CH2 Lexical Analysis

2.1 scanning

1.某些记号只有一个词义:保留字;某些记号有无限多个语义:标识符 ID 表示.

2.2 RE

1.一些 notation

* > • > |

M?	Optional, zero or one occurrence of *M*.
[a – zA – Z]	Character set alternation.
.	A period stands for any single character except newline.
"a.+*"	Quotation, a string in quotes stands for itself literally.

2.RE 匹配优先匹配保留字;最长字串优先

"if8" matches as an identifier by the longest-match rule

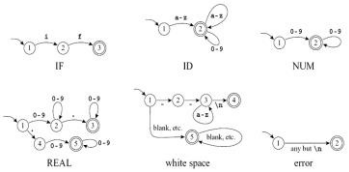
● 虽然 "if" 也能匹配 IF, but longest match "if" matches as a reserved word by rule priority

● 虽然 IF 和 ID 都满足 longest match ,但是 IF 优先级比 ID 高

2.3 有穷自动机

1.错误状态默认不画,但是存在;错误状态下的任何转移均回到自身,永远无法进入接受.

2.一些给出的基本 FA 图:



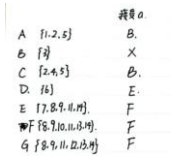
2.4 正则表达式到 DFA

1.由 RE 画出 NFA

Strings over the alphabet {a, b, c} where the first a precedes the first b.

c*a(a|c)*b(a|b|c)*

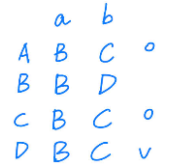
2.NFA->DFA:首先列出所有接受不同字符输入的节点,①初始化表格:列为 closure, input1, …, inputk。②计算初始状态的 ε 闭包;根据不同的输入字符计算出转移到的闭包;③新转移到闭包占据新的一行,重复第②步,直到不产生新的闭包为止。④包含原接受状态的所有闭包都是接受状态. PS:ε闭包首先包含自身.



4.DFA 状态数最小化:

最小状态数的 DFA 唯一

首先将终结状态分出来,接着列出如下每个状态在每个输入下的转移状态,相同归为一组,如下面例子



CH3 上下文无关文法分析

3.1 CFG

1.左递归:定义 A 的推导式的右边第一个出现的是 A;右递归:定义 A 的推导式右边最后一个出现的是 A;

3.2 parse tree & AST

1. 分析树 (concrete syntax tree/parse tree) 内部节点是非终结符,叶节点是终结符

2. 同一个串存在多种推导方式,对应分析树的不同遍历方式。最左推导对应前序遍历(根 -> 左子树 -> 右子树),最右推导对应后序(左子树 -> 右子树 -> 根)

3.AST 去除了终结符和非终结符信息,仅保留了语义信息;一般用左孩子右兄弟(多叉->二叉)

3.3 Ambiguity 二义性

1.定义:带有两个不同的分析树的串的文法

Equivalently, there is more than one right-most or left-most derivation for some string.

2.解决方法①设置消歧规则 disambiguating rule②修改文法(introduce new non-terminals)

3.修改文法时需要同时保证优先级 precedence 和结合律 associativity

4.在语法树中,越接近根,越高,优先级越低;左递归导致左结合 Left-association,右递归会右结合

自顶向下分析: LL(k)

parses many, but not all CFGs

LL(1)

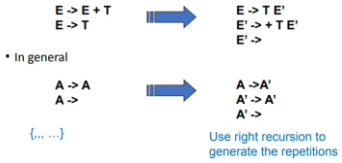
1.第一个 L 是从左到右处理,第二个 L 是最左推导,1 代表仅使用 1 个符号预测分析方向

2.动作:①生成**(generate)**,利用文法将栈顶的 N 替换成串,串反向进栈 ②**匹配(match)**:将栈顶的记号和下一个输入记号匹配 ③**接受(accept)**:接受字符串 ④**错误(error)**

3. LL(1)文法是无二义性的,对任意规则 $A\to\alpha1|\alpha2$, First(α1)∩First(α2)为空,否则不是 LL(1).

4. LL(1)面对重复和选择的解决方法:**消除左递归** left recursion removal.

和 **提取左因子** left factoring $A\to\alpha\beta|\alpha\gamma.A\to\alpha A',A'\to\beta|\gamma$



first follow sets:

1. First set 定义:

- **Base case**: if X is a terminal symbol
  - First(X) = {X}
- **Inductive case**: if X is non-terminal and X -> Y1 Y2 ... Yk
  - First(X) = First(X) ∪ First(Y1 Y2 ... Yk)
    = First(X) ∪ F1 ∪ F2 ∪ ... ∪ Fk, where:
    - F1 = First(Y1)
    - F2 = First(Y2) if Y1 ->* ε, otherwise ∅
    - ...
    - Fk = First(Yk) if Y1 Y2 ... Y_{k-1} ->* ε, otherwise ∅

2.Follow 定义:

- **Base case**: initially, we assume nothing follows X
  - Follow(X) = {}
- **Inductive case**:
  - For any strings α, β, if Y -> αXβ, then Follow(X) = Follow(X) ∪ First(β)
  - For any strings α, β, if Y -> αXβ and β->*ε, then Follow(X) = Follow(X) ∪ Follow(Y)

PS:③更常见的情况是 B→αA,那么 Follow(A) 包括 Follow(B)

3.Nullable set

$Nullable(X) = True\ if\ X->^{*}\epsilon$

Algorithm:

```
for each symbol X:
    Nullable(X) = False
repeat
    for each production X -> Y1 Y2 ... Yk:
        if Nullable(Yi) = True for all 1 <= i <= k:
            Nullable(X) = True
until Nullable did not change in this iteration
```

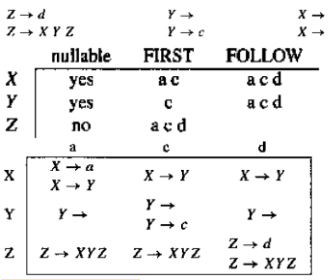
定理:A non-terminal A is nullable if and only if First(A) contains ε

期中错题 Given production A→Bαc, we have:Follow(A)⊂Follow(C),First(B)⊂First(A)

LL(1) 分析表构造算法

Build parsing table where row X, col T tells parser which clause to execute in function X(对应上面的实现) with next token T (T 是遇到的下一个 token)

- **if T ∈ First(γ) then enter (X → γ) in row X, col T**
- **if γ is Nullable and T ∈ Follow(X) enter (X → γ) in row X, col T**



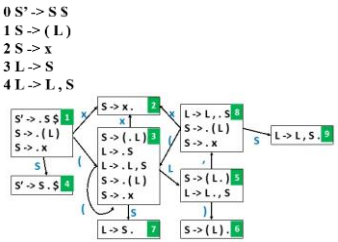
LL(1) 判别: LL(1) 分析表每个 box 里最多一条规则

自底向上分析: LR(k)

Yacc 基于 LALR(1).

