CS30017 编译

第七讲：线性IR

徐 辉

xuh@fudan.edu.cn

大纲

一、线性IR：LLVM IR

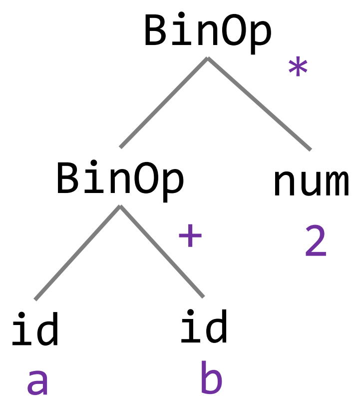
二、翻译线性IR

三、解释执行

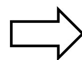
一、线性IR定义

线性IR的基本形式

- 指令名 + 参数
  - 参数：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



抽象语法树



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

三地址线性IR

Tea语言的IR

- 选取LLVM IR的子集
  - LLVM IR参考: https://llvm.org/docs/LangRef.html
- **好处：** 可使用lli工具执行IR

```
@g = global i32 10

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

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

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

标识符和基础类型

- 全局变量/函数名称：@name
- 局部变量/临时变量：%x、%0（不可重复，数字编号需连续）

```
@g = global i32 10

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

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

- `@g = global i32 10` → 创建全局变量g
- `define i32 @fib(i32 %0) {` → 创建函数fib
- `%x = alloca i32` → 创建局部变量%x
- `%g0 = load i32, ptr @g` → 创建临时变量%g0

数据存取

- 类型：void、i32、i32*、i8、i8*、i1、ptr
- 栈空间分配：alloca
- 数据存取：load/store

```
@g = global i32 10

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

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

- `@g = global i32 10` → 全局变量@g默认直接分配
- `%x = alloca i32` → 为局部变量%x分配空间，类型为ptr/i32*
- `store i32 %0, ptr %x` → 将临时变量%0保存到%x指向的内存中
- `%g0 = load i32, ptr @g` → 加载%g指向内存为临时变量%g0

函数

- 定义：define
- 调用：call
- 返回：ret

```
@g = global i32 10

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

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

函数fib：类型为(i32)->i32

返回%g0

函数main：类型为(void)->i32

调用函数fib

函数声明

- 声明：declare
- 声明和定义不能在一个ll文件中，使用llvm-link工具链接

```
declare i32 @fib( i32 )

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

在a.ll文件中声明函数fib

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

在b.ll文件中定义函数fib

```
#: llvm-link a.ll b.ll -o c.ll
```

数组类型存取

- 获取地址：getelementptr

```
%1 = alloca [2 x i32]
%2 = getelementptr [2 x i32],
     ptr %1,
     i32 0,
     i32 0
store i32 99, ptr %2
%3 = load i32, ptr %2
```

创建一维数组

数组基地址

索引为0的元素

```
@a = global [2 x i32] [i32 1, i32 2]
```

全局数组声明和初始化

结构体类型数据存取

```
%mystruct = type { i32, i32 }
define i32 @main() {
  %1 = alloca %mystruct
  %2 = getelementptr %mystruct,
      ptr %1,
      i32 0,
      i32 0
  store i32 1, ptr %2
  ret i32 0
}
```

定义mystruct数据类型

创建mystruct类型的对象

获取mystruct第一个成员的指针

更多关于GEP

- getelementptr指令的参数个数可以变化
- https://llvm.org/docs/GetElementPtr.html

```
let x = p[1].f1;
```

```
%t0 = getelementptr %struct.foo, ptr %p, i32 1, i32 0
```

```
%p1 = getelementptr %struct.foo, ptr %p, 1
%t0 = getelementptr %struct.foo, ptr %p0, i32 0, i32 0
```

算数运算

- 加、减、乘、除：add/sub/mul/sdiv
- 浮点数运算：fadd/fsub/fmul/fdiv

```
%3 = add i32 %2, 1
%4 = sub i32 %3, 2
%5 = mul i32 %3, 3
%6 = sdiv i32 %4, 4
```

关系（比较）运算

- 一条指令：icmp
- 多种参数：sgt/sge/slt/sle/eq/ne

