标识符和基本运算

- 标识符

- 全局变量：@name
- 局部变量：%name
- 临时变量：%1、%2
- 常量：1、2

- 二元整数运算：

- add/sub/mul/div/rem

- 二元浮点数运算：

- fadd/fsub/fmul/fdiv/frem

- 二元位运算：

- and/or/xor
- shl/ashr/lshr

```
%1 = a + b;  
%2 = %1 * 2;
```



```
%1 = add i32, %a, %b;  
%1 = mul i32, %1, 2;
```

## 练习：

- 假设%0=-2，%1=1，计算下列运算结果

```
%3 = and i32 %0, %1  
%4 = or i32 %0, %1  
%5 = xor i32 %0, %1  
%6 = shl i32 %0, 1  
%7 = ashr i32 %0, 1  
%8 = lshr i32 %0, 1
```

# 浮点数运算需要单独的指令

- IEEE-754标准
- 计算方式:  $mantissa \times (2^{exp} - 127)$ 
  - 如200可表示成01000011010010000000000000000000
  - $1.5625 \times 2^7 = 200$

01000011010010000000000000000000



exponent (8 bits)

mantissa (23 bits)

$$2^7 + 2^2 + 2^1 - 127 \\ = 7$$

$$1 + 2^{-1} + 2^{-4} \\ = 1.5625$$

# 将实数转换为浮点数

- 将实数11.25转换为二进制表示

$$11/2 = 5 + 1$$

$$5/2 = 2 + 1$$

$$2/2 = 1 + 0$$

$$1/2 = 0 + 1$$

$$0.25 * 2 = 0.5 + 0$$

$$0.50 * 2 = 0.0 + 1$$

$$\begin{array}{l} 1011.01 \\ 1+.01101 \end{array} \Rightarrow \text{exp} = 3$$

010000010011010000000000000000000000000000000000



# 练习

- 下列哪个小数可以使用浮点数精确表示？
  - 0.1, 0.2, 0.3, 0.4, 0.5

## IR定义2： 类型转换

- 数据截断：trunc/fptrunc
- 数据扩充：zext/setx/fpext
- 浮点数整数互换：fptoui/fptosi/uitofp/sitofp
- 指针整数互换：ptrtoint/inttoptr

```
%1 = trunc i32 %0, i16
%2 = sitofp i32 %0, float
%3 = uitofp i32 %0, float
%4 = sext i32 %0, i64
%5 = zext i16 %0, i64
%6 = fptoui float %2, i32
%7 = fpext float %2, double
%8 = fptrunc double %7, float
%9 = inttoptr i64 %4, i8*
%10 = ptrtoint i8* %9, i32
```

# IR定义3：数据存取

- 栈空间分配：stackalloc
- 堆空间分配：heapalloc
- 数据读取：load
- 数据存入：store

```
a:int = 1;  
a = a + 1;
```



```
%a = stackalloc i32;  
store i32 1, %a;  
%1 = load i32, %a;  
%2 = add i32 %1, 1;  
store i32 %2, %a;
```

## IR定义4：指针和地址操作

- 偏移地址： `getptr`

```
let a[2]:int;  
a[1] = 99;
```



```
%a = stackalloc [2 x i32];  
%1 = getptr i32*, %a, 0, 1;  
store i32 99, %1;
```

```
struct st{  
    i:int;  
    f:float;  
};  
struct st s;  
s.i = 1;
```



```
%struct.st = type { i32, float };  
%s = stackalloc %struct.st;  
%1 = getptr %struct.st.i*, %s, 0, 0;  
store i32 1, %1;
```

# IR定义5：控制流语句

- 比较运算：
  - icmp/fcmp
    - eq/ne
    - gt/ge/lt/le
    - ugt/uge/ult/ule
- 跳转指令：
  - jmp：直接跳转
  - cjmp：条件跳转
  - match

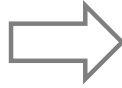
```
%0 = icmp eq i32 4, 5;  
%1 = icmp ne float 0.1, 0.2;  
%2 = icmp ult i16 4, 5;  
%3 = icmp sgt i16 4, 5;  
%4 = icmp ule i16 -4, 5;  
%5 = icmp sge i16 4, 5;
```

```
cjmp %0, %BB1, %BB2
```

```
match i32 %0, %BBdefault [  
    i32 0, %BB1  
    i32 1, %BB2  
    i32 2, %BB3 ]
```