对于 start symbol 而言,需要新增 S'→S$规则来引入.

LR(0)



由上面的状态图构造表如下:

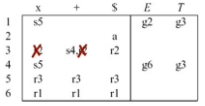
	(	)	x	,	$	S	L
1	s3		s2			g4	
2	r2	r2	r2	r2	r2		
3	s3		s2			g7	g5
4					a		
5		s6		s8			
6	r1	r1	r1	r1	r1		
7	r3	r3	r3	r3	r3		
8	s3		s2			g9	
9	r4	r4	r4	r4	r4		

1.符合 LR(0) 的文法不可能二义(对于每个输入字符串,LR(0) 分析器有且仅有一种分析路径(唯一的移入和规约序列),这意味着每个字符串只能对应唯一的解析树)

8.A grammar is LR(0) : 表格里每个只有一个动作

SLR:

1.比 LR(0) 改进:shift 规则不变; reduce 时要求输入必须在属于 Follow(A) 的终结符的项中,而不是全放



What are the valid next tokens for r2?
Idea: we can choose the reduce action only if the next input token t ∈ Follow(E)
Follow(E) = {$}
only T[3, $] can be r2!

2.SLR 不可能是二义性

3.自底向上右递归可能引起栈溢出,需要避免

4.SLR 中的两种冲突,sr 冲突使用消歧规则:优先移进; rr 冲突基本是设计出问题

LR(1)

1.LR(1) 的起始状态是 [S'→·S,$]的闭包

2.Closure(I) =

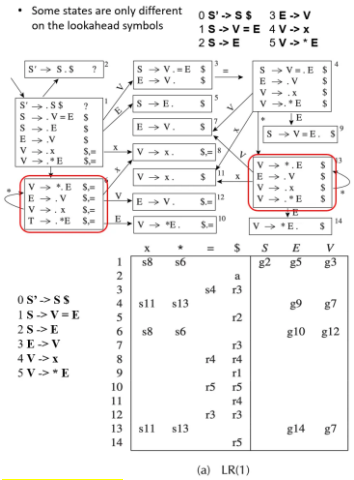
```
repeat
  for I 中任意项(A→ α.Xβ,z)
    for 任意产生式 X→γ
      for 任意 w∈First(βz)
        I:=I ∪ {(X→.γ, w)}
until I 没有变化
```

S' → .S $	?
S → .V = E	$
S → .E	$
E → .V	$
V → .x	$,=
V → .*E	$,=

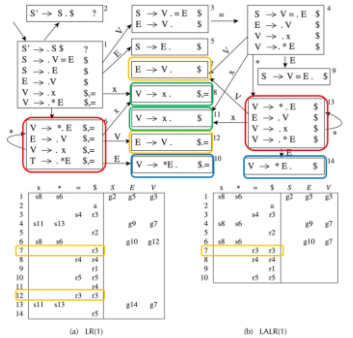
3.LR(1) 表的构造如下:在圆点到末尾的位置发生规约,对应项中填入 rn(n 是规则编号); 读入终结符则在对应项中写入 sn;读入非终结符则在对应项中写入 gn; 读入 S 后在标记符位置填 a 表示 accept

4.LR(1) 文法不可能二义性

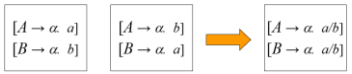
5. LR(1) 分析表示例



LALR(1) 将 look ahead symbol 进行合并:除了 look ahead symbol 不同以外全都相同的规则合并成一个

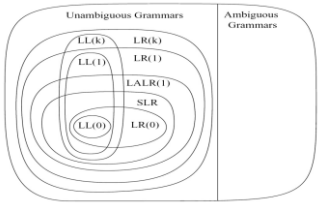


9.如果文法是 LR(1),那么 LALR(1)中必然没有 sr 冲突,但是可能有 rr 冲突.



10.如果文法是 SLR(1),那么必然是 LALR(1).

11. 各类文法的层次如下:



Error Recovery

LR(1) 比 LALR(1) 或 SLR(1)更早检测出错误;LALR(1)和 SLR(1)都比 LR(0)更早

There are three possible alternative actions:
① Pop a state from the stack.
②Successively pop tokens from the input until a token is seen for which we can restart the parse.
③Push a new state onto the stack.

When an error occurs is as follows:
①Pop states from the parsing stack until a state is found with nonempty Goto entries.
②If there is a legal action on the current input token from one of the Goto states, push that state onto the stack and restart the parse.
③If there is no legal action on the current input token from one of the Goto states, advance the input

---

LR(1) 就存在一些子集不能被 LL(k) 识别,存在 L∈LL(1)但是 L∈LALR(1) 的例子。

默认情况下,Yacc/Bison 在遇到移入-归约冲突时,采取的策略是 shift

CH5 语义分析

编译器完成的是 static semantic analysis

5.1 symbol table

1. also called environments mapping identifiers to their types and locations

```
1 function f (a:int, b:int, c:int) =
2   (print_int (a+c);
3    let var j := a+b
4        var a := "hello"
5    in print(a); print_int(j)
6   end;
7   print_int(b)
8   )
```

line 1: σ1 = σ 0 + {a ↦ int, b ↦ int, c ↦ int}

how does the compiler know the type of j?

line 3: σ2 = σ1 + {j ↦ int}

line 4: σ3 = σ2 + {a ↦ string}

Bindings in the right-hand table override those in the left(在命令式符号表更新中,{a→int}+{a→string} ={a→ int, a → string}。F)

line 4: σ3 = σ2 + {a ↦ string}

line 6: discard σ3, go back to σ1 (line 1)

line 7: look up b in σ1

line 8: discard σ1, go back to σ0

2.局部变量都有一个**作用域(scope)**, 变量仅在自己的作用域中可见.当语义分析到达每一个作用域的结尾时,所有属于该作用域的变量都被符号表抛弃不用.

5.2 命令式(imperative)

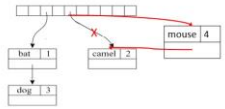
1.散列表(bucket list) 实现

2.特点:插入时将变量插入到对应 bucket 的链表头部

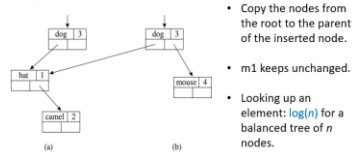
5.3 函数式(functional)

1.二叉搜索树(BST)实现

2.特点:不会直接操作原符号表,而是创建新的 BST 节点(和查找元素效率相同 logN),



已知 $m_1 = \{bat \to 1, camel \to 2, dog \to 3\}$,现在添加新绑定 mouse→4.