```
%3 = icmp sgt i32 %2, 0
%4 = icmp sge i32 %2, 0
%5 = icmp slt i32 %2, 0
%6 = icmp sle i32 %2, 0
%7 = icmp eq i32 %2, 0
%8 = icmp ne i32 %2, 0
```

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

类型转换

- 扩充：zext
- 截断：trunc

```
%a = alloca i32
%b = alloca i8
%t0 = load i32, ptr %a
%t1 = icmp ne i32 %t0, 0
%t2 = zext i1 %t1 to i32
store i32 %t2, ptr %a
%t3 = trunc i32 %t2 to i8
store i32 %t2, ptr %b
```

类型转换：i1=>i32

类型转换：i32=>i8

逻辑运算

- 方式一：使用位运算
- 方式二：转化为控制流（短路）实现

```
%r = xor i1 %a, true
%r = and i1 %a, %b
%r = or i1 %a, %b
```

控制流指令

- 直接跳转：br + 目标
- 条件跳转：br + 条件 + 目标1 + 目标2

```
  %2 = alloca i32
  store i32 0, ptr %2
  %3 = load i32, ptr %2
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %bb1, label %bb2        → 条件跳转
bb1:
  store i32 1, ptr %2
  br label %bb3                           → 直接跳转
bb2:
  store i32 0, ptr %2
  br label %bb3
bb3:
  %r0 = phi i32 [0, %bb1], [%3, %bb2]
  ret i32 %r0
}
```

数据流指令

- 条件赋值：phi

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

如前序代码块为%bb1，则%r0=0
如前序代码块为%bb2，则%r0=%3

基于控制流实现逻辑或和与

```
bb1:
  br i1 %a, label %bb2, label %bb3
bb2:
  br label %bb3
bb3:
  %10 = phi i1 [false, %bb1], [%b, %bb2]
```

→ %a && %b

```
bb1:
  br i1 %a, label %bb3, label %bb2
bb2:
  br label %bb3
bb3:
  %10 = phi i1 [true, %bb1], [%b, %bb2]
```

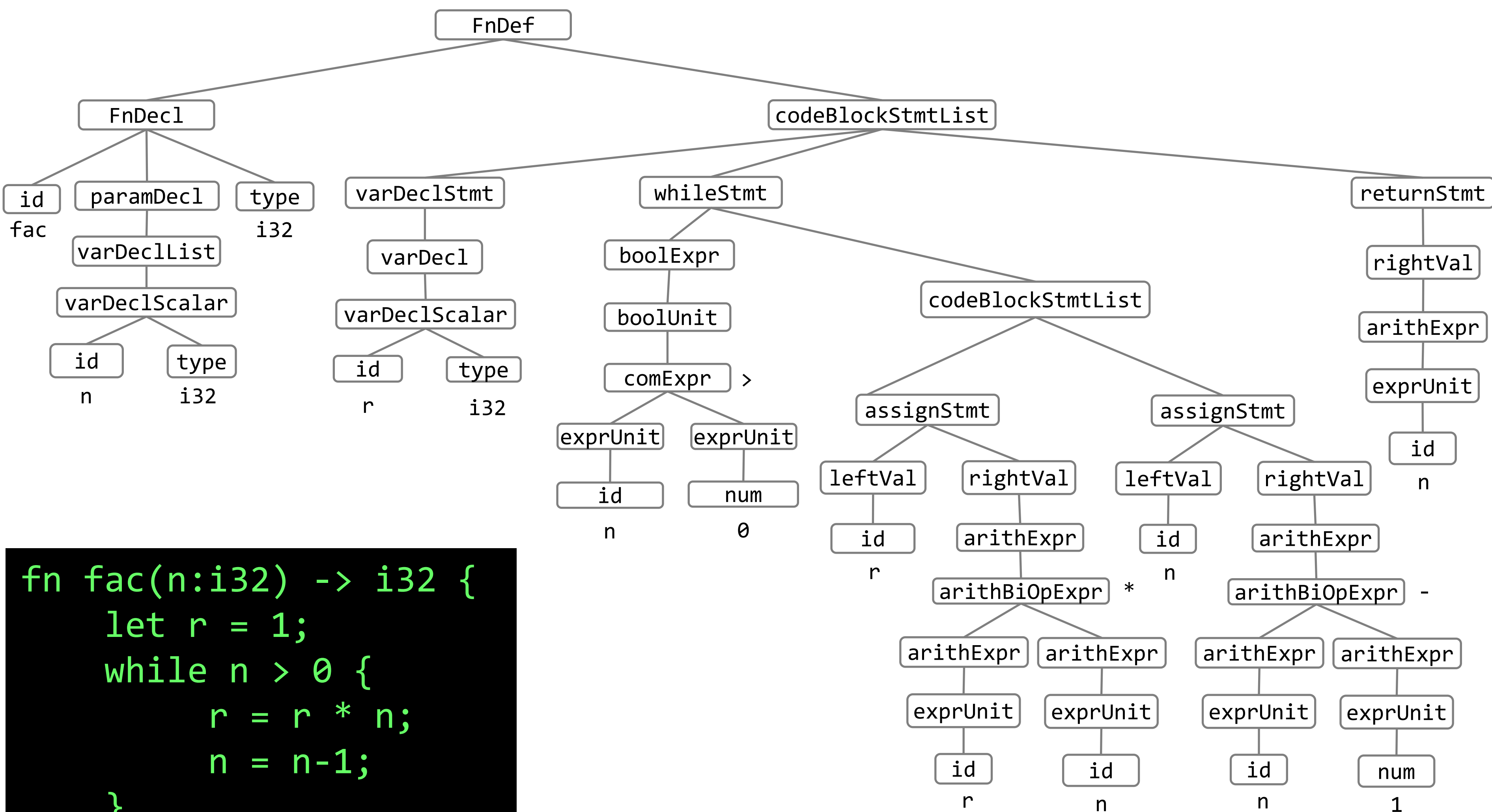
→ %a || %b

练习

- 使用上下文无关文法定义LLVM IR语法规则

二、翻译线性IR

思考：如何将AST翻译为线性IR