# if-else语句的IR

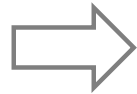
```
if(a)
    b++;
else
    b--;
```



```
%1 = load i32 %a;
%2 = icmp ne i32 %1, 0;
cjmp %2, %BB1, %BB2;
%BB1:
%3 = load i32 %b;
%4 = add i32 %3, 1;
store i32 %4, %b;
br %BB3;
%BB2:
%5 = load i32 %b;
%6 = sub i32 %5, 1;
store i32 %6, %b;
br %BB3;
%BB3:
```

# while语句的IR

```
while(a)
    a--;
```



%BB1:

```
%1 = load i32 %a;
%2 = icmp ne i32 %1, 0;
br i1 %2, %BB2, %BB3;
```

%BB2:

```
%3 = sub i32 %1, 1;
store i32 %3, %a;
br %BB1;
```

%BB3:

...

# match语句的IR

```
match(x){  
    0: => { x = 0; }  
    1: => { x = 1; }  
    _: => { x = -1; }  
}
```



```
%0 = load i32 %x;  
match i32 %0, %BB3 [  
    i32 0, %BB1  
    i32 1, %BB2  
]  
%BB1:  
    store i32 0, %x;  
    jmp %BB4  
%BB2:  
    store i32 1, %x;  
    jmp %BB4  
%BB3:  
    store i32 -1, %x;  
    jmp %BB4  
%BB4:  
    ...
```



## IR定义6： 函数

- 函数声明： fn
- 函数调用： call
- 返回指令： ret

```
@define void @foo(i32 %x){  
    ...  
    ret;  
}  
@define i32 @bar(i32 %y){  
    ...  
    ret i32 %1;  
}  
call void @foo(1)  
%1 = load %a  
%2 = call i32 @test(i32 %1)
```

