

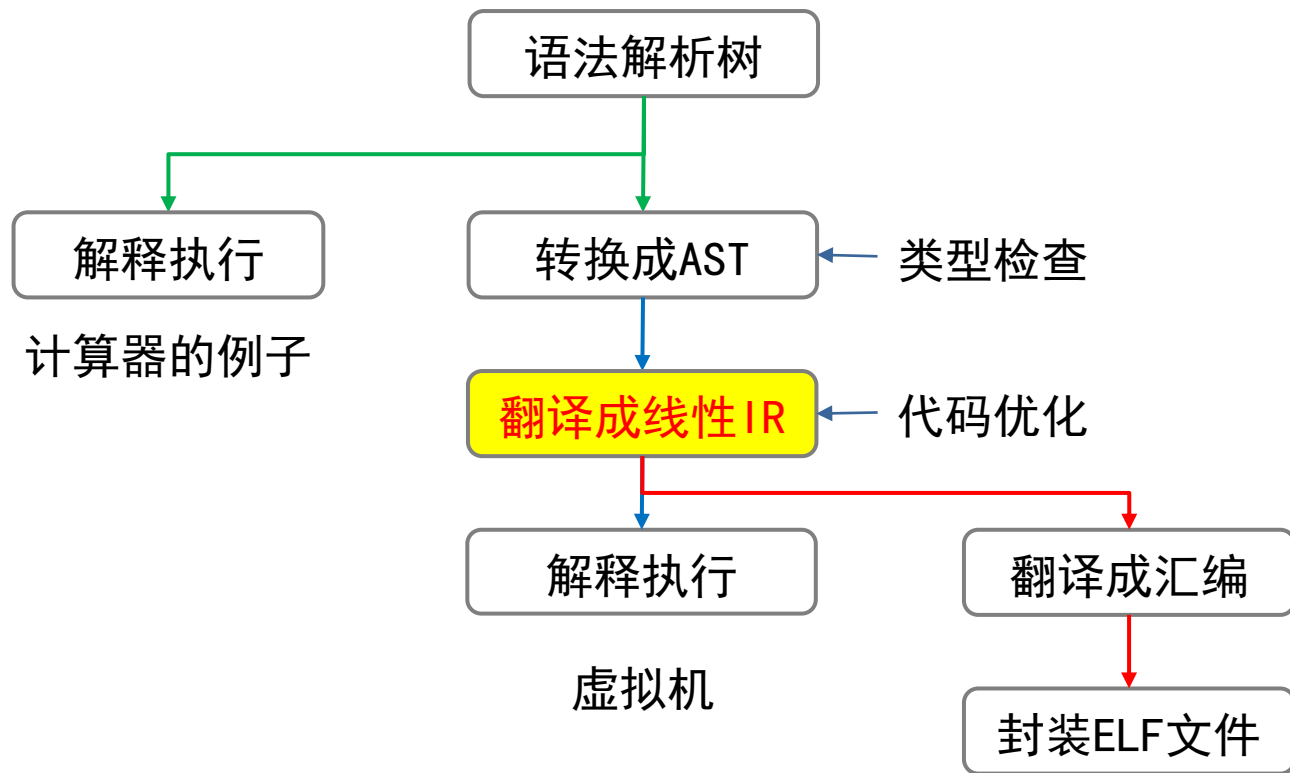
Lecture 8

线性IR

徐 辉

xuh@fudan.edu.cn





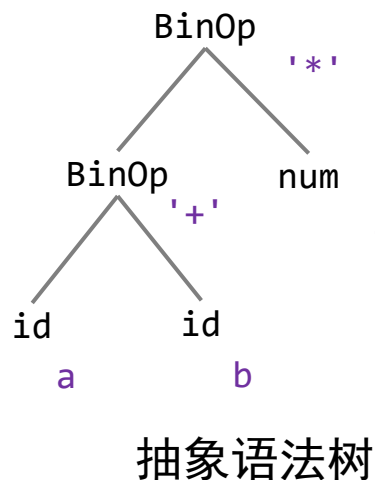
大纲

- 一、线性IR定义（LLVM IR）
- 二、AST转线性IR
- 三、解释执行

一、线性IR定义

线性IR的基本形式

- 指令 + 地址（一般为两地址或三地址）
 - 变量名、常量、编译器生成的临时变量或存储单元
- 比较有名的IR：LLVM IR、GCC GIMPLE、Java Bytecode