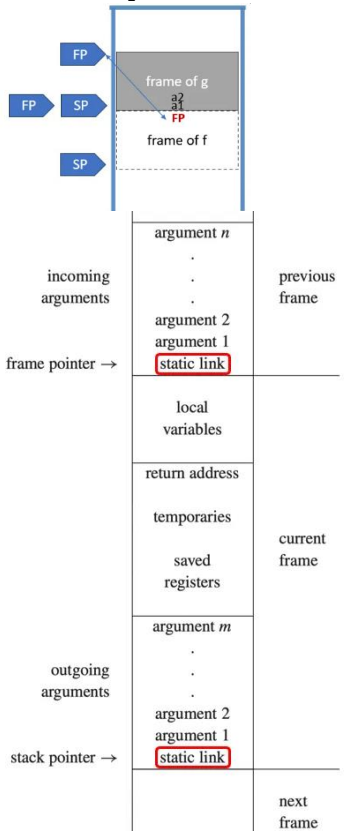


CH6 活动记录

6.1 stack frame

activation record/stack frame

2. **帧指针(frame pointer)**:指向当前帧的指针,一般是上一个 sp;有些栈帧会分配一个寄存器存 fp;fp=sp+size(frame),虚寄存器. 帧指针的变化:一个函数 g 调用 f 时①sp 指向 g 传给 f 的第一个参数②f 分配栈帧(sp-栈帧大小)③进入 f 时旧的 sp 变成当前帧指针;fp 旧值被保存到栈帧内,新的帧指针变成旧 sp④f 退出时把 fp 拷贝给 sp,取回原先保存的 fp 即可.



3.**参数传递**:现代计算机传参约定:前 K 个参数放在寄存器里传递,剩余在储存器传递.

寄存器传参的方法(4 种):

①不给叶子过程(**leaf procedure**)分配栈帧 ②过程间寄存器分配(**interprocedural register allocation**) ③直接重写寄存器不做保护 ④寄存器窗口技术(**register windows**)

4.**返回地址**:call 指令地址的下一条指令地址.

5.**Frame-resident variables**: 一般来说局部变量和中间结果会放到寄存器中,以下情况需要将变量储存到栈帧内(memory): ①变量传地址/引用(passed by reference) ②被嵌套在当前过程的函数调用(nested accessed) ③太大了放不下(too big to fit)④变量是数组⑤有特殊用途的变量(传参等) ⑥存在过多的临时变量和局部变量(溢出 spill)

6. **逃逸变量(escaping variable)**:①传地址变量 ②被取地址 ③被内层嵌套函数访问的变量.

7.**静态链(static link)**:本质是指向上一层嵌套层级的栈帧的指针.内层嵌套函数调用外层定义的变量需要用到静态链,否则无法寻址.

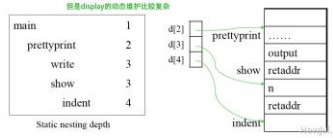
```
int g (int x){
    int f (int y) {
        …
    }
    return f(x)+1;
}
```

每当我调用 f 我都要把 g 栈帧的指针传给它,放在 f 的 static link 区域,对应左图中的第二个框(如果当前帧是 f),这个 static link 是 f 的声明之处(在哪个函数里声明)

而上面那个是 f 的调用之处(在哪个函数里调用)

9.**其他访问外层变量的方法:(实现 block structure)**

①display: 一个全局数组,位置 i 存放最近一次的,静态嵌套深度为 i 的过程的栈帧.是管理**静态链**的全局数组(管理的是静态链不是栈指针)



②lambda shifting:内层函数访问的外层声明变量,会作为函数参数传给内层嵌套函数.**注意:静态链层级是函数的嵌套深度,不是递归调用的深度,两者不同概念.**

CH7 中间代码(IR code)

1.**基本概念:**

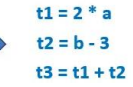
①前端(frontend):词法分析|语法分析|语义分析|翻译成中间代码

②后端(backend):IR 优化|翻译成机器语言.

2.**TAC 三地址码**

x = y op z

op 只能有一个, address 最多三个



7.1 中间表示树

1.**中间语法树的表达式:**

①CONST(i):整型常数 ②NAME(n):符号常数(汇编的 label) ③TEMP(t):临时变量 ④BINOP(o,e1,e2):对操作数 e1,e2 的二元操作 ⑤MEM(e):作为 MOVE 操作的左子式时表示对储存器 e 地址的存入;其他位置表示读取该地址的内容 ⑥CALL(f,l):过

程调,1 是参数列表 ⑦ESEQ(s,e): 先计算语句 s 形成副作用,然后计算 e 为该表达式的值

2.中间语法树的语句:

⑧MOVE(TEMP t, e): 计算 e 的值然后存到临时变量 t 中 ⑨MOVE(MEM(e1),e2)):计算 e2 的值然后存入到 e1 作为地址的内存中 ⑩Exp(e) 计算 e 抛弃值,只要副作用 (11) JUMP(e, labs):跳转到 e 或者 labs 为 label 的地址 (11)CJUMP(o, e1,e2,t,f):依次计算 e1 和 e2, 生成值 a,b;然后用比较运算符操作 aob,如果结果为 true 跳到 t,反之跳转到 f; (12)SEQ(s1,s2):语句 s1 后面跟 s2 (13)LABEL(n):n 的常数值为当前机器代码的地址.

7.2 翻译成中间树语言

对于 CJUMP 和 JUMP 语句,还不知道 label 的具体值,需要使用两张表:真值标号回填表(true patchlist)和假值标号回填表(false patch list).

①简单变量:存放在栈帧的变量 v 转化为 MEM(BINOP(PLUS, TEMP fp, CONST k)), k 是栈帧内 v 的地址偏移.**②追踪静态链**:MEM(+(CONST Kn, MEM(+CONST Kn-1, … MEM(+(CONST K1, TEMP fp))…))); k1~kn-1 是各个嵌套函数的静态链位移 **③数组变量下标:** 假设 b 是由 MEM(TEMP b) 表示的数组变量 ,b[i+1] 表示为 MEM(+(MEM(TEMP b), BINOP(MUL, +(TEMP i, CONST 1), CONST W)))

7.3 声明

函数被翻译为**入口处理代码(prologue)/函数体(body)**和**出口处理函数(epilogue)**组成的汇编语言代码. **①入口处理函数包含:**(1)声明一个函数开始的伪指令(2)函数名字的标号定义(3)调整栈指针的一条指令用于分配新的栈帧(4)将逃逸参数保存至栈帧的指令,以及将非逃逸参数传送的新临时寄存器指令(5)保存在此函数用到的 caller-save 寄存器 **②入口处理之后是:**(1)函数的函数体 **③出口函数位于函数体之后,包含:**(1)将返回值传送至专用与返回结果的寄存器(2)用于恢复 callee-save 的寄存器取数指令(3)恢复栈指针,释放栈帧(4)return 指令(5)声明函数结束的伪指令