# 大纲

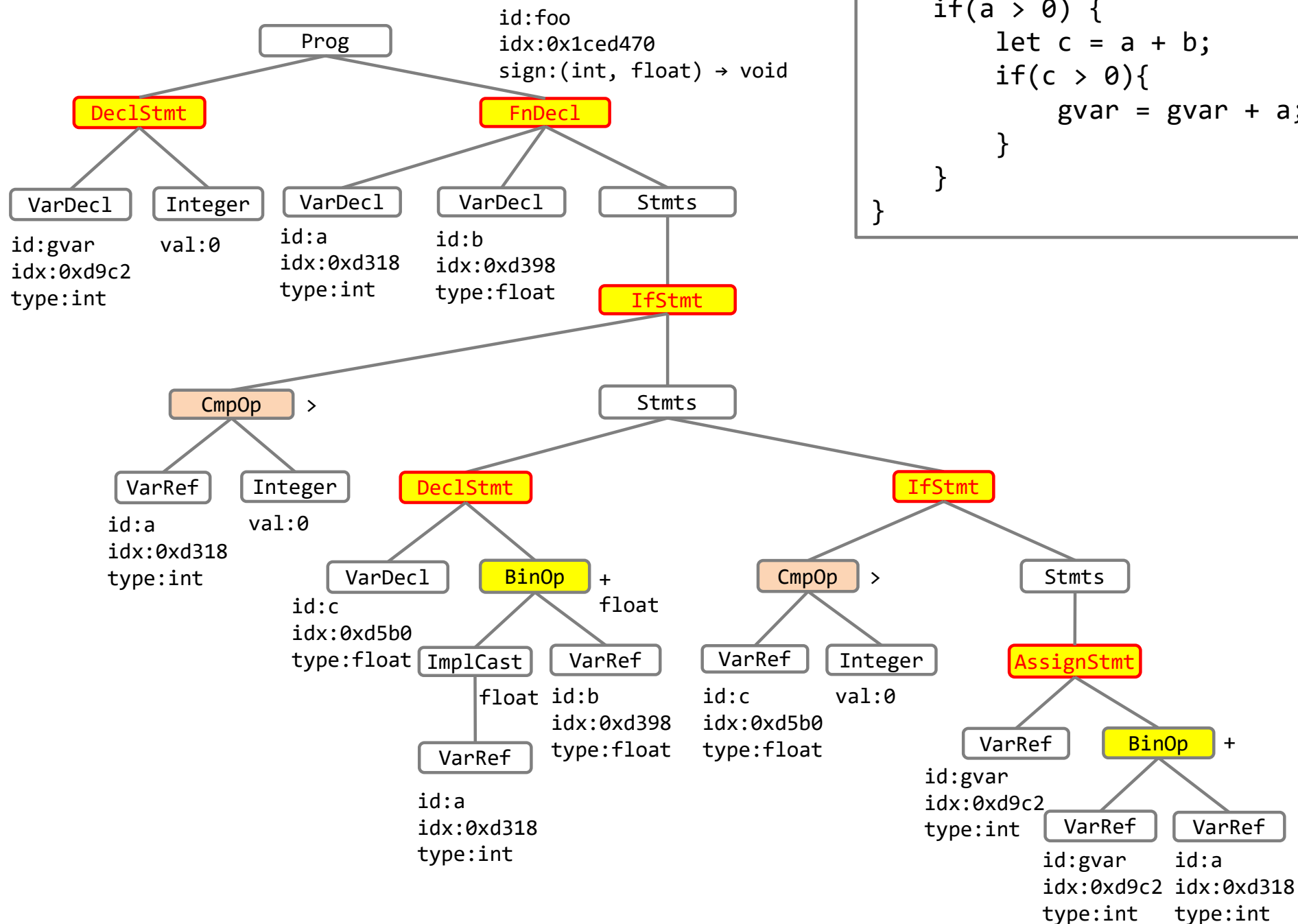
一、设计线性IR

二、翻译线性IR

三、解释执行和虚拟机

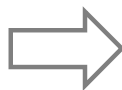
# 将该代码翻译为线性IR

```
let gvar:int = 0;
fn foo(int a, float b){
    if(a > 0) {
        let c = a + b;
        if(c > 0){
            gvar = gvar + a;
        }
    }
}
```



# 提示:

```
let gvar:int = 0;
fn foo(a:int, b:float){
  if(a > 0) {
    let c = a + b;
    if(c > 0){
      gvar = gvar + a;
    }
  }
}
```



```
@gvar = i32 0;
define fn foo(i32 %a, float %b){
  %a = stackalloc i32;
  store %-1, %a;
  %b = stackalloc float;
  store float %-2, %b;
  %1 = load i32 %a;
  %2 = icmp ge %1, 0;
  cjmp %2, %BB1, %BB4;
%BB1:
  ...
%BB2:
  ...
%BB3:
  ...
%BB4:
  ...
}
```

# 基本思路

- 为每一种AST节点定义转换方式
- 前序遍历AST树

```
GenIR(node) {  
    match (node.type) {  
        FNDECL => { ... }  
        DECLSTMT => { ... }  
        ASSIGN => { ... }  
        BINOP => { ... }  
        CMPOP => { ... }  
        IFSTMT => { ... }  
        ...  
    }  
}
```

# IR数据结构设计参考

```
struct ProgIR {  
    gvlist:list<GlobalVar>;  
    fnlist:list<FnIR>;  
}
```

← 程序IR组成：全局变量IR+函数

```
struct FnIR {  
    sign:FnSignIR;  
    bblist:list<BB>;  
}
```

← 函数组成：函数签名+代码块

```
struct BB {  
    id:int;  
    list<InstType> ilist;  
}
```

← 代码块组成：id+指令列表

# BINOP

```
BINOP => {  
    if (!IsLeaf(node.child[0])) {  
        GenIR(curbb,node.child[0]);  
    }  
    if (!IsLeaf(node.child[1])) {  
        GenIR(curbb,node.child[1]);  
    }  
    ir = CreateIR(node.op,  
                  node.child[0].id,  
                  node.child[1].id);  
    curbb.irlist.add(ir);  
}
```

