

# 编译器构造

马融

# 课程计划

- 上课次数： 8 次左右
- 截止日期： 5 月 15 日前后
- 评分政策： 基础分 + Bonus
- 代码提交： GitHub
- 期末答辩：
- 助教
  - 陈乐群、游宇榕、徐晓骏、徐世超、谢天成、李慧琛
  - 通过 code review 帮助大家解决问题

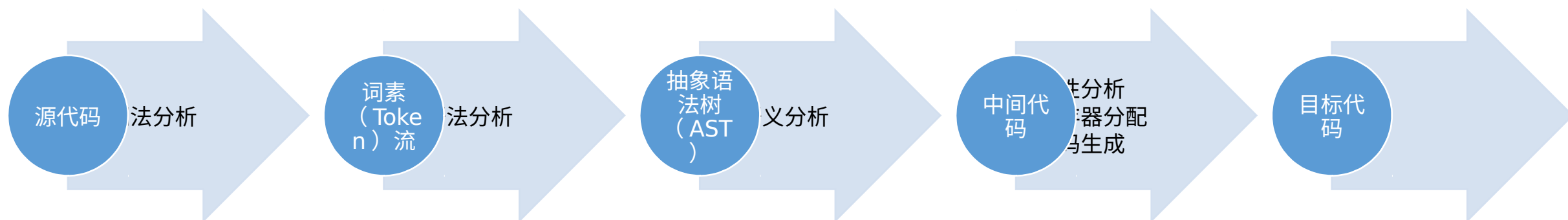
# Todo List

- 网站
- 分组
- 论坛
- 日程表
- 数据
- 语法说明
- 评分细则

# 参考书籍

- 《自制编译器》——青木峰郎
- 虎书——《现代编译程序实现》
- **PLP**——《程序设计语言——实践之路》
- LIP——《编程语言实现模式》 Terence Parr
- CaC——《Crafting a Compiler》 Charles N. Fischer
- EaC ——《编译器设计》 Keith D. Cooper

# 编译器的各个阶段



# C 语言的精神

- 信任程序员
  - Trust the programmer
- 不要阻止程序员去做需要的事
  - Do not prevent the programmer from doing what needs to be done
- 保持语言的小巧和简单
  - Keep the language small and simple
- 为效率可以牺牲可移植性
  - Make it fast, even if it is not guaranteed to be portable

只要能让程序员干的活  
编译器一律不管

C 语言的精神（里番）

# 为什么这么设计



- 640K is enough for everyone
- One-pass compiler
  - 先申明再使用 = 不能向后看
  - 信任程序员 = 优化神马的做不到呀



# 现代语言的一些特性

- 垃圾收集
  - 面向对象
  - 隐式指针
  - 原生 String
- 
- 一种语言包打天下的时代已经过去了

Bonus：假设所有的编译器都坏了，而且所有的源代码都不见了，但电脑都能工作，我们需要多少时间来重建 C++ 编译器？

(这个问题轮子哥在知乎上的回答是错误的)

造轮子的时候能不能用轮子？

# 自展 Bootstrapping

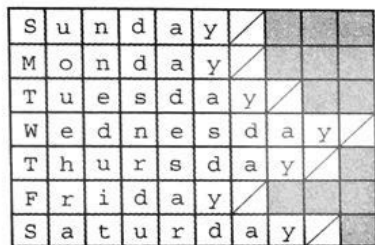
	源语言	编译器语言	目标语言
A 组	汇编	机器语言	机器语言
B 组	C--	汇编	汇编
C 组	C	C--	C--
D 组	C++	C	C

# “用 X 语言来写一个 Y 语言的编译器”

- 假设  $X = C$ 
  - `int MAMA_nikannagebianliangminghaochang = 1;`
- 假设  $X = C++$ 
  - 如果内存泄漏一次扣一分的话，全班的分数还不够给一个人扣的
- 假设  $Y = C$ 
  - C 已经过时了
- 假设  $Y = Tiger$ 
  - 抄袭太容易了
- 结论  $X = Java$ （不强迫）  $Y =$  新的语言，代号 ~~M~~ $\Theta Mx^*$

# 交错数组 PLP P375

```
char days[][10] = {  
    "Sunday", "Monday", "Tuesday",  
    "Wednesday", "Thursday",  
    "Friday", "Saturday"  
};  
...  
days[2][3] == 's'; /* in Tuesday */
```



```
char *days[] = {  
    "Sunday", "Monday", "Tuesday",  
    "Wednesday", "Thursday",  
    "Friday", "Saturday"  
};  
...  
days[2][3] == 's'; /* in Tuesday */
```

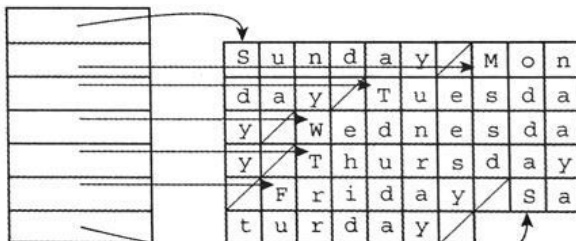


图 7.11 C 语言里的行指针数组和连续数组分配方式。左边声明的是一个真正的二维数组，其中画斜线的格子是 NUL 字节，画阴影的区域是空洞。右边声明的是一个指向字符数组的指针数组。对于这两种情况，声明中都忽略了界的描述，因为它们可以从初始化描述（聚集值）的大小中推导出来。两种数据结构都允许通过两重下标去访问单个的字符，但它们的内存布局（以及对应的地址算术）则是大不相同的。

# 谈论一些和分数有关 的小事

# 分数结构

- 客观分： 100 分
- 主观分： -5 分 ~+5 分
- 客观分中：
  - “低保”： 85 分
  - “天梯”： 15 分



# “低保”

- 数据公开，由助教给出
- 编译时间和运行时间合理即可
  - 合理=稍作优化
- 扣分：
  - 晚交：第一天扣 1 分，第二天扣 2 分，第三天扣 3 分……
  - 超时：

# 天梯

- 每个学生必须贡献一个数据，这个数据会以该同学的名字命名并公开，而且会流传到下一届
- 这个数据必须符合规范，不涉及任何未定义的内容
- 目标
  - 卡其他人的编译时间和运行时间
  - 让自己编译时间和运行时间尽量短
- 双 5s 原则：
  - 自己的编译时间  $< 5s$
  - 自己的运行时间  $< 5s$

# 天梯排名的计算依据

- 假设有编号为 A, B, C, D 的测试点
- 小明在 A 测试点的运行时间为 4s，全班排名第 1，+30 分
- 小明在 B 测试点的运行时间为 8s，全班排名第 2，+18 分
- 小明在 C 测试点的挂了，+0 分
  - 挂 1 = 编译时间  $\geq 15s$
  - 挂 1 = 运行时间  $\geq 15s$
- 小明在所有测试点上的分数之和，就是天梯排名的唯一依据

# 天梯排名活动结束后获得的分数

- 第一名： +15 分
- 第二名： +14 分
- 第三名： +13 分
- 第四名、第五名： +11 分
- 第六名、第七名： +9 分
- 第八名~第十名： +7 分
- 第十一名~第十五名： +5 分
- 第十六名~第二十名： +3 分
- 之后： +1 分

# 约定

- 所有测试程序都需要公开
  - 测试程序公开截止日期：5月1日
  - 如果 TA 认为数据不合适，会联系修改
- 排名开始日：5月1日
- 排名结束日 = 最晚提交截止日
- OJ 会每天公布每人的分数明细，但不公布运行时间
- 最后一天的排名是有用，前几天都只作为参考
- 所有编译器的源代码在排名结束之后，也必须公开