```
fn fac(n:i32) -> i32 {
    let r = 1;
    while n > 0 {
        r = r * n;
        n = n-1;
    }
    return r;
}
```

AST=>LLVM IR

- 基本思路：
  1) 遍历AST，创建全局函数/变量IR
  2) 遍历函数AST，创建代码块编号
  3) 翻译每个代码块的内容
- 关键：
  - 代码块编号和引用（br）
  - 变量编号和引用（def-use）

代码块编号和引用

- 每个代码块都应以terminator结尾：br/ret

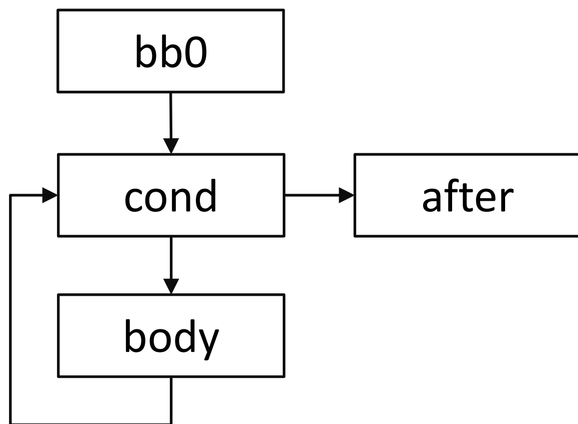
```
fn fac(n:i32) -> i32 {
    let r = 1;
    while n > 0 {
        r = r * n;
        n = n - 1;
    }
    return r;
}
```