如子节点不是叶子节点，  
=>递归生成子节点的IR，  
=>子节点取得临时变量标识

使用子节点变量标识  
如为临时变量，则直接使用，  
如为局部变量，则先load

将IR加入当前代码块

# ASSIGN

```
ASSIGN => {  
    if (!IsLeaf(node.child[1])) {  
        GenIR(curbb,node.child[1]);  
    }  
    ir = CreateIR(ASSIGN,  
                  node.child[0].id  
                  node.child[1].id);  
    curbb.irlist.add(ir);  
}
```

← 先计算，后store



# DECLSTMT

```
DECLSTMT => {  
    if (node.child.length < 2)) {  
        ir = CreateIR(DECL,  
                      node.child[0]);  
    } else {  
        if (!IsLeaf(node.child[1])) {  
            GenIR(curbb,node.child[1]);  
        }  
        ir = CreateIR(DECL,  
                      node.child[0]  
                      node.child[1].id);  
    }  
    curbb.irlist.add(ir);  
}
```

← 无初始化, 仅stackalloc

← 含初始化, 先stackalloc  
后赋值

# IFSTMT

```
IFSTMT => {  
    let childnum = node.child.length;  
    let bbs:[BB;] = CreateBB(curfn,childnum+1) ← 分配执行完后的BB  
    if (!IsLeaf(node.child[0])) { ← 条件语句模块  
        GenIR(curbb,node.child[0]);  
    }  
    ir = CreateIR(CJMP, ← 参考if跳转关系  
                  node.child[0].id,  
                  bbs);  
    curbb.irlist.add(ir);  
    GenIR(bbs);  
}
```

# FNDECL

```
FNDECL => {  
    let curfn = CreateIR(FNDECL,node.sign); ← 创建函数签名  
    prog.add(curfn);  
    let childnum = node.child.length;  
    let curbb = CreateBB(curfn);  
    for i in 0..childnum-1 {  
        if (node.child[i].type == ARG) { ← 参数节点, stackalloc  
            ir = CreateIR(ARG,  
                           node.child[i]);  
            curbb.irlist.add(ir);  
        }  
        if (node.child[i].type == STMTS) { ← 函数体  
            GenIR(curbb,node.child[i]);  
        }  
    }  
}
```

# 大纲

一、设计线性IR

二、翻译线性IR

三、解释执行和虚拟机

# 解释执行：interpreter

- 解释执行另外一个程序：源代码/AST/IR
- 无需考虑后端，简化了语言的实现
- 计算器的例子：直接翻译为目标机器指令

# 如何设计IR解释执行器？

- 通过循环不断获取下一条指令并执行
  - 来源可以是Bytecode或编译过程的中间结果

```
enum {  
    loadInst,  
    addInst,  
    subInst,  
    mulInst,  
    divInst,  
    icmpInst,  
    cjmpInst,  
    callInst,  
    ...  
} instType;
```

```
static prog:[instType;n] = { ... };  
let pc:*instType = prog;  
while(1) {  
    match (*pc++) {  
        addInst => { ... }  
        subInst => { ... }  
        ...  
    }  
}
```

# 指令语义实现关键

- 每个指令都有独立的代码块，如何在一个代码块中使用另外一个代码块的结果？
- 对于符号表中的变量：
  - 保存为全局变量？
    - 不可行：递归+浪费空间
  - 保存在调用栈上，使用相对固定的位置
    - 使用数组模拟
- 对于临时变量：
  - 当成符号表中的变量处理？
    - 不可行：语义混淆+浪费空间
  - 保存在寄存器数组中或栈上
    - 使用数组模拟

```
%1 = load i32 %a;  
%2 = load i32 %b;  
%3 = add i32 %1, %2;  
store i32 %3, %c;
```

# addInst: 寄存器版本

- 如果操作数是变量，要求先load
- 还需要约定IR的哪些内容？

```
%1 = load i32 %a;  
%2 = load i32 %b;  
%3 = add i32 %1, %2;  
store i32 %3, %c;
```

```
id = 0;  
loadInst => {  
    r[id++] = *arg1;  
}  
addInst => {  
    r[id++] = r[id-1]+r[id-2];  
}  
storeInst => {  
    *arg1 = r[id];  
}
```