# 测试数据参数

- 编译时间 <5s
- 运行时间 <5s
- 源文件大小 <1M
- 目标文件运行时大小 <128M
- 不要在编译器里做陷阱

Bonus：要做些什么事情才能获得 Bonus？

# 助人为乐者可以获得 BONUS

- 不要吝啬分享知识，有输入还需要输出
- 授予乐意助人，耐心解答同学问题的人
  - 可以是线上文字材料，也可以是线下活动
- 提名方式：自评或者他评都可以



# 词法分析

```
While(Level>=Wallace)excited++;
```

```
while ( Level >= Wallace ) excited ++ ;
```

# Token (词素)

- Type
  - 保留字
  - 运算符, 分隔符 (如 ;)
  - 标识符 (ID)
  - 常数 (Num, Str)
- Semantic Value
  - ID , Num 和 Str 需要更多信息
- row 和 col
  - 用于编译器报错定位

```
public class Token {  
    public int row, col;  
    public int type;  
    public String text;  
}
```

# Scanner

```
public Token nextToken() {
    while stream.peek() = 空格 do stream.consume();
    if stream.peek() = EOF the return <<EOF>>;
    switch(stream.peek()){
        case ',': stream.consume(); return <<COMMA>>;
        case '+': stream.consume();
            if stream.peek() = '+' // ++ 运算符
                stream.consume(); return <<PLUSPLUS>>;
            else return <<PLUS>>;
        case '0'..'9': return matchDigit();
        case 'a'..'z': return matchLetter();
        case '"': return matchLiteral();
        default : throw 异常
    }
}
```

```
public Token matchLetter() {
    String text = stream.consume();
    while stream.peek() = 字母, 数字, 下划线 {
        text += stream.consume();
    }
    if (text = 关键字 )
        return <<关键字>>
    else
        return <<ID, text>>
}
// text 用 StringBuilder 会更自然一点
```

# 其他要点

- 单行注释
- 大嘴法
- 假设源代码中反复出现一段字符串常量—— Four score and seven years ago—— 是否需要压缩 string space ?
  - 最高效: hashtable
  - 最优雅: string.intern()
  - 最方便: 不理他

# 语法分析

destiny ::= T00 unpredictable

unpredictable ::= NAIVE | SIMPLE | ANGRY | EXCITED

# Context-Free Grammars

- 非终结符——小写
- 终结符——Token，大写
- 语法分析树

$\text{exp} ::= \text{exp op exp} \mid \text{ID}$

$\text{op} ::= + \mid - \mid * \mid /$

- ~~推倒~~推导 (derivation)

$\text{exp} \Rightarrow \text{exp} + \text{exp}$

$\quad \Rightarrow \text{exp} + \text{exp} * \text{exp}$

$\quad \Rightarrow y + k * x$

# 语法——语句

```
stmt ::= block_stmt
      | var_decl_stmt
      | expr_stmt
      | IF (expr) stmt
      | IF (expr) stmt ELSE stmt
      | RETURN expr; | BREAK; | CONTINUE;
      | WHILE (expr) stmt
      | FOR (expr; expr; expr) stmt
```

# 语法——表达式

```
expr ::= additive_expr  
      | logical_expr  
      | relational_expr  
      | unary_expr
```

- 左：符合阅读习惯，但不能表示优先级
- 下：能表示优先级，但不能表示结合性

```
expr          ::= logical_and_expr  
log_and_expr  ::= log_and_expr && log_and_expr  
              | relational_expr  
relation_expr ::= relation_expr < relation_expr  
              | additive_expr  
additive_expr ::= additive_expr + additive_expr  
              | ...
```



# 语法——表达式

```
expr ::= creation_expr
      | call_expr      // "naive".length()
      | binary_expr
      | unary_expr     // ++expr, expr++, ~expr
      | primary        // ID 和 literal
      | array_index
      | field_access
      | null
```

# 语法——类和函数声明

```
class_decl ::= CLASS ID { member_decl_stmt_list }
member_decl_list ::= type ID;
                  | type ID; member_decl_list
func_decl ::= type ID (param_decl_list) block_stmt
            | VOID ID (param_decl_list) block_stmt
param_decl_list ::= type ID
                  ::= type ID, param_decl_list
type ::= INT | STRING | ID | type []
```

# 语法——Block

```
block_stmt      ::= { stmt_list }  
stmt_list      ::= stmt  
                | stmt stmt_list  
var_decl_stmt  ::= type ID;  
                | type ID = expr;
```

# 语法—— Program

```
program ::= class_decl program  
         | func_decl program  
         | var_decl_stmt program
```

# 细节

- 避免左递归
  - (Y) `stmt_list ::= stmt`
  - `| stmt stmt_list`
  - (N) `stmt_list ::= stmt_list`
  - `| stmt_list stmt`
- 是否允许一个 `stmt` 和 `exp` 为空?
- BNF 语法可参考
  - <http://acm.sjtu.edu.cn/compiler/start>
  - <http://bcmi.sjtu.edu.cn/~mli/tiger/> ( 很老的资料在这里 )
  - Google 一下 java/C/C++/C# 的 BNF

# 抽象语法树

```
Life (  
    Mayor(SH), Prof(SH, retired), Chairman(BJ, things[3])  
)
```



# 抽象语法树

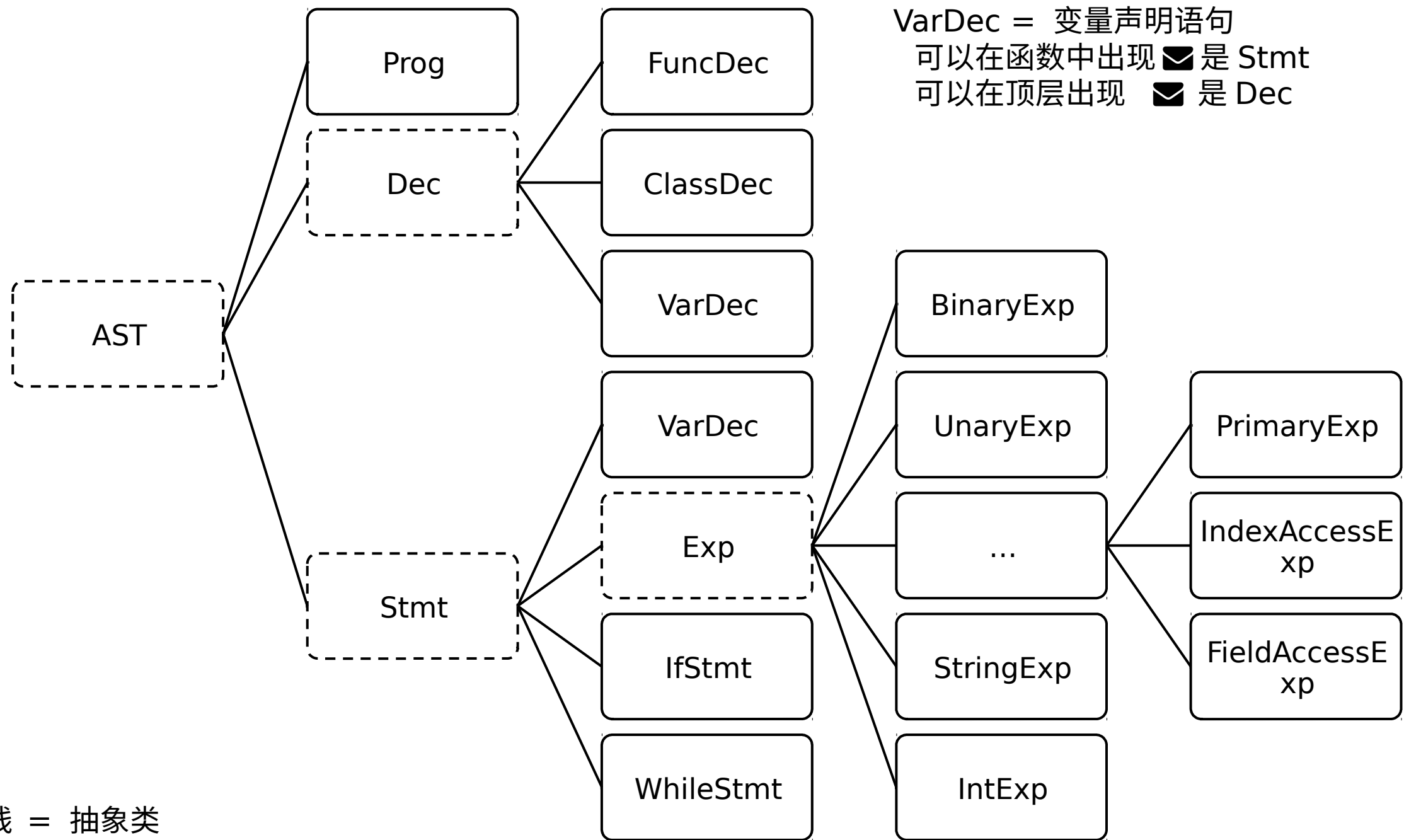
- 词法分析去除了一部分冗余
  - 换行，空白，注释
- 但还有一部分冗余存在
  - 各种关键字、分隔符
- 去除所有冗余之后，用树来表示程序的数据结构成为抽象语法树
  - 具体语法树？就是哪些冗余还在……
- 抽象语法树（AST）是个很重要的东西
  - 承上启下，前端的最终产物，后端的开始
  - 是研究 OO 设计方法论的非常好的一个实例（Design Pattern）