```
define i32 @fac(i32 %0) {
bb0:
    ...
    br label %bb1
bb1: ; while cond (test)
    ...
    br i1 %cond? label %bb2, label %bb3
bb2: ; while body (true)
    ...
    br label %bb1
bb3: ; after while (false)
    ...
    ret
}
```

控制流嵌套的例子：递归下降编号

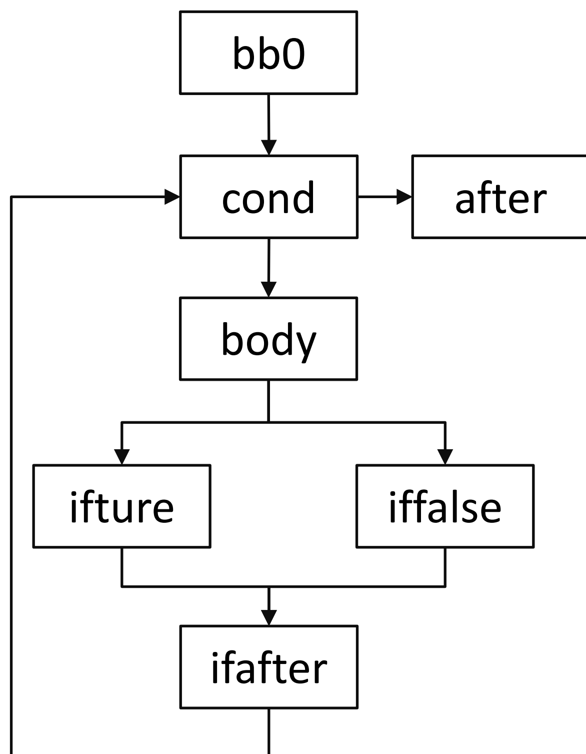
```
fn collatz(n:i32) -> i32 {
    while n != 1 {
        if n%2 == 0 {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    return n;
}
```

```
define i32 @collatz(i32 %0) {
bb0:
...
ret i32 ...
}
```



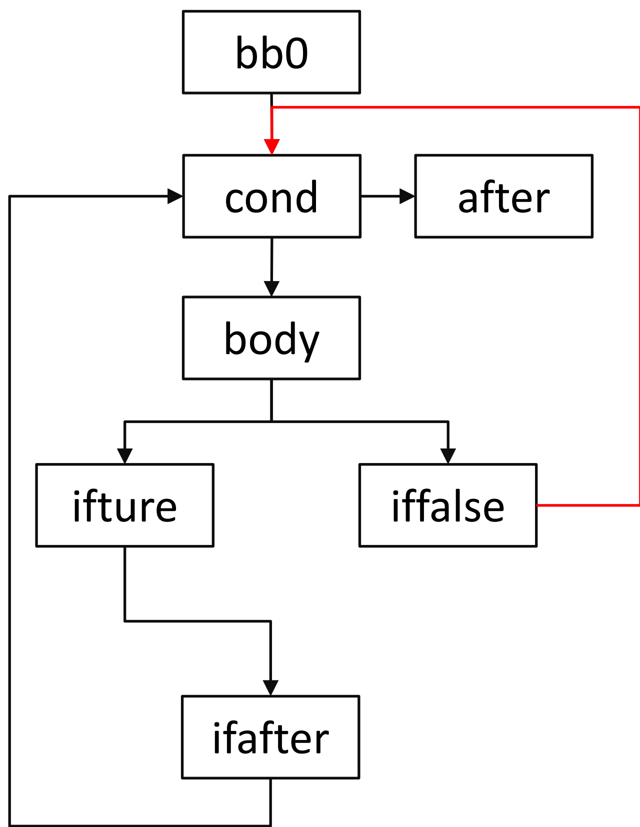

```
define i32 @collatz(i32 %0) {
bb0:
    ...
    br label %test
cond: ; while cond (test)
    ...
    br i1 %t1, label %body, label %after
body: ; while body (true)
    ...
    br %wcond
after: ; after while (false)
    ...
    ret i32 ...
}
```

控制流嵌套的例子：递归下降编号

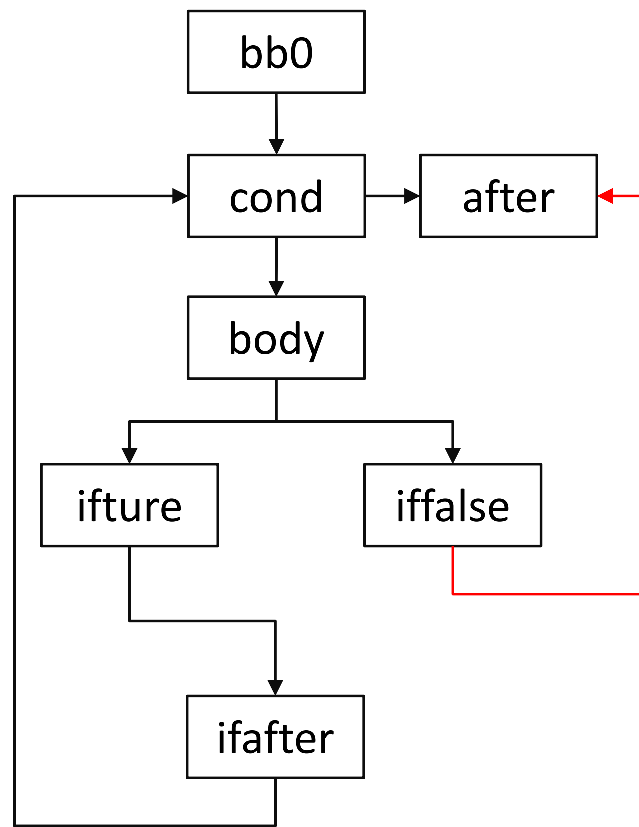