# addInst: 栈版本

```
%1 = load i32 %a;  
%2 = load i32 %b;  
%3 = add i32 %1, %2;  
store i32 %3, %c;
```

```
stack s;  
loadInst => {  
    s.push(*arg1);  
}  
addInst => {  
    //有两个变量参数的情况  
    v1 = s.pop();  
    v2 = s.pop();  
    v2 = v1 + v2;  
    s.push(v2);  
}  
storeInst => {  
    v1 = s.pop ();  
    *arg1 = v1;  
}
```

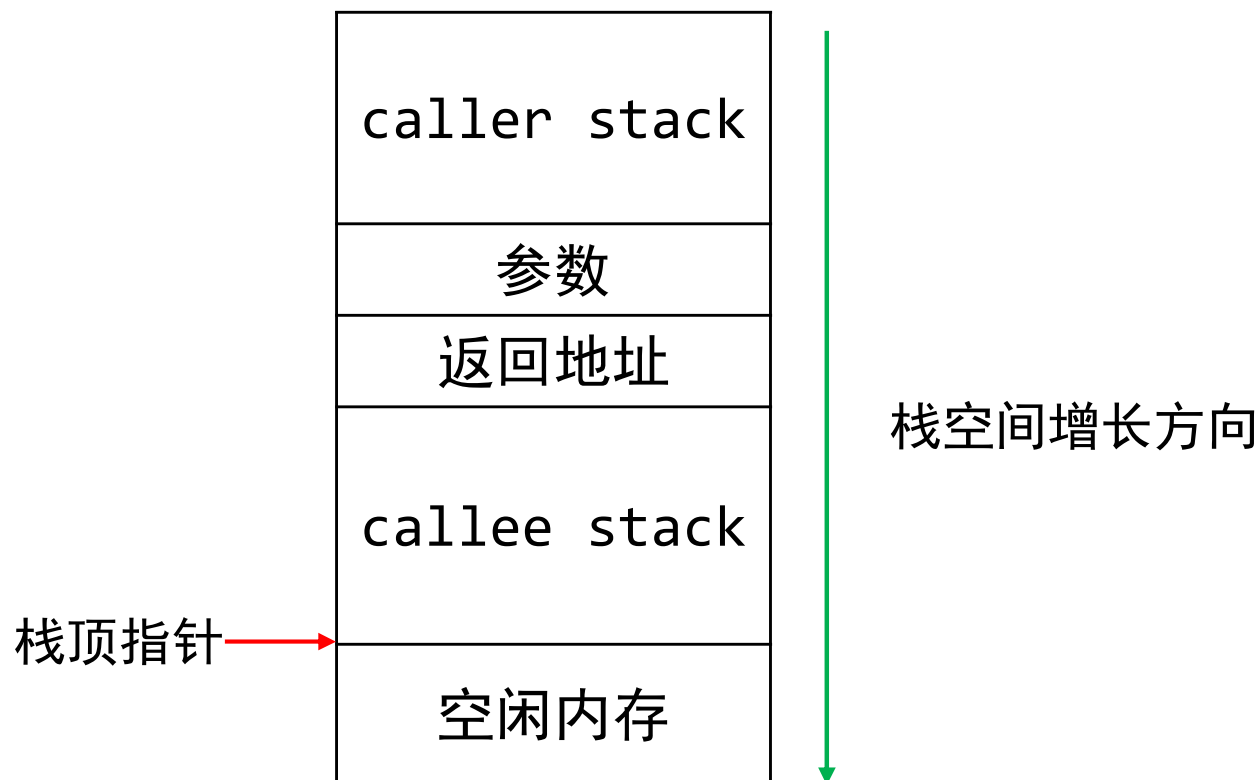
## 练习：

- 写出下列代码的线性IR
- 分析寄存器版本和栈版本的解释执行过程

```
r = a + b + c + d;
```

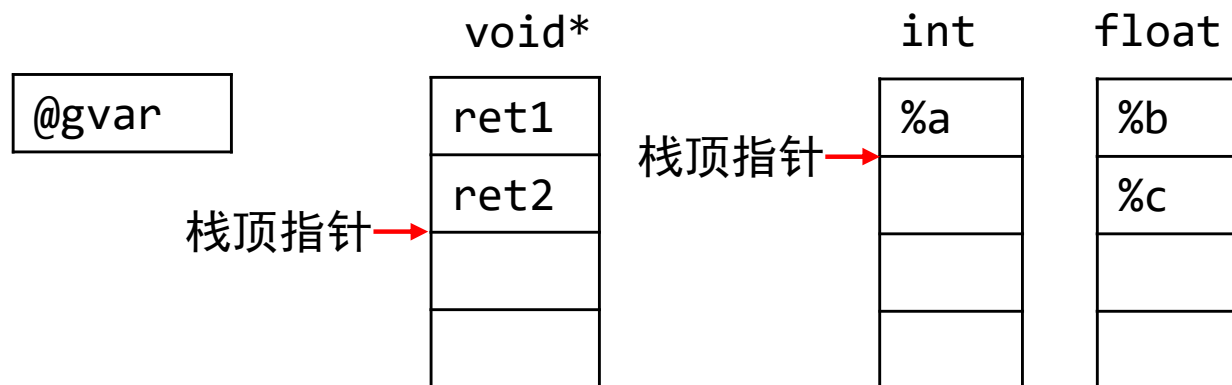
# 函数调用：Activation Record

- 为每个函数调用分配一块儿内存空间