CH8 基本块

1.为什么? ①CJUMP 能够跳转到两个标号的任意一个,但实际机器代码是条件为假时跳转到下一条②ESEQ 会使得子树的不同计算顺序产生不同结果③表达式使用 CALL 会有计算顺序不同的问题④CALL 的嵌套调用(作为另一个 CALL 的参数)会出问题,覆盖存放返回值的寄存器的值

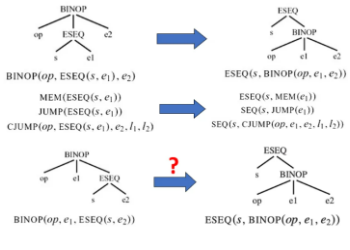
2.重写流程:①一棵树重写成规范树②将树分组合成不含转移和标号的基本块(basic block)集合 ③对基本块进行排序形成一组轨迹(trace);每一个 CJUMP 后就是其 false 标号

8.1 canonical tree

1.定义: 一棵不含 SEQ 和 ESEQ 的等价 IR 树,且每一个 CALL 的父亲节点不是 EXP(…)就是 MOVE(TEMP t, …).

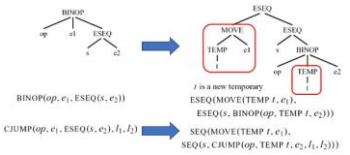
- Property 1:
  - Each canonical tree only contains one statement node, i.e., the root node. Other nodes are all expression nodes.
- Property 1 and property 2:
  - The parent of a CALL node must be the root node of a canonical tree and must be EXP(..) or MOVE(TEMP t, ..).
  - There can only be one CALL node in a canonical tree, because EXP(...) and MOVE(TEMP t, ...) can only contains one CALL.

2.ESEQ 转化:



Suppose:
s = MOVE(MEM(x), y); e1 =MEM(x)

此处 ESEQ 中 s 和 e1 不可交换要将 e1 存到 temp 中



如果 ESEQ 中 s 和 e1 是**可交换的(commute)**,那么可以直接把 s 和 e1 交换,ESEQ 提出来

- We conservatively approximate whether statements commute
  - commute(s, e) = True if s and e definitely commute
  - commute(s, e) = False otherwise
- This commute function estimates (very naively) whether a statement commutes with an expression:
  - A constant (CONST(i), NAME(n)) commutes with any statement
  - An empty statement (e.g., EXP(CONST(i))) commutes with any expression
  - Anything else is assumed *not* to commute

3.CALL 移到顶层: 以 BINOP(op,CALL(),CALL(),…)为例, 第二个 CALL 会在 BINOP 执行前覆盖第一个 CALL 返回在寄存器里的值.解决办法是使用 ESEQ 将返回值保存到一个新的临时变量

CALL(fun, args) ->
ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)

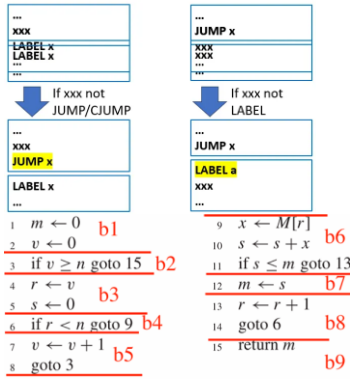
4.消除 SEQ: 将 SEQ 全部移到右边, 然后消除

SEQ(SEQ(a, b), c) = SEQ(a, seq(b, c))

8.2 处理条件分支

1.基本块(basic block): ①第一个语句是 LABEL②最后一个语句是 JUMP 或 CJUMP ③没有其他的 LABEL,JUMP 或 CJUMP

2.划分基本块方法:



3.轨迹(trace):程序执行期间可能连贯执行的语句序列.轨迹集合能够覆盖整个程序的,且每一个基本块仅出现在一条轨迹中.

方法:从根开始一路往下走,边走边标记走过的节点,一路走到又遇到已标记节点/没有子节点位置,路径上的节点就是一个 trace,然后从剩余节点中某一个开始重复,直到所有节点都被标记为止

上图的 Traces:
b1->b2->b3->b4->b5;
b6->b7->b8; b9

4.完善:①所有后面跟 false 标号的 CJUMP 不变②对任何后面跟 true 标号的 CJUMP,交换器 true 标号和 false 标号以及判断条件取反③对其后跟随的既不是 true 也不是 false 标号的 CJUMP,生成新的标号 f'并重写 CJUMP,使得其 false 标号紧跟其后.

CJUMP(cond, a, b, lt, lf)->
CJUMP(cond, a, b, lt, l'f)
LABEL l'f JUMP(NAME lf)

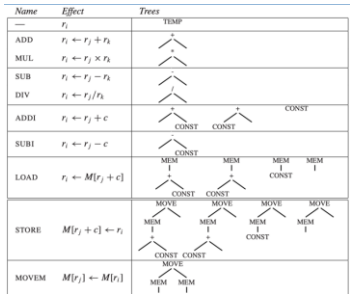
CH9 指令选择

9.1 树型到指令

1.optimum:瓦片的代价和是所有可能值中最小 global

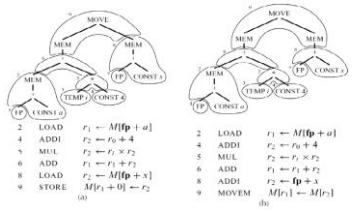
2.optimal:不存在两个相邻的瓦片能连接成一个代价更小的瓦片覆盖. local

It's easier to find optimal tilings.



$d \to \text{MEM}(+(a, \text{CONST}))$
$\to \text{MEM}(+(\text{CONST}, a))$
$d \to \text{MEM}(\text{CONST})$
$d \to \text{MEM}(a)$
$d \to a$
$a \to d$

3.一棵树可以有多种 tiling 的方式:



4.Optimum ⊆ Optimal.

Suppose every instruction costs one unit, except for MOVEM which costs m units. Then either Figure(a)is optimum (if m > 1) or Figure(b)is optimum (if m < 1) or both (if m = 1); but both trees are optimal.

9.2 指令选择算法

1.Maximal Munch 算法:①是 optimal, 不一定是 optimum②Maximal Munch 算法的 tiling 是自顶向下的,但是指令的生成是逆序的(因为上层的覆盖指令需要下层的指令提供操作数,所以逆序)

2.动态规划:①自底向上, 可以找到 optimum 覆盖,子问题是子树的覆盖 ②会给每个节点指定一个代价:可以覆盖以该节点为根的子树的最优指令序列的指令代价之和.