```
define i32 @collatz(i32 %0) {
bb0:
    ...
    br label %bb1
cond:    ; while condition
    br i1 %t1, label %body, label %after
body:    ; while body; if condition
    ...
    br i1 %t2, label %ifture, label %iffalse
ifture: ; if true branch
    ...
    br label %ifafter
iffalse: ; if false branch
    ...
    br label %ifafter
ifafter:
    br label %cond
after:
    ...
    ret %r
}
```

控制流嵌套：continue/break



continue



break

Teac代码：函数最后一条指令必须是return

```
43        /// Returns an error if the function is not registered in the type registry,
44        /// an argument name is redefined, or the return type is unsupported.
45        pub fn generate(&mut self, from: &ast::FnDef) -> Result<(), Error> {
46            let identifier = &from.fn_decl.identifier;
78            for stmt in &from.stmts {
79                self.handle_block(stmt, None, None)?;
80            }
81
82            // Append an implicit return if the last instruction is not already a
83            // return.  `FunctionType::try_from` whitelists return types to
84            // Void/I32 at registration, so any other variant here would be a
85            // broken invariant in the front-end.
86            if let Some(stmt) = self.irs.last() {
87                if !matches!(stmt.inner, StmtInner::Return(_)) {
88                    match &return_dtype {
89                        Dtype::I32 => self.emit_return(Some(Operand::from(0))),
90                        Dtype::Void => self.emit_return(None),
91                        other => unreachable!(
92                            "function {} has return type {other} which \
93                             FunctionType::try_from should have rejected",
94                            identifier
95                        ),
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/function_gen.rs

Teac代码实现

```
112      pub fn handle_block(
113          &mut self,
114          stmt: &ast::CodeBlockStmt,
115          con_label: Option<&BlockLabel>,
116          bre_label: Option<&BlockLabel>,
117      ) -> Result<(), Error> {
118          match &stmt.inner {
119              ast::CodeBlockStmtInner::Assignment(s) => self.handle_assignment_stmt(s),
120              ast::CodeBlockStmtInner::VarDecl(s) => match &s.inner {
121                  ast::VarDeclStmtInner::Decl(d) => self.handle_local_var_decl(d),
122                  ast::VarDeclStmtInner::Def(d) => self.handle_local_var_def(d),
123              },
124              ast::CodeBlockStmtInner::Call(s) => self.handle_call_stmt(s),
125              ast::CodeBlockStmtInner::If(s) => self.handle_if_stmt(s, con_label, bre_label),
126              ast::CodeBlockStmtInner::While(s) => self.handle_while_stmt(s),
127              ast::CodeBlockStmtInner::Return(s) => self.handle_return_stmt(s),
128              ast::CodeBlockStmtInner::Continue(_) => self.handle_continue_stmt(con_label),
129              ast::CodeBlockStmtInner::Break(_) => self.handle_break_stmt(bre_label),
130              ast::CodeBlockStmtInner::Null(_) => Ok(()),
131          }
132      }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/function_gen.rs

Teac代码实现：While语句

```
390 ⌄     pub fn handle_while_stmt(&mut self, stmt: &ast::WhileStmt) -> Result<(), Error> {
391           let test_label = self.alloc_basic_block();
392           let true_label = self.alloc_basic_block();
393           let false_label = self.alloc_basic_block();
394
395           // Jump unconditionally into the loop test from the predecessor block.
396           self.emit_jump(test_label.clone());
397
398           // Emit the loop condition test.
399           self.emit_label(test_label.clone());
400           self.handle_bool_unit(&stmt.bool_unit, true_label.clone(), false_label.clone())?;
401
402           // Loop body; `continue` → test_label, `break` → false_label.
403           self.emit_label(true_label);
404           self.enter_scope();
405           for s in &stmt.stmts {
406               self.handle_block(s, Some(&test_label), Some(&false_label))?;
407           }
408           self.exit_scope();
409           // Back-edge: jump back to the loop condition.
410           self.emit_jump(test_label);
411
412           self.emit_label(false_label);
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/function_gen.rs