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

三地址线性IR



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

四元式

TeaPL的IR

- 选取LLVM IR的子集
- 可以使用现成工具执行IR: `lli`

```
@g = global i32 10

define i32 @fib(i32 %0) {
    %2 = alloca i32
    %3 = load i32, i32* @g
    ret i32 %3
}

define i32 @main() {
    %1 = call i32 @fib(i32 1)
    ret i32 %1
}
```

```
#: lli foo.ll
#: echo $?
```

IR定义：标识符、基础类型、和数据存取

- 全局变量/函数名称：@name
- 局部变量/临时变量：%x、%0（不可重复定义，纯数字编号需连续）
- 类型：void、i32、i32*、i8、i8*、i1
- 栈空间分配：alloca
- 数据存取：load/store

```
@g = global i32 10
```

声明全局变量g，类型*i32，初始值10

```
define i32 @fib(i32 %0) {
```

声明函数fib，参数名为%0，类型i32

```
    %x = alloca i32
```

声明局部变量%x，类型为i32*

```
    store i32 %0, %x
```

```
    %g0 = load i32, i32* @g
```

声明临时变量%g0，类型为i32

```
    ret i32 %g0
```

```
}
```

```
define i32 @main() {
```

```
    %r0 = call i32 @fib(i32 1)
```

```
    ret i32 %r0;
```

```
}
```

IR定义：函数

- 声明：define
- 调用：call
- 返回：ret

```
@g = global i32 10
```

```
define i32 @fib(i32 %0) {  
    %x = alloca i32  
    store i32 %0, %x  
    %g0 = load i32, i32* @g  
    ret i32 %g0  
}
```

→ 声明函数fib，参数类型i32

```
define i32 @main() {  
    %r0 = call i32 @fib(i32 1)  
    ret i32 %r0;  
}
```

→ 声明函数main

→ 调用函数fib

IR定义：复合类型数据定义和存取

```
define i32 @main() {  
  %1 = alloca [2 x i32]      —————> 创建一维数组，并返回数组指针  
  %2 = getelementptr [2 x i32], [2 x i32]* %1, i32 0, i32 0  
  store i32 99, i32* %2      —————> 获取元素地址  
  %3 = load i32, i32* %2  
  ret i32 %3  
}
```

索引为0的元素

```
%mystruct = type { i32, i32 } —————> 定义mystruct数据类型  
define i32 @main() {  
  %1 = alloca %mystruct      —————> 创建mystruct对象，并返回指针  
  %2 = getelementptr %mystruct, %mystruct* %1, i32 0, i32 0  
  store i32 1, i32* %2  
  ret i32 0  
}
```

IR定义：算数运算

```
%2 = alloca i32
%3 = add i32 %0, 1
%4 = sub i32 %3, 2
%5 = mul i32 %3, 3
%6 = sdiv i32 %4, 4
store i32 %6, i32* %2
%7 = load i32, i32* %2
ret i32 %6
```

浮点数运算用fadd/fsub/fmul/fdiv

IR定义：比较运算和类型转换

```
%2 = alloca i32
%3 = alloca i8
store i32 %0, i32* %2
%4 = load i32, i32* %2
%5 = icmp sgt i32 %4, 0
%6 = icmp sge i32 %4, 0
%7 = icmp slt i32 %4, 0
%8 = icmp sle i32 %4, 0
%9 = icmp eq i32 %4, 0
%10 = icmp ne i32 %4, 0
%11 = zext i1 %10 to i8
store i8 %11, i8* %3
%12 = load i8, i8* %3
%13 = trunc i8 %12 to i1
```

s: signed
g: greater
l: less
e: equal
n: not

类型转换：zero extend

类型转换：truncate

IR定义：跳转语句和Phi指令

- 基于br实现if-else和while等控制流功能

```
%2 = alloca i32
store i32 0, i32* %2
%3 = load i32, i32* %2
%4 = icmp sgt i32 %3, 0
br i1 %4, label %bb1, label %bb2
bb1:
store i32 1, i32* %2
br label %bb3
bb2:
store i32 0, i32* %2
br label %bb3
bb3:
%r0 = phi i32 [ 0, %bb1 ], [ %3, %bb2 ]
ret i32 %r0
}
```

条件跳转

直接跳转

如前序代码块为%bb1, 则%8=0,
如前序代码块为%bb2, 则%8=%3

IR定义：逻辑运算

- 无需定义专门的逻辑“与”和“非”指令

```
%7 = xor i1 %6, true
```

Not运算: !%6

二、翻译线性IR

TeaPL的代码如何对应到LLVM IR?