# 任务一：设计 AST ， 分清继承关系

- 基类： AST
- 子类
  - Stmt 表示一条语句
  - Exp 表示一条表达式
  - Dec 表示一条声明



VarDec = 变量声明语句  
可以在函数中出现  是 Stmt  
可以在顶层出现  是 Dec



# 如何优雅地避免泛型？

```
class Program extends AST {  
    List<Dec> list;  
}
```

```
class Program extends AST {  
    Dec head;  
    Program tail;  
}
```

# Design Pattern—— 组合模式

```
class BinaryExp extends Exp {  
    Exp left, right  
    int op;  
}
```

- 继承： BinaryExp 继承了 Exp
- 组合： BinaryExp 的两个成员也是 Exp
- 组合和继承如何取舍，是 OO 设计里永恒的话题

# Java 支持多继承吗？~~支持啊~~ 不支持啊

- VarDec 需要同时继承 Stmt 和 Dec

- 法一：把 Dec 和 Stmt 中一个变成 interface

- 公共祖先 AST 必须是 interface，就不能有 position 这个成员了，用 getPos 代替

- 法二：VarDec 不继承 Dec，修改 Prog 的代码：

```
class Program extends AST {  
    List<Dec> list;    ==>    List<AST> list;  
}
```

- 法三：【继承和组合】

- 新建一个 VarDecStmt 继承 Stmt，
    - VarDec 继承 Dec，除了持有 VarDecStmt 外什么都没有

# 备查：其他没有在图上出现的 AST

- BreakStmt
- ContinueStmt
- ReturnStmt
- ForStmt
- CreationExp
- NullExp

## 任务二：设计 AST 的方法

- toString()
  - Debug 的时候，一般需要把 AST 打印出来看看有没有出错
- check()
  - 检查语义是否有问题——二目运算符两侧类型是否一致，函数调用时参数数量和类型是否一致等等
- translate()
  - 将 AST 翻译成中间代码
- prettyPrint()
  - 支线任务，输出排版优美的源代码

# 方法一：内嵌式

```
class BinaryExp extends Exp {
    Exp left, right; int op;
    String toString(int d) {
        return indent(d)+ "Binary" + <<op>> + "\n"
            + left.toString(d+1)
            + right.toString(d+1);
    }
}

class Prog extends AST {
    Dec head; Prog tail;
    String toString(int d = 0) {
        return head.toString(d) + tail?.toString(d);
    } // ?. 是 C# 的文法糖，意思是如果 tail 不为空，调用 toString
}
```

```
class ForStmt extends Stmt {
    Exp start, cond, loop; Stmt body;
    String toString(int d) {
        return indent(d)+ "For" + "\n"
            + start?.toString(d + 1)
            + cond?.toString(d + 1)
            + loop?.toString(d + 1)
            + body.toString(d + 2);
    }
}

class FieldList extends AST {
    Field head; FieldList tail;
    String toString(int d) {
        return head.toString(d) + tail?.toString(d);
    } // FieldList 用于声明类的成员
}
```

# 内嵌式的设计缺陷

```
class BinaryExp extends Exp {
    Exp left, right;
    int op;
    String toString(int d) { }
    bool check() {}
    IR translate() {}
    void print() {}
}

class UnaryExp extends Exp {
    Exp child;
    int op;
    String toString(int d) { }
    bool check() {}
    IR translate() {}
    void print() {}
}
```

- 维护不便
  - 每个 AST 里面都要写上各自的 N 种方法
  - 《CaC》P265：Java 的语法有 50 种 AST，而 GCC 有 200 多个 Phase
  - 我们希望最好是把 AST 写死，而不是在开发到一半的时候，经常往上加东西
- ~~实际上这些指责都有些吹毛求疵~~



## 方法二： Downcast

```
class Printer {
    String visit(UnaryExp e, int d) {
        return indent(d)+ "Unary" + <<op>> + "\n"
            + visit(e.child, d + 1);
        // Double Dispatch problem
    }
}

String visit(WhileStmt e, int d) {
    return indent(d)+ "While" + "\n"
        + visit(e.cond, d + 1)
        + visit(e.body, d + 2);
}

.....
```

```
String visit(AST e, int d = 0) {
    // 没有 double dispatch, 编译器不会自动 Downcast
    if (e instanceof BinaryExp)
        return visit((BinaryExp) e, d);
    if (e instanceof IntExp)
        return visit((IntExp) e, d);
    if (e instanceof StringExp)
        return visit((StringExp) e, d);
    if (e instanceof WhileStmt)
        return visit((WhileStmt) e, d);
    if (e instanceof ForStmt)
        return visit((ForStmt) e, d);
    .....
    throw new Error(...);
}
```

# Double Dispatch

```
visitor.visit(Exp);
```

- Single Dispatch

- 选择 `visit` 的时候仅根据 `visitor` 的运行时型别 (Run time type)
- 简单的多态概念
- Java 和 C++ 只支持这个

- Double Dispatch

- 不但要根据 `visitor` 的类别, 还要根据 `Exp` 的运行时类别

- Multiple Dispatch

- 所有参数都考虑

# 方法三：访问者（Visitor）模式

- 主要目的
  - separating an algorithm from an object structure (WIKI)
  - 隔离“遍历过程”和“树的定义”
  - “使用 Ruby 或 Python 就很容易做到，因为支持运行时动态添加方法”《LIP》，P106
- 访问者模式是用来克服 Java 或 C++ 的固有缺陷的
  - 总的来说，很多设计模式就是用来解决语言的先天问题的
  - （陈乐群）C++ 和 Java 只支持 Single dispatch 是为了效率的折衷

# 树的定义

```
abstract class AST {
    abstract void accept(Visitor v);
    // 基类的调度方法是抽象的
}

class BinaryExp extends Exp {
    Exp left, right; int op;
    void accept(Visitor v) { v.visit(this); }
}

class UnaryExp extends Exp {
    void accept(Visitor v) { v.visit(this); }
}

// 小白：为何不把基类的 accept 方法写成具体的呢？这样
// 子类不就可以直接继承基类的方法而不必到处重写了？
```