Teac代码实现：If语句

```
341       pub fn handle_if_stmt(
342           &mut self,
343           stmt: &ast::IfStmt,
344           con_label: Option<&BlockLabel>,
345           bre_label: Option<&BlockLabel>,
346       ) -> Result<(), Error> {
347           let true_label = self.alloc_basic_block();
348           let false_label = self.alloc_basic_block();
349           let after_label = self.alloc_basic_block();
350
351           // Evaluate the condition; jump to the appropriate branch.
352           self.handle_bool_unit(&stmt.bool_unit, true_label.clone(), false_label.clone())?;
353
354           // Emit the then-branch; a new scope is opened so that any locals are cleaned up.
355           self.emit_label(true_label);
356           self.enter_scope();
357           for s in &stmt.if_stmts {
358               self.handle_block(s, con_label, bre_label)?;
359           }
360           self.exit_scope();
361           // Jump past the else-branch to the merge point.
362           self.emit_jump(after_label.clone());
363
364           // Emit the (possibly absent) else-branch in its own scope.
365           self.emit_label(false_label);
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/function_gen.rs

Teac代码实现：label编号

```
166        /// Allocates and returns a new unique [`BlockLabel::BasicBlock`] label, then
167        /// advances the internal counter.
168        pub fn alloc_basic_block(&mut self) -> BlockLabel {
169            let idx = self.next_basic_block;
170            self.next_basic_block += 1;
171            BlockLabel::BasicBlock(idx)
172        }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/function.rs

变量编号和引用

- 消除块与块之间的数据依赖关系
- 块内依赖：使用变量前先load，更新后立即store

```
fn fac(n:i32) -> i32 {
    let r = 1;
    while n > 0 {
        r = r * n;
        n = n-1;
    }
    return r;
}
```

```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, ptr %n
    store i32 1, ptr %r
    br label %bb1
bb1:
    %t1 = load i32, ptr %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, ptr %r
    %t4 = load i32, ptr %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, ptr %r
    ...
}
```

编号要求和方法

- lli要求：
  - 每个变量（编号）只能定义一次
  - 如果使用纯数字编号，必须从%0开始且连续（代码块和变量名共享）
- 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

Teac代码实现：计算编号

```
145        /// Allocates the next unique [`LocalId`] and advances the internal
146        /// counter.  Use [`fresh_local`] instead when a typed handle is wanted;
147        /// this raw-id form exists for passes (e.g. mem2reg's phi insertion)
148        /// that build their own [`Local`] by combining an id with a
149        /// context-specific type.
150        ///
151        /// [`fresh_local`]: FunctionGenerator::fresh_local
152        pub fn fresh_local_id(&mut self) -> LocalId {
153            let idx = self.next_vreg;
154            self.next_vreg += 1;
155            LocalId(idx)
156        }

162        pub fn fresh_local(&mut self, dtype: Dtype) -> Local {
163            Local::new(dtype, self.fresh_local_id())
164        }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/function.rs

Teac代码实现：自动load

```
480        /// Lowers a single expression unit to an [`Operand`].
481        ///
482        /// After resolving the unit's inner form, performs an implicit load for
483        /// addressable scalar pointers and for global `i32` values, so the caller
484        /// always receives a value-typed operand rather than a pointer.
485        fn handle_expr_unit(&mut self, unit: &ast::ExprUnit) -> Result<Operand, Error> {
486            let operand = match &unit.inner {