- 函数自身所需栈空间可在编译时确定（栈vs堆）
- 函数返回后收回



# 如何设计？

- 全局变量在整个程序运行期间有效，单独存放
- 函数调用栈使用数组模拟
  - 存储返回地址和数据
  - 可放在一个栈中，需要类型转换
  - 或将不同数据类型分开存放
- 动态维护栈顶指针



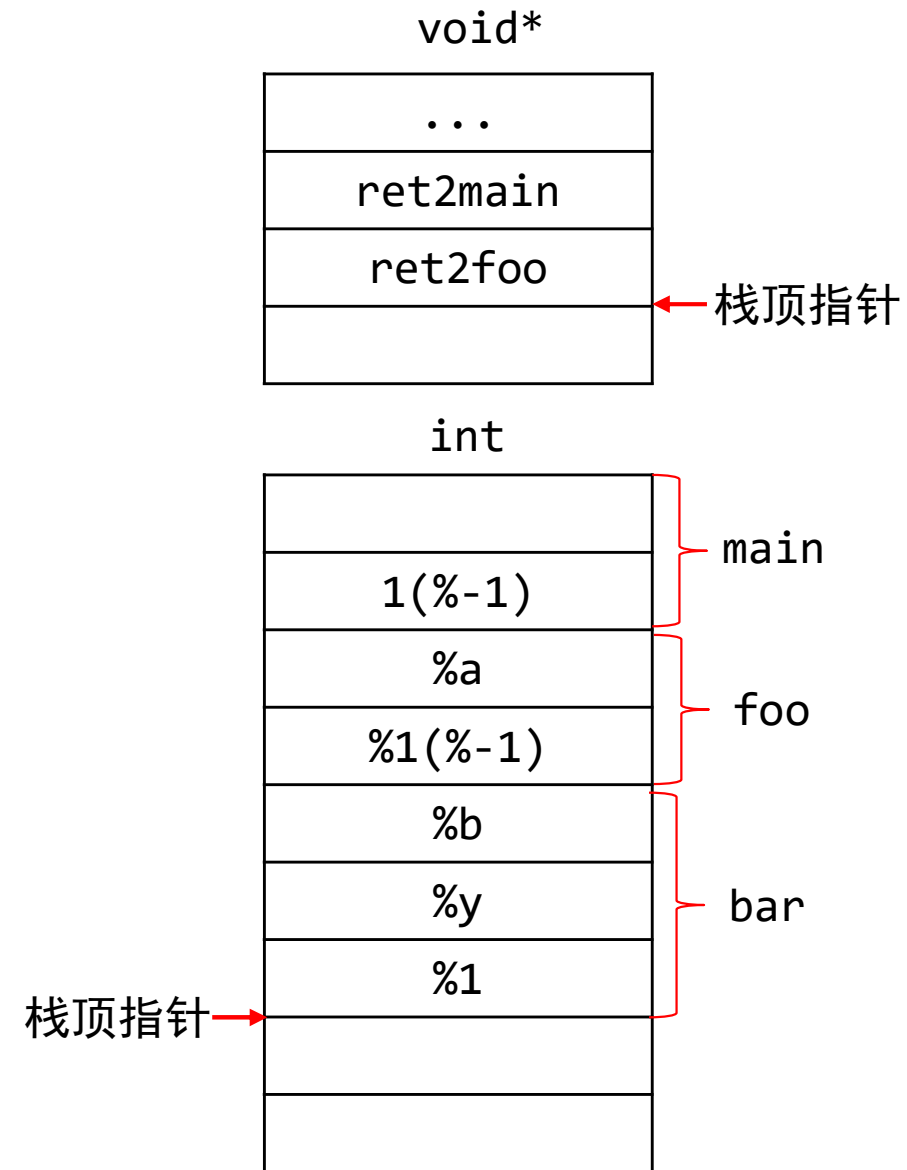
# 应用举例

```
define fn i32 bar(i32 %b){
    %b = stackalloc i32;
    store i32 %-1, %b;
    %y = stackalloc i32;
    ...
    ret %1;
}

define fn void foo(i32 %a){
    %a = stackalloc i32;
    store i32 %-1, %a;
    %1 = load %a
    %2 = call bar(%1);
    ret;
}

define fn main(i32 argc, i8** argv) {
    ...
    call foo(1);
}
```