- 逻辑运算: `&&`、`||`
- 控制流语句: `if-else`、`while`
- 算术运算: `x = a + b + c`

逻辑运算

```
%5 = xor i1 %4, true  
br i1 %5, label %6, label %9
```

→ %5 && %7

6:

```
%7 = load i8, i8* %2,  
%8 = trunc i8 %7 to i1  
br label %9
```

9:

```
%10 = phi i1 [ false, %0 ], [ %8, %6 ]
```

```
br i1 %5, label %9, label %6
```

→ %5 || %7

6:

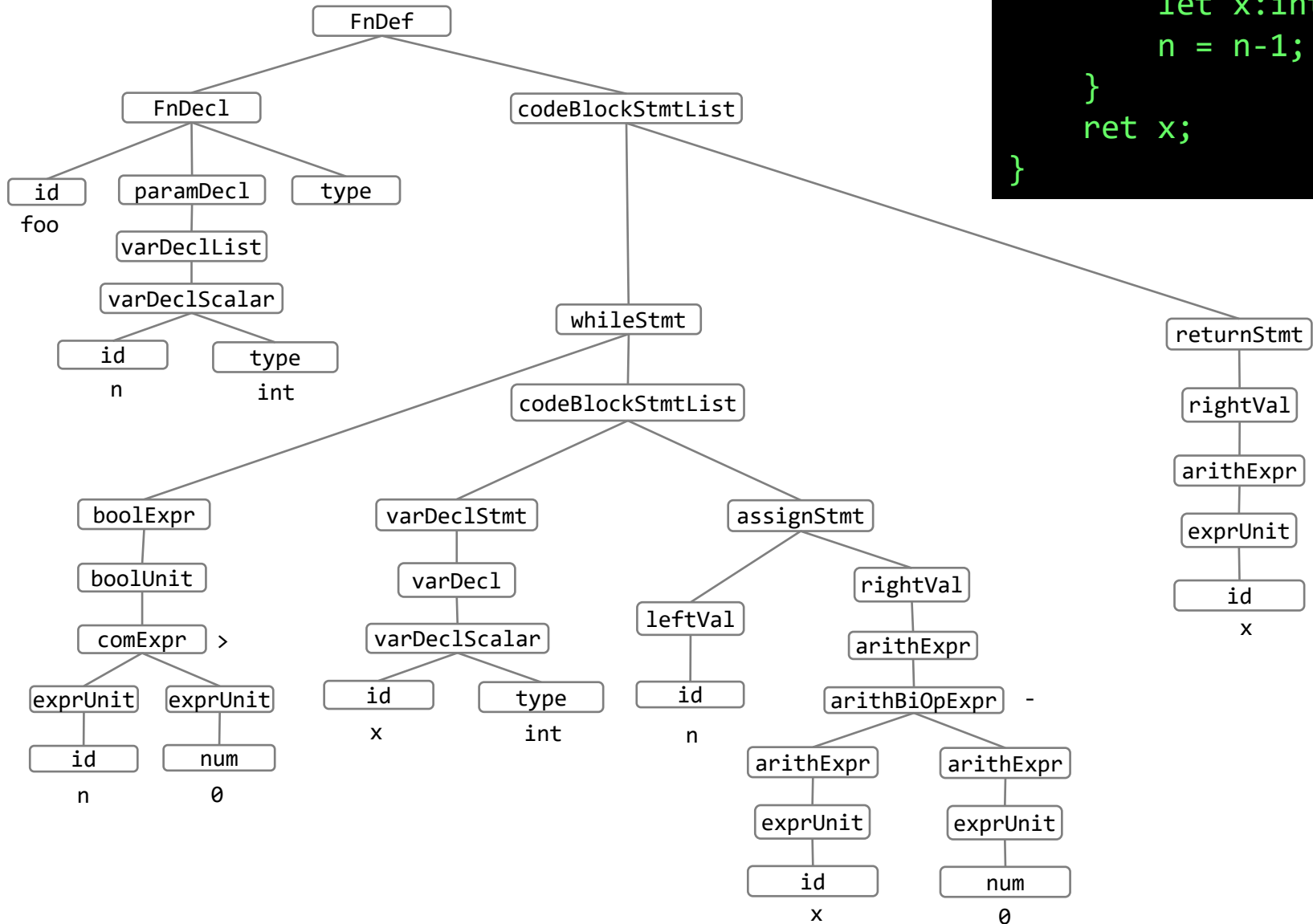
```
%7 = load i8, i8* %2, align 1  
%8 = trunc i8 %7 to i1  
br label %9
```