# 树的遍历

```
class Visitor {
    void visit(AST e) { e.accept(this); }
}

class Printer extends Visitor {
    StringBuilder buf;
    int depth;
    void visit(UnaryExp e) {
        buf.append(indent(depth)+ "Unary" + <<op>> +
"\n");
        depth++;
        visit(e.child) // 变相做到了 double dispatch
        depth--;
    }
}
```

# 总结

- 内嵌式
  - 直接但不优雅，参考《编程语言实现模式》的模式 12
- Downcast 方法
  - 参考虎书第一版（第一版 Tiger，第二版 MiniJava）可以在 <http://bcmi.sjtu.edu.cn/~mli/tiger/>
  - 下到电子版以及原书附送的代码，看 Absyn 包
  - 《CaC》P267，P277 的 Ex20 和 Ex21
- 访问者
  - 号称是最复杂的设计模式，炫酷屌炸天，用起来还是有点痛苦的
- 究极方法是用个 compiler writer 来自动生成各种 visitor，比如 antlr

# 符号表

Rain = Journalist.lookup(Wind)

# 三个概念

- 名字
  - 变量名，函数名，类型名
- 类型
  - 基础类型，数组，结构
- 作用域
  - 全局，函数作用域，类作用域，局部作用域
- 符号表=在某个作用域里，找某个名字对应的类型和其他信息

# String Interning——Flyweight Pattern

- 资料
  - wiki "Flyweight pattern" "String Interning"
  - 虎书第二版 Ch5.1
- 构造 Name 的时候，为每个字符串分配一个唯一的 code
- 优势
  - 比较 code 比较字符串快
  - 相同字符串只需要存储一份



# 虎书第二版 P110

```
class Name {
    String name;
    private static Dictionary<String, Name> dict
        = new Hashtable<String, Name>;
    private Name(String text) { name = text; }           // 私有化构造函数
    public static Name getSymbolName(String text) {
        String unique = text.intern(); // 虎书上有这句话。游宇榕指正，这句话其实可以不要
        Name s = dict.get(unique);      // 陈乐群进一步指出，如果要 intern，可以用 IdentityHashMap
        if (s == null) {
            s = new Name(unique);
            dict.put(unique, s);
        }
    }
} // 问题：说好的唯一 code 在哪里？ Name 的内存地址就是唯一 code
```

# 类型的好处

- 《EaC》 P122
  - 安全：确保运行时安全
    - 类型推断：为每个表达式确定类型的过程
    - 隐式转换 VS 显式转换：我们要求显示
  - 表达力：提高表达能力
    - 但我们不允许用户重载运算符
  - 效率：生成更好的代码

# 类型的组件

- 内建类型
  - Int
  - String
  - Bool
- 复合类型
  - 数组
  - 结构

# 符号类型的声明

```
class Type {}
class IntType extends Type;
class ArrayType extends Type {
    Type element;
    // Type bounds;
    // 如果允许枚举类型做下标，还需要记录下标的类型
};
class ClassType extends Type {
    Type head;
    ClassType tail;
}
```

# typedef

- 如果有 typedef，那么每种类型还有个 actual 类型
- 例如 typedef LL long long
- LL 这个类型的 actual 类型其实是 long long

```
class Type {
    public Type actual() {return this;}
}
```

- 求 actual 时，不要忘了路径压缩（类似于并查集）

# 符号表的接口设计——虎书版

```
class SymbolTable {  
    void put(Name key, Object Value);  
    Object get(Name key);  
    void beginScope();  
    void endScope();  
}
```

- Table 是一个全局变量
- 一个典型的 KV 表 + 可持久化的数据结构
- 当执行 endScope 的时候，需要和最近一次 beginScope 配对，消除之间的所有 put 操作

# 双链表实现， from 虎书源代码

```
public Object get(Name key) {
    Binder e = dict.get(key);
    return e?.value;
}

public void put(Name key, Object value) {
    dict.put(key,
        new Binder(value, head, dict.get(key)));
    head = key;
}

class Binder {
    Object value;
    Name next; // 连接了同层作用域的下个名字
    Binder parent; // 连接了上层作用域的同个名字
    Binder(Object v, Name n, Binder p) {
        value = v; next = n; parent = p;
    } // tail 记录了外层作用域的 value
}
```

```
class SymbolTable {
    private Name head;
    private Stack<Name> stack;
    public void beginScope() {
        stack.push(head);
        head = null;
    }
    public void endScope() {
        while (head != null) {
            Binder e = dict.get(head);
            if (e.parent != null) // 如果外层作用域还存在 head
                dict.put(head, e.parent);
            else
                dict.remove(head); // 若否，从符号表中直接删除
            head = e.next;
        }
        head = stack.top(); stack.pop();
    }
}
```

# 符号表是唯一的么？

- 变量，函数和类型可以分不同的命名空间
  - Pascal 的结构名和变量名可以重复
  - 如果区分命名空间，应该有多个符号表
    - tenv, venv
    - 类型环境，变量环境
- M语言变量，类型，函数都共用一个命名空间，不支持
  - `symbol symbol = symbol();`
- 规范的写法应该是
  - `Symbol symbol = getSymbol();`

# 可持久化红黑树

- 参见虎书 ch5.1
- “函数化”编程风味更重一点
- 不能加分！
- 不能加分！！
- 不能加分！！！

# 符号表的接口设计—— LIP 版本

```
public interface Scope {
    void define(Symbol symbol);
    Symbol resolve(String name);
    Scope getEnclosingScope();
    // 返回外层符号表
}

Symbol resolve(String name) {
    Symbol s = member.get(name);
    if (s != null) return s;
    return enclosingScope?.resolve(name);
}

currentScope = new LocalScope(currentScope);    // = table.beginScope();
currentScope = currentScope.getEnclosingScope(); // = table.endScope();
Symbol s = new << 名字, 类型 >>
currentScope.define(s);                          // = table.put( 名字, 类型 );
Symbol symbol = currentScope.resolve(name);        // = 类型 = table.get( 名字 );
```



# Symbol 类的设计

- 普通设计

```
Public Symbol {  
    String name;  
    Type type;  
}
```

- 文艺设计

- 将 Type 和 Scope 设计成接口
- 让我们来烧脑欣赏这个超变态复杂 OO 设计的实例

	名字	类型	作用域
int	Y	Y	N
C	Y	Y	Y
member	Y	N	N
method	Y	N	Y
内层 {}	N	N	Y

```
class C {  
  int member  
  int method( ) { }  
}  
int main() { { } }
```

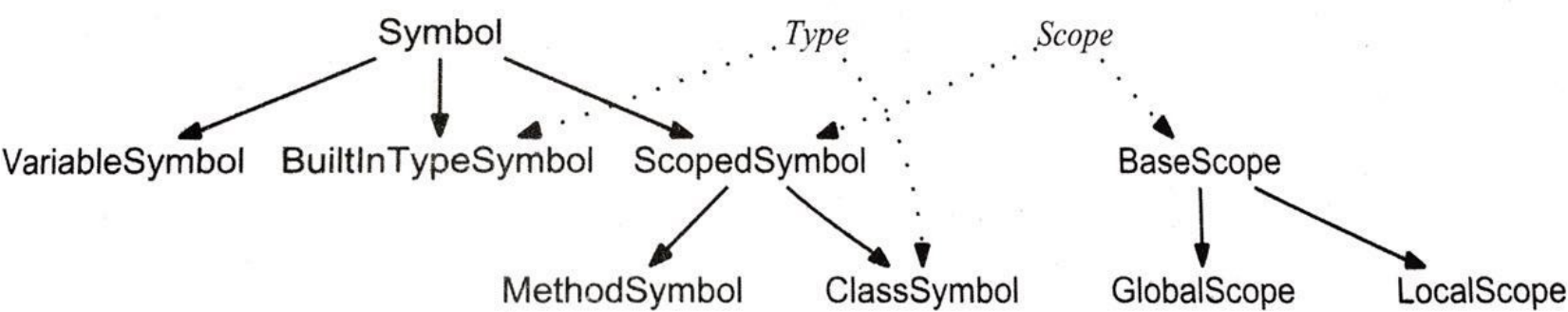


图 7.4 符号表管理相关的类继承图

# LIP 关于符号表的阅读指南

- 模式 16：没有函数，没有类，只有 int 和 bool
- 模式 17：有函数，没有类
- 模式 18：有函数，有结构，有成员函数
- 模式 19：支持继承