3.树文法(Tree Grammar): ①对于具有复杂指令集以及若干类寄存器和寻址模式的机器, 是动态规划的推广 ② 使用 brain-damaged Jouette 体系:有两类寄存器(a 寄存器存地址;d 寄存器存数据)③使用 CFG 来描述瓦片,文法具有高度歧义性,但是动态规划可以很好处理.

LOAD,MOVEA 和 MOVED:

$d \to \text{MEM}(+(a, \text{CONST}))$
$\to \text{MEM}(+(\text{CONST}, a))$
$d \to \text{MEM}(\text{CONST})$
$d \to \text{MEM}(a)$
$d \to a$
$a \to d$

4.快速匹配(fast match): 使用 switch-case 来匹配非叶子节点的 label.

5.算法效率:T 个瓦片,平均每个匹配的瓦片有 **K** 个非叶子节点.**K'** 是在给定子树中为确定匹配那个瓦片需要检查的最大节点个数(近似于最大瓦片的大小).假定平均每个树节点可以与 **T'** 个瓦片匹配.输入树的节点为 **N**.①Maximal Munch: $O((K'+T')*N/K)$ ②动态规划:$O((K'+T')*N)$ 动规的比例常数比 Maximal 大,因为要遍历两遍.K',K,T 是常数,线性复杂度.

9.3 CISC 机器

RISC machine
32 registers
only one class of integer/pointer register
arithmetic operations only between registers
"three-address" instructions of the form r1 ← r2 ⊕ r3
load and store instructions with only the M[reg+const] addressing mode
Every instruction exactly 32 bits long
one result or effect per instruction

CISC machine
few registers (16, or 8, or 6)
registers divided into different classes, with some operations available only on certain registers;
arithmetic operations can access registers or memory through "addressing modes";
"two-address" instructions of the form r1 ← r1 ⊕ r2;
several different addressing modes;
variable-length instructions, formed from variable-length opcode plus variable-length addressing modes;
instructions with side effects such as "autoincrement" addressing modes.

CH10 活跃分析

10.0 定义

1.如果一个变量的值在将来还需要使用,则变量是**活跃的(live)**

2.**控制流图(control flow graph)**:程序的每条语句都是流图的节点.

10.1 数据流方程的解

1.succ[n]是节点 n 的后继节点;pred[n]是节点 n 的前驱节点

2.定值(define):对变量和临时变量的赋值成为变量的定值;**使用(use)**:出现在赋值号右边的变量为其使用.

注意 if v>n 是 use v 和 n

3.活跃性:变量在边上活跃是指存在一条边通向该变量的一个 use 的有向路径,且不经过该变量的任何 def.如果变量在一个节点的所有入边上全是活跃的,那么该变量是**入口活跃的(live-in)**;若一个变量在一个节点的所有出边上都是活跃的,那么该变量在该节点是**出口活跃的(live-out)**.

10.2 活跃性计算

1.活跃性计算:就是计算流图每一个节点的 in 和 out 集合

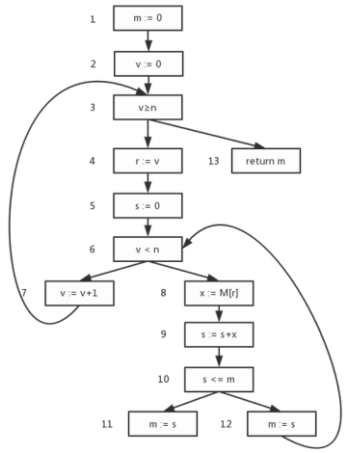
方程 10.2

$$in[n] = use[n] \cup (out[n] - \text{def}[n])$$

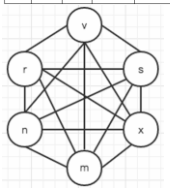
$$out[n] = \bigcup_{s \in succ[n]} in[s]$$

2.活跃性计算的迭代方法:

```
1  m ← 0                    b1
2  v ← 0
3  if v ≥ n goto 15         b2
4  r ← v                    b3
5  s ← 0
6  if r < n goto 9          b4
7  v ← v + 1                b5
8  goto 3
9  x ← M[r]                 b6
10 s ← s + x
11 if s ≤ m goto 13         b7
12 m ← s
13 r ← r + 1                b8
14 goto 6
15 return m                 b9
```



			1st		2nd		3rd	
	use	def	out	in	out	in	out	in
13	m			m		n		n
12	r	r		r	vrsnm	vrsnm	vrsnm	vrsnm
11	s	m	r	rs	vrsnm	vrsn	vrsnm	vrsn
10	ms		rs	rsm	vrsnm	vrsnm	vrsnm	vrsnm
9	sx	s	rsm	rsmx	vrsm	vrsnmx	vrsnm	vrsnmx
8	r	x	rsmx	rsm	vrsnmx	vrsnm	vrsnmx	vrsnm
7	v	v		v	vnm	vnm	vnm	vnm
6	vn		vrsm	vrsnm	vrsnm	vrsnm	vrsnm	vrsnm
5		s	vrsnm	vrnm	vrsnm	vrnm	vrsnm	vrnm
4	v	r	vrnm	vnm	vrnm	vnmx	vrnm	vnm
3	vn		vnm	vnm	vnm	vnmx	vnm	vnm
2		v	vnm	nm	bnm	nm	nnm	nm
1		m	nm	n	nm	n	nm	n



3.适当排序可以显著加快算法的收敛过程,**一般要从程序末尾往前算,先算 out 再算 in,可以显著提高速度和正确率**.信息活跃性是沿控制流箭头的反方向流动的,计算顺序同理.

4.时间复杂度:for 循环初始化节点 in,out 需要 O(N²);repeat 循环的时间复杂度是 O(N⁴).由于活跃信息大部分稀疏,实际运行时间在 O(N)和 O(N²)之间.

10.3 in 和 out 集合表示

1.位数组(bit array):程序中有 N 个变量,用 N 位数组表示集合①求并集/对位数组求按位或②时间效率:对每个字有 K 位的计算机,并运算需要 N/K 次操作

2.有序变量表:链表的成员是组成集合的元素①并集通过合并链表实现②时间开销和求并集的集合大小成正比.

3.方法比较:集合稀疏(平均少于 N/K)用有序链表表示速度会更快(越稀疏越快);集合密集:位数组表示更好.

10.4Least Fixed Points

1.定理: 方程 10.2 有一个以上的解(in,out 计算公式方程)

2.定理: 方程 10.2 所有解都包含最小解(least solution).

10.5 静态/动态活跃性

1.定理:不存在程序 H,它以任意程序 P 和输入 X 作为自己的输入.当 P(X)停止时返回真,当 P(X)无限循环时返回假. **推论**:不存在程序 H'(X,L),对任何程序 X 和 X 中标号 L,可以判断出 X 在执行中是否曾经到达了标号 L.