544            Ok(match operand.dtype() {
545                // Auto-load: dereference addressable scalar pointers (but leave arrays/structs as-is).
546                Dtype::Pointer { pointee }
547                    if operand.is_addressable()
548                        && !matches!(pointee.as_ref(), Dtype::Array { .. } | Dtype::Struct { .. }) =>
549                {
550                    let dst = Operand::from(self.fresh_local(pointee.as_ref().clone()));
551                    self.emit_load(dst.clone(), operand);
552                    dst
553                }
554                // Auto-load global i32 values which are stored behind a pointer.
555                Dtype::I32 if matches!(&operand, Operand::Global(_)) => {
556                    let dst = Operand::from(self.fresh_local(Dtype::I32));
557                    self.emit_load(dst.clone(), operand);
558                    dst
559                }
560                _ => operand,
561            })
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/function_gen.rs

Teac代码实现：自动store

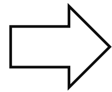
```
134        /// Lowers an assignment statement (`left = right`).
135        ///
136        /// `handle_left_val` yields a pointer to the destination's stack slot and
137        /// `handle_right_val` yields the value to store, so the assignment is a
138        /// single `store` instruction.
139        pub fn handle_assignment_stmt(&mut self, stmt: &AssignmentStmt) -> Result<(), Error> {
140            let left = self.handle_left_val(&stmt.left_val)?;
141            let right = self.handle_right_val(&stmt.right_val)?;
142            self.emit_store(right, left);
143            Ok(())
144        }
```

```
688        /// Resolves a left-hand-side value to an addressable [`Operand`] (a pointer).
689        ///
690        /// For a simple identifier, looks up the symbol; for array and member
691        /// expressions, delegates to the respective handlers.
692        fn handle_left_val(&mut self, val: &ast::LeftVal) -> Result<Operand, Error> {
693            match &val.inner {
694                ast::LeftValInner::Id(id) => self.lookup_variable(id),
695                ast::LeftValInner::ArrayExpr(expr) => self.handle_array_expr(expr),
696                ast::LeftValInner::MemberExpr(expr) => self.handle_member_expr(expr),
697            }
698        }
```

https://github.com/tea-compiler/teac/blob/main/src/ir/gen/function_gen.rs

IR翻译结果

```
fn fac(n:i32) -> i32 {
    let r = 1;
    while n > 0 {
        r = r * n;
        n = n-1;
    }
    return r;
}
```



```
define i32 @fac(i32 %0) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, ptr %n
    store i32 1, ptr %r
    br label %bb1
bb1:
    %t1 = load i32, ptr %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, ptr %r
    %t4 = load i32, ptr %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, ptr %r
    %t6 = load i32, ptr %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, ptr %n
    br label %bb1
bb3:
    %t8 = load i32, ptr %r
    ret i32 %t8
}
```

练习：翻译IR

```
fn collatz(n:i32) -> i32 {
    while n != 1 {
        if n%2 == 0 {
            n = n / 2;
        } else {
            n = 3 * n + 1;
        }
    }
    return n;
}
```

练习：翻译IR

```
let a[i32;10] = {1,3,5,7,9,2,4,6,8,10};
fn binsearch(x:i32) -> i32 {
    let high:i32 = 9;
    let low:i32 = 0;
    let mid:i32 = (high+low)/2;
    while a[mid]!=x && low < high {
        mid = (high+low)/2;
        if x<a[mid] {
            high = mid-1;
        } else {
            low = mid +1;
        }
    }
    if x == a[mid] {
        return mid;
    }
    else {
        return -1;
    }
}
```

```
fn main() -> int {
    let r = binsearch(2);
    ret r;
}
```

三、解释执行

解释执行

- 解释执行对象：线性IR
- 主要思路：
  - 找到程序入口，按照线性IR指令出现顺序和跳转关系执行
  - 遇到函数创建栈帧，为变量分配空间
  - 为全局变量分配空间

按照IR指令顺序执行

- 通过循环不断获取下一条IR指令并执行

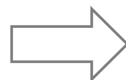
```
enum {
    loadInst,
    addInst,
    subInst,
    mulInst,
    divInst,
    brInst,
    callInst,
    ...
} instType;
```