# 处理 Struct 的声明

```
public class StructSymbol extends Symbol implements Scope {  
    Map<String, Symbol> fields = new HashMap<String, Symbol>();  
    public Symbol resolveMember(String name) {  
        fields.get(name);  
    }  
}  
  
// 处理 class/struct 声明时  
Symbol ss = new StructSymbol(text, currentScope);  
currentScop.define(ss);  
currentScope = ss;  
  
// 离开 class/struct 声明时  
currentScope = currentScope.getEnclosingScope();
```

# Type checking 时如何处理 ID.expr

```
// a = va.b;
```

```
// todo: 检查 a 的类型和 v.b 的类型是否一致
```

```
// 先得到 v.b 的符号
```

```
StructSymbol scope = currentScope.resolve("va");
```

```
Symbol right = scope.resolveMember("b");
```

```
Symbol left = scope.resolve("a");
```

```
检查 left 和 right 的类型是否一致
```

# 总结

- 关于符号表，简单的写法见虎书
  - 这是一个面向过程的符号表，简单直接高效
- 优雅的，真正面向对象的符号表见 LIP
  - 比较烧脑子

# 语义分析

# 提纲

- Type checking 需要在抽象语法上走几遍
- 这是因为我们支持类和函数的“前向引用”
- 第一遍 + 第二遍
  - 将 class 和 function 先放到符号表里去
  - 但很多东西还没办法确定，比如一个函数的返回值是一种类，这种类的声明在函数声明的后面
- 第三遍
  - 解析所有类型：变量类型，返回值类型。一旦确定了变量的类型，就立即更新相关符号表对象。



MIPS

# 寄存器

- 通用寄存器
  - \$t0-\$t9
  - \$s0-\$s7
  - \$a0-\$a3
  - \$v0, \$v1
- \$fp
- \$sp
- \$ra

# 指令集的体系结构 PLP P214

- 数据移动
  - load store move
- 计算
  - add sub and or shift
- 控制转移
  - jump branch call(=jump and link)
- 特殊
  - trap——用不到

# 推荐开发编译器后端的顺序

- 第一步
  - 顺序
  - 条件
  - 循环
- 第二步
  - 函数调用
- 第三步
  - 数组
  - 字符串
  - 结构

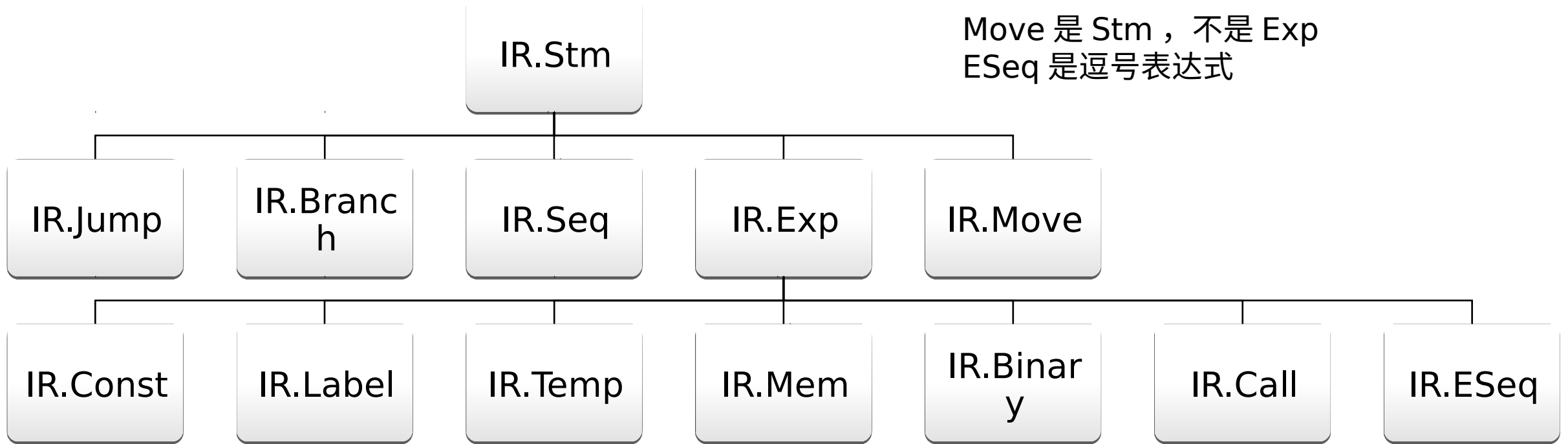
中间表示

# 四种中间表示的区别

- 汇编指令—— MIPS
  - 寄存器数量有限，立即数比较小
- 三地址码—— EaC 的 ILOC
  - 寄存器数量无限，立即数大小合理
- 树形 IR—— 虎书的 IR，或 EaC 中的低级 A
  - 不需要临时寄存器，分叉灵活
- AST
  - 作用域



**IR 有四种写法**



# IR 的内存模型

- 寄存器无限多
- 内存无限大
- 内存地址
  - 通过 label 获得绝对地址（只用在字符串上）
  - 通过 fp+const 获得相对地址



# 翻译 AST.WhileExp

```
Ast.Exp test; // 测试表达式
Ast.Stm body; // 循环体
Label begin = new Label(); // 测试通过，开始循环体的位置
Label end = new Label();    // 测试不通过，循环体结束的位置
public IR.Stm translate() {
    return new IR.Seq(new IR.Branch(test.transalte(), begin, end),
        new IR.Seq(begin,
            new IR.Seq(body.translate(),
                new IR.Seq(new IR.Jump(begin), end))));
}
```

# 翻译 AST.VarDec

```
Ast.ID id;  
Ast.Exp init;  
public IR.Stm translate() {  
    return new IR.Move(  
        getAccessExp(id),    // 需要一个机制获得变量 id 所在的位置  
        init.translate()  
    );  
}
```

# 变量存在哪里？——内存模型 EaC 5.4.3

- Memory-to-Memory Model
  - 适用 CISC （可用寄存器只有两三个）
  - 所有变量在内存里都有一个对应位置，容易找到变量在哪里
  - 取值和写入都需要同步到内存
- Register-to-Register Model
  - 适用 RISC （因为寄存器多）
  - 变量能存在寄存器里，就不要放到内存里
  - 缺点： 32 个寄存器不可能分配给 N 个变量，需要活性分析 + 寄存器分配
  - 也有一种做法是，遇到寄存器不够用的时候就拒绝编译……

# 好架构可以做到在这两种模型上切换

```
class InMeM extends Access {  
    int offset;  
    IR.Exp access(IR.Exp fp) {  
        return new IR.Mem(  
            new(new IR.Binary(  
                +, fp,  
                new IR.Const(offset))));  
    }  
}
```

```
class InReg extends Access {  
    IR.Temp reg = new IR.Temp();  
    IR.Exp access(IR.Exp fp) {  
        return reg;  
    }  
}
```