2.动态活跃: 程序在实际运行时,变量在执行过程中是否被使用。依赖于程序的具体运行情况和输入数据

3.静态活跃: 在编译时通过程序的源代码分析,确定变量在某个程序点是否可能在将来被使用。通常不依赖于运行时输入

10.6 冲突图

1.冲突(interference):阻止将两个同时活跃的临时变量分配到同一个寄存器.

2.冲突原因:①临时变量在程序的同一点同时活跃②某些寄存器必须被使用时,临时变量不能占用这些寄存器.

3.绘制冲突图的办法: (1) 对 def 变量 a 的节点标号为 n 的非 MOVE 指令,任意 $b_i \in out[n]$,添加冲突边 (a,b_i) **(2)**对于节点标号为 n 的 MOVE 指令 a←c,对 $b_i \in out[n]$ 且 $b_i \neq c$,添加边 (a,b_i). **注**:可以给(a,c)画上虚线,便于寄存器分配的 coalesce.

CH11 寄存器分配

11.1 通过简化进行着色

1.基于上一章的冲突图,寄存器分配问题转化为图着色问题:"颜色"就是寄存器,相邻节点不能着同一种颜色.这会导致部分变量必须保存到 memory 里,称为 spill.

2.简单图着色算法:①**构造(build)**:构造冲突图.②**简化(simplify)**:启发式图着色,如果一个图 G 的节点 n 的度小于颜色 K,那么去掉该节点后的图 G'如果能被 K 着色,G 也可以.方法:使用一个显式栈,将度小于 K 的节点压入栈中并从原图中删除,直到图不能化简为止.③**溢出(spill)**:简化过程中如果只有**高度数(significant degree)**点(度≥K),此时简化失效;需要按一定标准选择高度数点,将其**潜在溢出(potential spill)**,从图中删除并压入栈内打上潜在溢出标记.然后继续进行简化过程 ④**选择(select)**:将颜色指派给图节点,从空图开始重复地 pop 栈中图节点,重建图.当 pop 潜在溢出节点时,可能会发生无法着色的情况,这时发生**实际溢出(actual spill)**,这时不指派颜色而是继续执行选择阶段继续识别其他的实际溢出.但是潜在溢出邻居节点颜色数可能<K,可以着色不会成为实际溢出,成为乐观着色(optimistic color) ⑤ **重开(start over)**:如果不能为某些节点着色,那么需要重写程序,导致活跃性分析变化,重复上述所有流程,直到没有实际溢出.

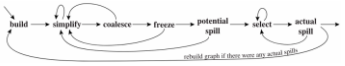
11.2 合并(coalesce)

1.合并的基础:冲突图可以删除冗余的 MOVE 指令,如果冲突图中如果传送指令的源操作数和目的操作数对应节点间不存在边,那么可以合并这两个节点.新节点的边是被合并的节点的边的并集.

2.安全的合并策略:①**Briggs 策略**:如果 a,b 合并后的节点 a&b 的高度数邻节点个数少于 K,则 a 和 b 可以合并.②**George 策略**:节点 a 和 b 可以合并的条件是:对于 a 的**每一个邻居 t**,满足两个条件之一即可合并 a 和 b:(1)t 与 b 已有冲突(2)t 是低度数节点.

这两种策略都是**保守的(conservative)**,是因为合并不会改变图的着色性,合并不成功时仍然是安全的.可能有多余的 MOVE 指令,但是不会出现溢出.

3.带合并的图着色算法图示



①**构造**:构造冲突图,将节点分类为 move-related 和 non-move-related.

move-related 是传送指令的源操作数或目的操作数,可以在一对传送有关节点之间画一条虚线 ②**简化**:逐个从图中删除低度数的 non-move-related 节点压栈. ③**合并**:对简化的成果按照上述合并策略进行保守合并,删除关联它们的 MOVE 指令虚线。合并产生的结点不再是 move-related。重复简化和合并的过程,直到剩下的节点全部是高度数节点或 move-related 节点. ④**冻结(freeze)**:简化和合并都不能进行时,寻找一个低度数的 move-related 节点,冻结这个节点所关联的那些传送指令(放弃对这些 move-related 节点的合并)把虚线删除,创造更多的简化机会.然后重新开始简化和合并. ⑤**溢出**:如果没有低度数节点,选择潜在可能溢出的高度数节点并压入栈中 ⑥**选择**:弹出整个栈并指派颜色.

4. 受抑制的(constrained):冲突图中传送相关的两个节点之间有冲突边.既有虚线相连,又有实线相连.

11.3 预着色(precolor)

1.定义:一些变量直接使用了真实寄存器,相当于已经固定了寄存器的使用,节点已经有了"颜色".所以叫"预着色".注意:预着色的节点一定**相互冲突**, 在冲突图上表现为**两两互联**.

2. 预着色结点中那些没有被显式使用过的(例如在参数传

递约定中）结点将不会与任何一个普通结点（非预着色的）相冲突。但是，一个显式使用了的机器寄存器将会有一个活跃范围，因此会与任何在此范围内也同时活跃的其他变量相冲突

3.预着色节点特性：①无法简化②无法指派颜色③无法溢出(认为寄存器节点的度是无限大).④可以参与合并.

Suppose r7 is a callee-save register:

4. 当这个程序存在较大的寄存器压力（对寄存器有较大的需求）时，t231 将会倾向于溢出，否则的话，寄存器分配器会倾向于与 r7 合并，并且可以删除与 t231 和 r7 关联的那两条 MOVE 指令

5.caller-saved & callee-saved regs

```
foo (){
    t = ...
    ... = ... t ...
    s = ...
    f()
    g()
    ... = ... s ...
}
```

A local variable or compiler temporary that is not live across any procedure call should usually be allocated to a **caller-save register**

Any variable that is live across several procedure calls should be kept in a **callee-save-register**

5.溢出优先级计算公式： 要选择那些度数高但是很少使用的结点。对于节点 a 而言，变量 a 在循环外层的 use 和 def 的总数记为$Out_{use+def}$，循环内层的 use,def 总数记为$In_{use+def}$；节点的度为D.节点 a 的溢出优先级=

$$\frac{(Out_{use+def}+10\times In_{use+def})}{D}$$

注意：D不包含虚线的计数，只包含实线的计数；Priority的值越小，优先级越高

6.对实际溢出节点的处理：假设变量 a 发生了实际溢出，那么 a 必须保存到 memory 中

- Before each use -> fetch
- After each def -> store

（可以看例题里的例子）

预着色例子：

r1 and r2 are caller-save, r3 is callee-save. r3 是 callee save 的，所以第一条插入 MOVE 指令把 r3 保存到一个临时变量 c 中，结束的时候再把 r3 从 c 中读出来。