9:

```
%10 = phi i1 [ true, %0 ], [ %8, %6 ]
```


思考：如何将AST翻译为线性IR

```
fn foo(n: int) -> int {  
  while (n>0) {  
    let x:int;  
    n = n-1;  
  }  
  ret x;  
}
```



自顶向下翻译

```
GenIR(node) {  
  match (node.type) {  
    fnDef => { ... }  
    whileStmt => { ... }  
    varDeclStmt => { ... }  
    assignStmt => { ... }  
    retStmt => { ... }  
    ...  
  }  
}
```

根据AST节点类型递归翻译线性IR


```
struct ProgIR { // 程序IR  
  gvlist:list<GlobalVar>;  
  fnlist:list<FnIR>;  
}  
struct FnIR { // 函数组成  
  sign:FnSignIR;  
  bblist:list<BB>;  
}  
struct BB { // 代码块组成  
  id:int;  
  list<InstType> ilist;  
}
```

目标数据结构


消除块与块之间的数据依赖关系

- 每个代码块使用变量值前先load
- 每个代码块更新变量值后立即store

```
bb0:
    %x = alloca i32
    store i32 0, i32* %x
    %x0 = load i32, i32* %x
    %x1 = add i32 %x0, 1
    store i32 x1, i32* %x
    br label %bb1
bb1:
    %x2 = load i32, i32* %x
    %x3 = add i32 %x0, 1
    store i32 x3, i32* %x
    br label %bb2
bb2:
    ...
```



```
bb0:
    %x = alloca i32
    store i32 0, i32* %x
    %x0 = load i32, i32* %x
    %x1 = add i32 %x0, 1
    br label %bb1
bb1:
    %x3 = add i32 %x1, 1
    store i32 x3, i32* %x
    br label %bb2
bb2:
    ...
```



优化时再做化简处理

编号要求和方法

- 命名要求：
 - 每个变量（编号）只能定义一次
 - 纯数字编号（代码块+变量名）必须从%0开始，且连续
- 编号方法：
 - 翻译IR时为由于顺序影响，如难以保证编号连续性，避免重复即可
 - 按出现顺序（线性）重命名每一个代码块和变量名
 - 可读性考虑：
 - 代码块用bb编号或纯数字
 - 局部变量用%x名称或纯数字
 - 临时变量用%r1或纯数字

```
bb0:
    %x =
    ...
    %r6 =
bb1:
bb2:
    %r10 =
    %r11 =
bb3:
    %r13 =
    %r14 =
```

```
1:
    %2 =
    ...
    %7 =
8:
9:
    %10 =
    %11 =
12:
    %13 =
    %14 =
```

三、解释执行

解释执行

- 解释执行对象：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IR指令并执行

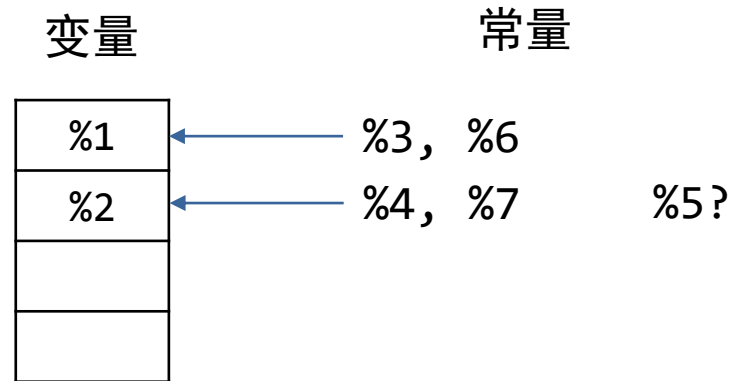
```
enum {  
    loadInst,  
    addInst,  
    subInst,  
    mulInst,  
    divInst,  
    brInst,  
    callInst,  
    ...  
} instType;
```