# 基本块构造

# Rewrite Tree

- 将 IR 重写成 canonical tree
- 构造成基本块
- 然后找到 traces ，按照 IR.Branch （虎书 CJump ）的要求调整基本块的顺序

# Canonical tree

- 消去 Seq 和 ESeq 结点
- 对每个 Call 函数的 IR 结点，父亲必须形如 `Move(temp, Call())...`
  - 如果是过程，无所谓
- 这是因为 Seq，Eseq 和 Call 都难以由指令选择翻译成线性代码

# Basic block

- Basic block is a straight-line code sequence with no branches in except to the entry and no branches out except at the exit.
- 形式上看
  - 第一句是 Label （方便跳进来）
  - 最后一句是 Jump 或者 Branch
- 基本块之间怎么排列都可以
- 我们需要让每个 Branch 的 false label 跟在 Branch 的下一句
- 直接构造



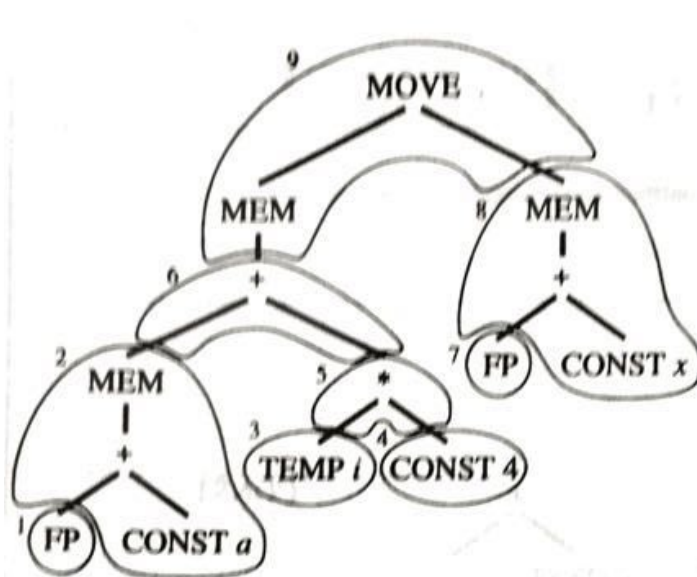
# 指令选择

苟利国家生死以，岂因祸福避趋之

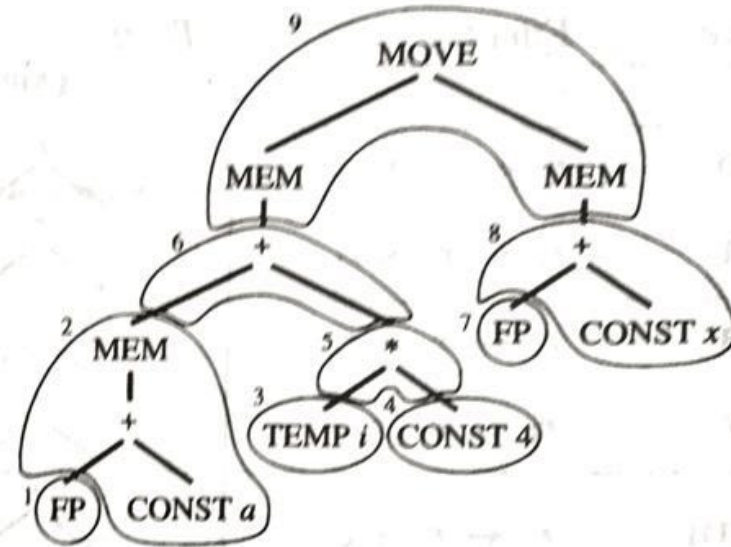
Name	Effect	Trees
—	$r_i$	TEMP
ADD	$r_i \leftarrow r_j + r_k$	$\begin{array}{c} + \\ \swarrow \quad \searrow \end{array}$
MUL	$r_i \leftarrow r_j \times r_k$	$\begin{array}{c} * \\ \swarrow \quad \searrow \end{array}$
SUB	$r_i \leftarrow r_j - r_k$	$\begin{array}{c} - \\ \swarrow \quad \searrow \end{array}$
DIV	$r_i \leftarrow r_j / r_k$	$\begin{array}{c} / \\ \swarrow \quad \searrow \end{array}$
ADDI	$r_i \leftarrow r_j + c$	$\begin{array}{cc} + & + \\ \swarrow \quad \searrow & \swarrow \quad \searrow \\ \text{CONST} & \text{CONST} \end{array}$
SUBI	$r_i \leftarrow r_j - c$	$\begin{array}{c} - \\ \swarrow \quad \searrow \\ \text{CONST} \end{array}$
LOAD	$r_i \leftarrow M[r_j + c]$	$\begin{array}{ccc} \text{MEM} & \text{MEM} & \text{MEM} \quad \text{MEM} \\   &   &   \quad   \\ + & + & \text{CONST} \quad \text{CONST} \\ \swarrow \quad \searrow & \swarrow \quad \searrow & \\ \text{CONST} & \text{CONST} & \end{array}$
STORE	$M[r_j + c] \leftarrow r_i$	$\begin{array}{ccc} \text{MOVE} & \text{MOVE} & \text{MOVE} \quad \text{MOVE} \\ \swarrow \quad \searrow & \swarrow \quad \searrow & \swarrow \quad \searrow \quad \swarrow \quad \searrow \\ \text{MEM} & \text{MEM} & \text{MEM} \quad \text{CONST} \quad \text{MEM} \\   &   &   \\ + & + & \\ \swarrow \quad \searrow & \swarrow \quad \searrow & \\ \text{CONST} & \text{CONST} & \end{array}$
MOVEM	$M[r_j] \leftarrow M[r_i]$	$\begin{array}{c} \text{MOVE} \\ \swarrow \quad \searrow \\ \text{MEM} \quad \text{MEM} \\   \quad   \end{array}$

**FIGURE 9.1.** Arithmetic and memory instructions. The notation  $M[x]$  denotes the memory word at address  $x$ .

# Tiling Tree Problem



2    LOAD     $r_1 \leftarrow M[\mathbf{fp} + a]$   
 4    ADDI     $r_2 \leftarrow r_0 + 4$   
 5    MUL     $r_2 \leftarrow r_1 \times r_2$   
 6    ADD     $r_1 \leftarrow r_1 + r_2$   
 8    LOAD     $r_2 \leftarrow M[\mathbf{fp} + x]$   
 9    STORE     $M[r_1 + 0] \leftarrow r_2$   
 (a)