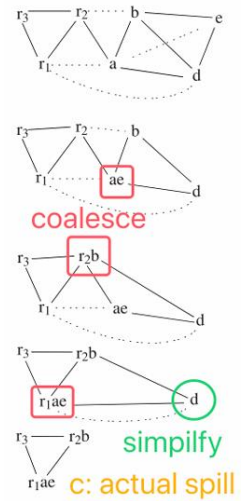
```
int f(int a, int b) {
    int d=0;
    int e=a;
    do {d = d+b;
        e = e-1;
    } while (e>0);
    return d;
}
```

Node	Uses+Defs outside loop	Uses+Defs within loop	Degree	Spill priority
a	(2 + 10 × 0) /	4 =		0.50
b	(1 + 10 × 1) /	4 =		2.75
c	(2 + 10 × 0) /	6 =		0.33
d	(2 + 10 × 2) /	4 =		5.50
e	(1 + 10 × 3) /	3 =		10.33

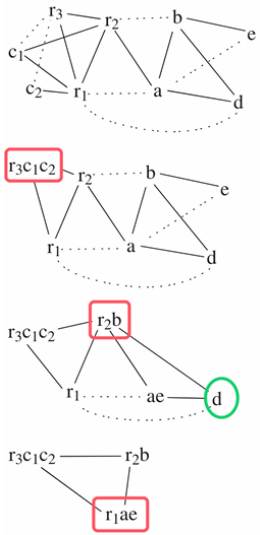
```
enter:  c ← r3
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop:   d ← d + b
        e ← e − 1
        if e > 0 goto loop
        r1 ← d
        r3 ← c
        return                (r1, r3 live out)
```

(b)

```
enter:  c1 ← r3
        M[cloc] ← c1
        a ← r1
        b ← r2
        d ← 0
        e ← a
loop:   d ← d + b
        e ← e − 1
        if e > 0 goto loop
        r1 ← d
        c2 ← M[cloc]
        r3 ← c2
        return
```



重写程序后：



最后分配：a-r1,b-r2,c-r3,d-r3,e-r1

r1 and r2 are (caller-save) argument registers and r3 is a callee-save

1.Freeze没有可以简化的节点，采用Briggs，也没有可以合并的节点。可以冻结节点 u 和 r1 之间的 MOVE**2.** Simplify u**3.** Freeze 此时又没有可以简化的节点了，采用briggs，也没有可以合并的节点。可以冻结 t 和 r1 之间的 MOVE**4.** Simplify t 此时 degree[p] = 4,degree[s] = 4,degree[c] = 4

5. Spill c 仍然没有可以简化的节点，采用briggs也没有可以合并的节点，也不存在可以冻结的的节点。因此只能进行spill。计算 spill 分数，程序中没有循环，因此不需要考虑循环内的 def 和 use 选择 c 进行 spill 此时，degree[p] = 3, degree[s]=3

6.Spill s**6.** Simplify p 图中只剩下了预着色节点

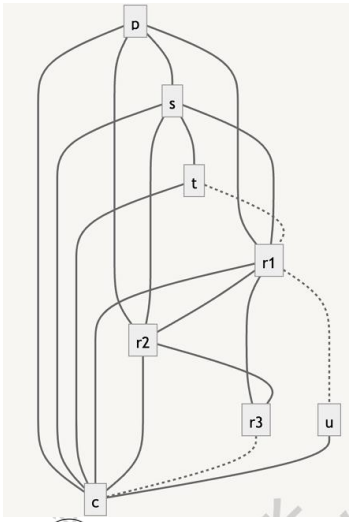
7. Select

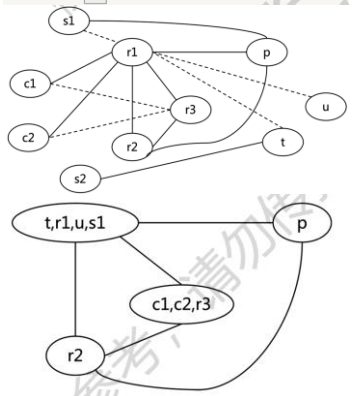
```
pop p, color r3
pop s, actual spill
pop c, actual spill
pop t, color r1
pop u, color r1
```

8. Rewrite

9.冲突图 **10.** Simplify s2 **11.**Coalesce t, r1, u, s1 由 Briggs 标准可知，他们可以被合并 **12.** Coalesce c1,c2,r3 由 Briggs 可知，可以被合并

```
f:  c ← r3
    p ← r1
    if p = 0 goto L1
    r1 ← M[p]
    call f              (uses r1, defines r1, r2)
    s ← r1
    r1 ← M[p + 4]
    call f              (uses r1, defines r1, r2)
    t ← r1
    u ← s + t
    goto L2
L1: u ← 1
L2: r1 ← u
    r3 ← c
    return              (uses r1, r3)
```





s1,t,u-r1;c1,c2,p-r3;s2-r2

CH13 垃圾收集

13.1 定义

1.垃圾(garbage) 在堆中分配且通过任何程序变量形成的指针链都无法到达的记录

2.保守近似：由于变量的活跃性不总能知道(停机问题的等价)，只是将不可到达的内存当作垃圾.

13.2 MARK-AND-SWEEP

1.算法原理：①标记阶段：使用 DFS 标记所有可达节点

```
function DFS(x)
    if x 是一个指向堆的指针
        if 记录 x 还没有被标记
            标记 x
            for 记录 x 的每一个域 f_i
                DFS(x.f_i)
```

Howjul.

```
标记阶段：
for 每一个根 v
    DFS(v)
```

Howjul.

②清扫阶段：未被标记的节点一定是垃圾：通过从头到尾扫描堆内存，对未标记的节点连接到空闲表(freelist)中.同时清除所有已标记节点的标记.

```
清扫阶段：
p← 堆中第一个地址
while p < 堆中最后一个地址
    if 记录 p 已标记
        去掉 p 的标记
    else 令 f_i 为 p 中的第一个域
        p.f_i ← freelist
        freelist ← p
        p ← p + (size of record p)
```

2.朴素算法的时间复杂度：标记阶段时间和标记节点个数成正比；清扫阶段时间与堆大小成正比.假设大小为 H 的堆中有 R 个字可到达数据，则一次垃圾收集的代价是c_1R+c_2H，c_1c_1为常数；好处是可用大小为$H-R$个字的自由储存单元补充空闲表. 通过摊还分析，最终的垃圾收集代价为$(c_1R+c_2H)/(H-R)$

3.朴素 DFS 改进：DFS 算法是递归算法，最坏情况下可能存在长度为 H 的路径。那么我们就需要 H 个活动记录来保存函数的各种信息（活动记录的大小一定远大于 1 个字），开销较大。