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

使用Threaded Code

- while-match的问题：需要两次跳转
  - 跳转到分支代码
  - 返回循环入口
- 跳转一次：为每条指令设计一个处理函数或代码块

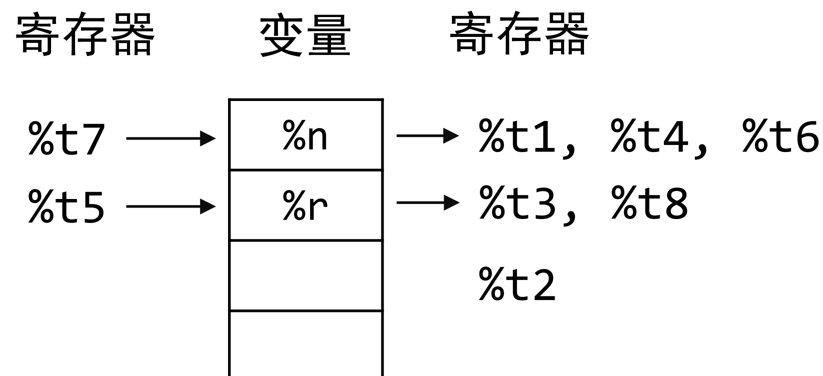
```
while(1) {
    match (*pc++) {
        addInst => { ... }
        subInst => { ... }
        ...
    }
}
```



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

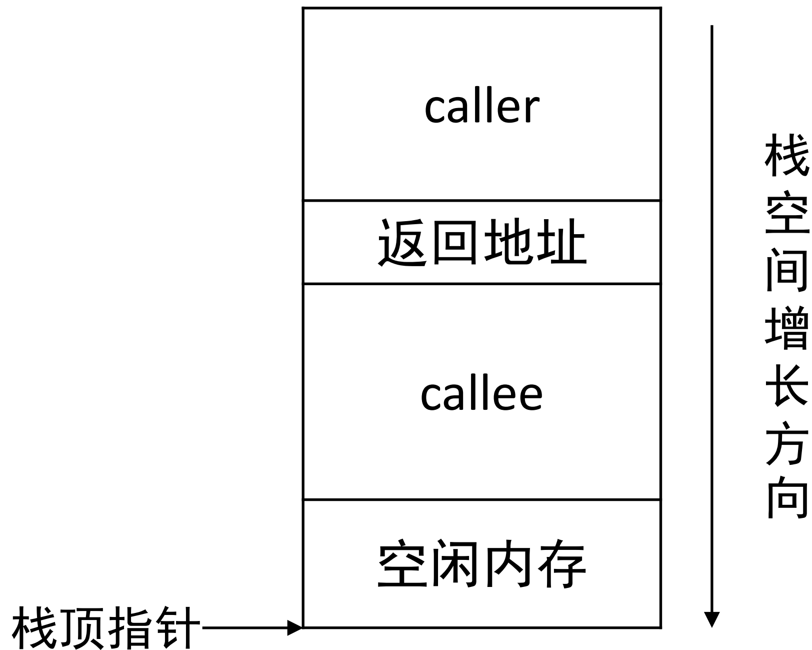
如何保存每条指令的运行效果？

```
define i32 @foo( i32 %0 ) {
bb0:
    %n = alloca i32
    %r = alloca i32
    store i32 %0, ptr %n
    store i32 1, ptr %r
    br label %bb1
bb1:
    %t1 = load i32, ptr %n
    %t2 = icmp sgt i32 %t1, 0
    br i1 %t2, label %bb2, label %bb3
bb2:
    %t3 = load i32, ptr %r
    %t4 = load i32, ptr %n
    %t5 = mul i32 %t3, %t4
    store i32 %t5, ptr %r
    %t6 = load i32, ptr %n
    %t7 = sub i32 %t6, 1
    store i32 %t7, ptr %n
    br label %bb1
bb3:
    %t8 = load i32, ptr %r
    ret i32 %t8
}
```



函数栈帧：Activation Record

- 栈帧：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）
- 栈帧空间在函数返回后收回



栈空间增长方向

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

Bug!!!

逃逸分析?

栈虚拟机/寄存器虚拟机

- LLVM IR为三地址IR，与Java Bytecode/WebAssembly不同

```
//Java Bytecode
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];
}
```

寄存器虚拟机实现方式

```
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;
}
```

栈虚拟机实现方式

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
  - WebAssembly: Wasmtime、Wasm3、Wasmer
- 虚拟机优化思路：
  - JIT优化
  - ...

Sun
microsystems

Meta
1 Hacker Way
ONE WAY

facebook
1 Hacker Way