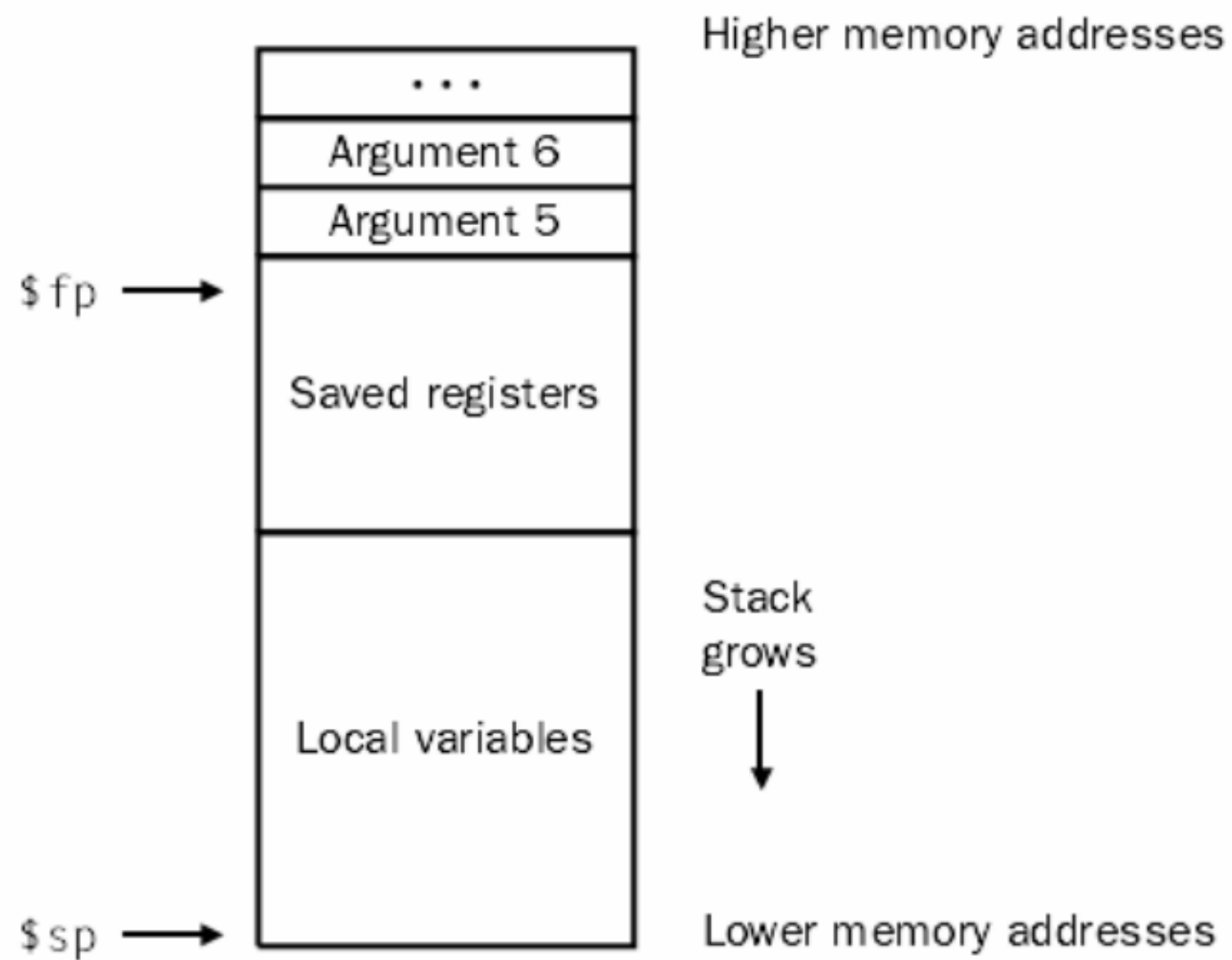


2    LOAD     $r_1 \leftarrow M[\mathbf{fp} + a]$   
 4    ADDI     $r_2 \leftarrow r_0 + 4$   
 5    MUL     $r_2 \leftarrow r_1 \times r_2$   
 6    ADD     $r_1 \leftarrow r_1 + r_2$   
 8    ADDI     $r_2 \leftarrow \mathbf{fp} + x$   
 9    MOVEM     $M[r_1] \leftarrow M[r_2]$

# 第二步

- 第一步只考虑顺序，循环和条件
- 第二步考虑函数调用
  - 先调试考虑简单的调用
  - 然后调试递归
- 第三步考虑数组字符串结构（运行时结构）

# 活动记录



# Caller vs Callee

- Caller 调用前
  - 保存 Caller-Saved 寄存器—— t0-t9, a0-a9
  - 传参数
  - 执行 jal
- 进入 Callee 前
  - 保存 callee-saved 寄存器—— s0-s7 , fp 和 ra
  - 调整 fp
- 离开 Callee 前
  - 恢复 callee-saved 寄存器,
  - 恢复 fp
- Caller 调用后
  - 恢复 Caller-Saved 寄存器

区分 Callee 和 Caller 有意义么？



```
abstract class Frame implements TempMap {  
    Label name;  
    Access allocLocal(bool inMem);  
    InstrList genCode(IR.Stm s);  
    IR.Stm procEntryExit1(IR.Stm s);  
    InstrList procEntryExit2(InstrList s);  
    InstrList procEntryExit3(InstrList s);  
}
```

# 申明变量的方法

- `Access allocLocal(bool inMen);`
  - 新开一个局部变量
  - `escape` 表示逃逸变量
    - M 语言里没有
    - C++ 里有引用
    - Pascal 里有函数层叠
  - 逃逸的变量必须放在内存里，不逃逸的先放在抽象寄存器里，如果寄存器分配溢出，自然会放到内存里

# 栈上分配变量

```
Access allocLocal(Boolean inMem, int size) {  
    if (inMem) {  
        Access ret = new InMem(this, offset);  
        offset += size; // bool 和 int 占用的字节不同  
        return ret;  
    } else {  
        return new InReg();  
    }  
}
```

# Frame 的构造

- `newFrame(Lable name, IntList size)`
  - 参数超过四个需要放在栈里

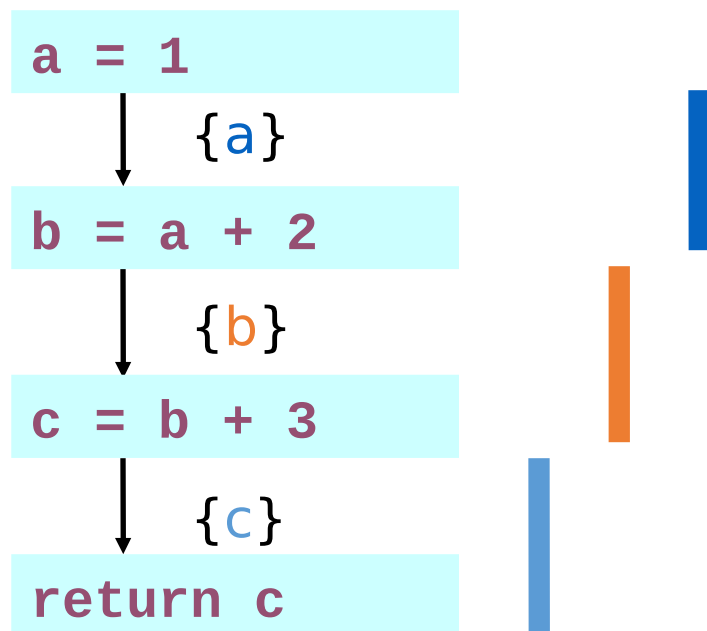
三个 procEntryExit

# 一些基本常识

- 每个函数是一个独立王国
  - 永远考虑其他函数的最坏情况
- 内存无限
  - 基地址为 `$fp`
  - 通过 `Mem[$fp+offset]` 访问内存