代码执行 →



# 有几类信息需要预处理

- 局部变量在栈上的位置
- 跳转地址

符号表

name	offset
%a	
%b	

name	offset
%BB1	
%BB2	

```
id = 0;
loadInst => {
    addr = getAddr(arg1);
    s.push(*addr);
}
storeInst => {
    addr = getAddr(arg1);
    *arg1 = s.pop();
}
```

# callInst

```
stack s;  
retval;  
callInst => {  
    s.push(getRetaddr());  
    s.push(base);  
    base = s.top();  
}  
stackAllocInst => {  
    if(arg1 == i32) {  
        s.push(0);  
    }  
    ...  
}
```

```
retInst => {  
    v1 = s.pop();  
    retval = v1;  
    ClearStack();//清除临时变量  
    base = s.pop();  
    rip = s.pop();  
}
```

# 逃逸分析

- 局部变量在函数返回后是否继续被使用?
  - Bug!!!
  - 指向该变量的指针逃逸到其它函数或线程
  - 应在堆上分配内存
- 对于Java这种默认在堆上分配内存的语言，可通过逃逸分析将内存分配在栈上
  - 对象内存与函数栈帧生命周期绑定
  - 降低垃圾回收负担，节约内存

```
fn foo -> &i32(){  
    let i:i32 = 999999;  
    return &i;  
}
```

```
public static void example() {  
    Foo foo = new Foo();  
}  
  
class Foo {  
    private int i;  
    public void set(int i) {  
        this.i = i;  
    }  
}
```



# 虚拟机

- 为解释执行提供了程序运行抽象
  - 内存管理（栈、堆、垃圾回收）
  - 寄存器
  - 多线程
  - ...
- 优点：
  - 高效：优化策略
  - 方便： "Write once, run anywhere"
- 比较有名的虚拟机：
  - Java: HotSpot、Dalvik (Android)
  - Javascript: Chrome v8、Chakra、SpiderMonkey、JavaScriptCore

# Java虚拟机



# 优化思路

- 使用寄存器储存临时变量
  - 寄存器分配问题
- Threaded code
- JIT
  - 优化问题
- ...

# 使用Threaded Code

- Match-Case的问题：需要两次跳转
  - 一次间接跳转：跳转到分支代码
  - 一次直接跳转：返回循环入口
- 可否跳转一次？
  - 为每个指令设计一个处理函数或代码块
  - 开启尾递归

```
static prog:[instType;n] = { ... };  
let pc:*instType = prog;  
  
static fn add() {  
    ...  
    (*++pc.fnaddr)();  
}  
...  
(*pc.fnaddr)();
```

# 总结

- 线性IR的设计：指令集和翻译模式
  - 标识符、基本运算、数据存取、控制流、函数...
- 如何将AST翻译成线性IR
  - 前序遍历AST树
- 解释执行IR
  - 函数调用栈、Activation Record