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

关键问题

- 如何保存每条指令的运行效果？
- 维护变量表：alloca的变量或所有变量？

```
%1 = alloca i32
%2 = alloca i32
%3 = load i32 %1;
%4 = load i32 %2;
%5 = add i32 %3, %4;
store i32 %5, %2;
%6 = load i32 %1;
%7 = load i32 %2;
%5 = add i32 %6, %7;
```



传统虚拟机指令和解释执行

```
Load a  
Load b  
Add  
Store c
```

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

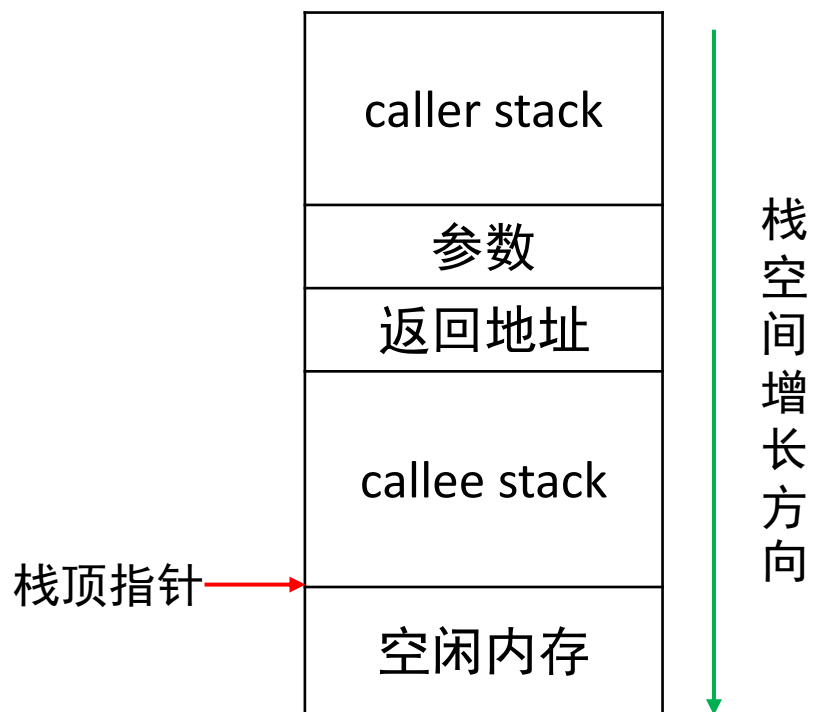
栈版本

```
stack s;  
loadInst => {  
    s.push(*arg1);  
}  
addInst => {  
    v1 = s.pop();  
    v2 = s.pop();  
    v2 = v1 + v2;  
    s.push(v2);  
}  
storeInst => {  
    v1 = s.pop ();  
    *arg1 = v1;  
}
```

寄存器版本

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（alloca）
- 函数返回后收回



```
fn foo -> &i32(){  
    let i:int = 100;  
    ret &i;  
}
```

Bug!!!

逃逸分析?

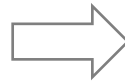
虚拟机

- 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
- 比较有名的虚拟机：
 - Java: HotSpot、Dalvik（Android）
 - Javascript: Chrome v8、Chakra、SpiderMonkey
- 虚拟机优化思路：
 - 使用寄存器储存临时变量
 - Threaded code
 - JIT优化

Threaded Code

- while-match的问题：需要两次跳转
 - 1：跳转到分支代码
 - 2：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

```
while(1) {  
    match (*pc++) {  
        addInst => { ... }  
        subInst => { ... }  
        ...  
    }  
}
```



```
static fn add() {  
    ...  
    (*++pc.fnaddr)();  
}  
...  
(*pc.fnaddr)();
```