# 活性分析

# 动机

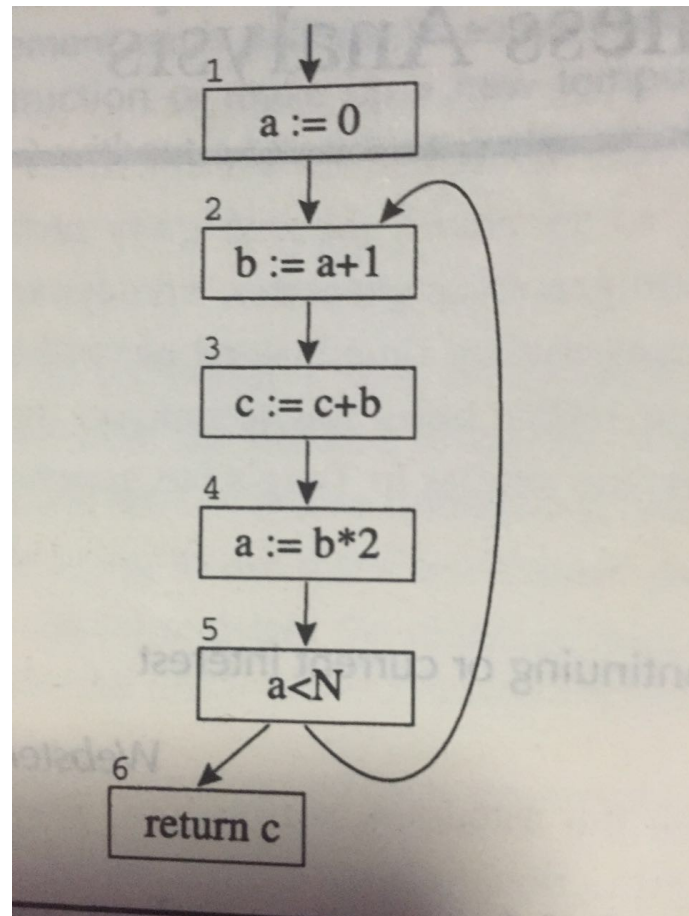


- 变量的颜色表示变量的活跃区间
- a,b,c 的活跃区间不重叠
- 可以分配在同一个寄存器里



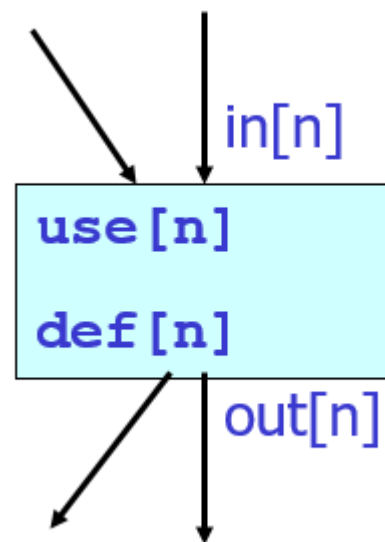
# CFG --- Control Flow Graph

- 每个点是一条三地址码
  - Use: 这个点所使用的变量
  - Def: 这个点所定义的变量
  - $\text{Use}[3] = \{c, b\}$
  - $\text{Def}[4] = \{a\}$
- 变量  $x$  live on an edge
  - 从这条边出发存在一条路径
    - 通向一个“use 中有  $x$ ”的点
    - 而且路上没有“def 中有  $x$ ”的点



# Live-In vs Live-Out

- $x$  —— 一个变量
- $n$  —— 结点 (一行汇编)
- $x$  live-in at  $N$  iff
  - $x$  live at  $e$ ,  $e$  属于  $\text{in-edge}[n]$
- $x$  live-out at  $N$  iff
  - $x$  live at  $e$ ,  $e$  属于  $\text{out-edge}[n]$



# Inside Basic Blocks (从后向前推)

$$\text{In}[n] = \text{Use}[n] \vee (\text{Out}[n] - \text{Def}[n])$$

进 N 前活跃的 = N 需要用的 + 出 N 时活跃的且没在 N 里被定义过

// Example:

a = 1

↓ int

b = a + 2

↓ out

c = b + 3

return c



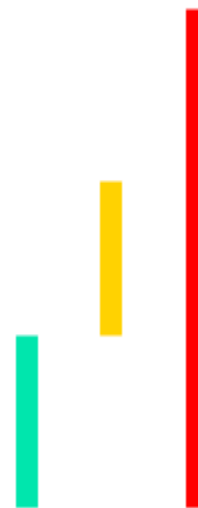
// Example:

a = 1

b = a + 2

c = b + 3

return a + c



# Data Flow Equations for general CFG

$$\mathbf{In}[n] = \mathbf{use}[n] \setminus (\mathbf{Out}[n] - \mathbf{Def}[n])$$

$$\mathbf{Out}[n] = \bigcap_{s \in \text{succ}[n]} \mathbf{In}[s]$$

- 计算 Out 需要向后看后继结点的 In
- 朴素算法： FixPoint Algorithm
  - 一开始 In 和 Out 都是空集
  - 不断循环直到没有变化
- 正确性： 能证明有解且有唯一解
- 复杂度：  $O(N^4)$  ， 存在更好的算法

# 干涉图

- 变量  $a$  和  $b$  不能分配在一个寄存器里 iff  $a$  和  $b$  之间有一条边
- 构建方法：
  - 非 move 指令  $n$  :
    - $\text{Def}[n]$  和  $\text{Out}[n]$  连边
  - move 指令：
    - 设指令为  $a=b$
    - $\text{Def}[n]$  (也就是  $a$ ) 和  $\text{Out}[n]-\text{Use}[n]$  (也就是  $b$ ) 连边

# 分块完成活性分析（稍微高级一点）

- Step 1: calculate def and use for each basic block b
  - one pass backward calculation
- Step 2: do liveness analysis on each block
  - 将每个块看做一个点，用 fixpoint algorithm
- Step 3: （建图） calculate liveness information for each statement in each block
  - one pass backward calculation

	out/in	out/in	out/in	out/in
3	{}	{c}	{c}	{c}
2	{}	{c}	{a,c}	{a,c}
1	{}	{a,c}	{a,c}	{a,c}

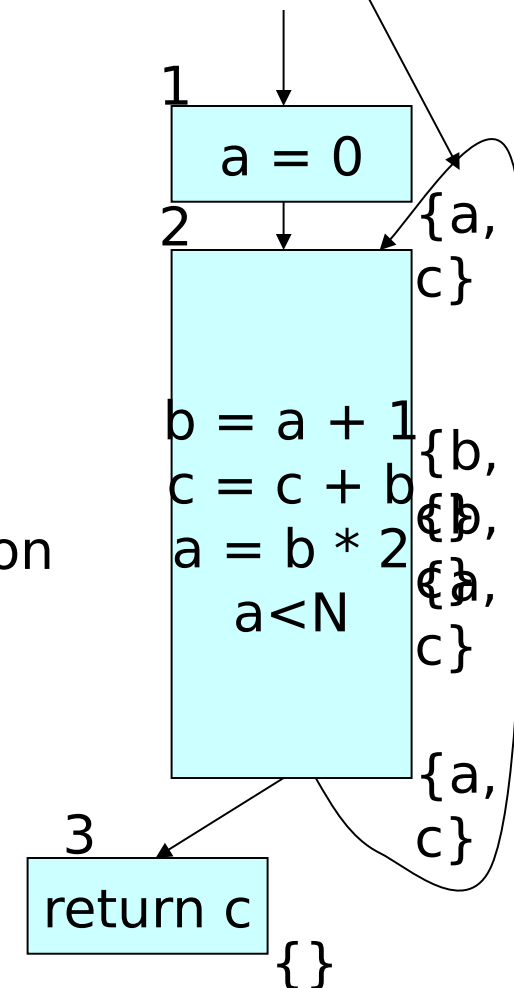
Blocks are reverse  
topo-sort ordered

Backward calculation  
of live\_out for each  
statement.

block	1	2	3
def	{a}	{a,b,c}	{c}
use	{}	{a,c}	{}

This set does  
NOT contain  
variable "b".  
Why?

live\_out for  
each block



# 寄存器分配



# 参考资料

- <http://staff.ustc.edu.cn/~bjhua/courses/compiler/2014/>
- 不错的网站，有 ppt 也有实验项目

# 寄存器分配

- 图染色分配比较繁琐
- 有更简单的分配方法，叫做 Linear Scan
  - 搜索一下论文有很多

# 学习编译器设计的收获

- 有的时候让程序来写程序比人来写程序更快更方便
- OO Programming needs design
- 一些算法问题有其理论背景