使用显式栈节约空间，可能生长到 H 大小个字（只有指针）(不是 H 个活动记录).但是辅助栈的空间大小与被分配的堆空间相同仍然不可接受.

```
function DFS(x)
  if x is a pointer and record x is not marked
    t ← 1
    stack[t] ← x
    while t > 0
        x ← stack[t];  t ← t − 1
        for each field f_i of record x
            if x.f_i is a pointer and record x.f_i is not marked
                mark x.f_i
                t ← t + 1;  stack[t] ← x.f_i
```

4.Deutch-Schorr-Waite（DSW）指针逆转 (pointer reversal)：在记录域 x.fi 被

压入栈后，不再查看原来的 x.fi，而是用 x.fi 存储其父节点的指针（指向父节点 x）

不使用显式的栈，而是重用图结点的某个部分来进行回溯。

当从栈中弹出 x.fi 的内容时，再将域 x.fi 恢复为原来值。 同时要求每个记录有一个名为 done 域，用以记录中有多少域已经被处理过.使用指针逆转的 DFS：

```
function DFS(x)
 if x 是一个指针并且记录 x 没有标记
  t ← nil
  标记 x; done[x] ← 0
  while true
       i ← done[x]
       if i < 记录 x 中域的个数
         y ← x.fi
         if y 是一个指针并且记录 y 没有标记
           x.fi ← t; t ← x; x ← y
           标记 x; done[x] ← 0
         else
           done[x] ← i + 1
       else
         y ← x; x ← t
         if x = nil then return
         i ← done[x]
         t ← x.fi; x.fi ← y
         done[x] ← i + 1
```

变量 t 用于指明栈顶，栈内每一个记录 x 都是已经标记的记录.如果 i=done[x]则 x.fi 是连接下面一个节点的"栈链".当对栈执行弹出操作时，x.fi 恢复为原来值.

5.空闲表数组：使用简单 freelist 的效率低，为了找到空间大小合适的记录需要找到很深.使用空闲表组成的数组，freelist[i]中存放大小为 i 的空闲区域.当要分配大小为 i 的记录时，从 freelist[i]的表头取一个即可.清扫垃圾时可以把大小为 j 的插入到 freelist[j]中.若想从 freelist[i]的空表中分配，可以从 freelist[j](j>i)抢夺一个较大的记录，再把剩余的(j-i)插入到 freelist[j-i].

6.碎片(fragment)：①外部碎片：想分配一个 n 大小的空间，但是空闲空间均小于 n②内部碎片：实际使用大小为 n 的分配了大小为 K 的空间(K>n)，未使用的空间在记录内而不是空闲记录中.

13.3 引用计数算法

1.算法原理：记住每一个记录有多少指针指向它.计数的域和每一个记录储存在一起.如果执行了某个语句之后，某个记录的引用计数变为 0，说明程序已经没有任何的变量会引用这个记录，那么就可以被回收。有一个改进是在将 r 从 freelist 中删除时，递归地减少 r.fi 的计数(不是在将 r 放到空闲表时，就马上减少 r.fi 的所指向的记录的计数，而是将 r 从空闲表中删除时，才去"递归地"减少）：(1)能将递归减少的动作分解为较短的操作，是程序的运行更加平滑(对交互式程序或实时程序好)(2)递归减少动作只需在一个地方进行，即在内存分配器中 **2.优缺点**：①优点：简单易于实现②缺点：(1)无法回收成环的垃圾.(2)增加引用计数所需的操作代价很大，启动大规模收集时可能会很慢.(3)显著减慢赋值操作

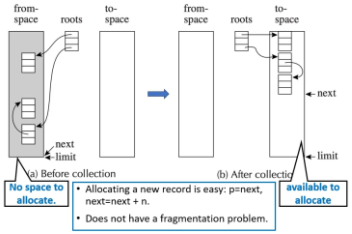
3.解决"环问题"办法：(1)简单地使用数据结构时显式地解开所有的环，但是难度较大(2)将标记清理和引用计数相结合.

13.4 复制式算法

没有 fragmentation 问题

1.算法概述：(1)堆中的可到达部分是一个有向图，遍历这个图（堆中称为 from-space）(2)在堆的新区域(称为 to-space)建立一个同构的副本.(3)原来指向 from-space 的所有的根在复制之后变成指向 to-space 副本；在此之后，整个 from-space(垃圾，加上原本是可到达的结点一起）便成为不可到达的

2.算法流程：●next 指针就是一个分配内存的辅助指针，需要分配内存的时候往下拉， limit 指针是整个区域的大小。当 next 指针和 limit 指针重合，说明已没有可供分配的内存。



●在收集之前，因为 next 已到达 limit，from-space 充满了可到达结点和垃圾，没有剩余的空间可用于分配，需要进行垃圾收集。●在收集之后，from-space 被我们清空，位于 next 和 limit 之间的 to-space 区域也可用于分配新记录。给指针 p 分配一个大小为 n 的记录：将 next 复制给 p，并使 next 增加 n 即可。●最终，程序将由于已分配了足够多的空间而使 next 到达 limit；于是需要另一次垃圾收集。此时 from-space 和 to-space 将交换角色，并再次复制可到达数据

3.传递指针算法伪代码：

```
function Forward(p)
 if p 指向 from-space
   then if p.f1 指向 to-space
          then return p.f1
          else for p 的每一个域 fi
                 next.fi←p.fi
               p.f1←next
               next←next + 记录 p 的大小
               return p.f1
   else return p
```

4.Cheney 算法：用 BFS 对可达数据进行遍历.

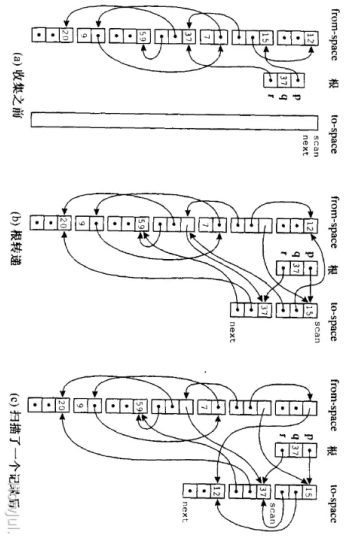
```
scan←next←to-space 的开始
for 每一个根 r
   r←Forward(r)
while scan < next
   for scan 处的那个记录的每一个域 fi
      scan.fi←Forward(scan.fi)
   scan←scan + scan 处的那个记录的大小
```

①位于 scan 和 next 之间区域包含的是已复制到 to-space 但子域还没有传递的记录：子域指向 from-space ②位于 to-space 开始和 scan 之间的是已复制并且以转递的记录，这一区域的所有指针均指向 to-space.③while 循环会使 scan 向 next 移动，复制记录也会导致 next 移动.④当所有可达数据都被复制到 to-space 后 scan 追上 next.

⑤算法优点：不需要外部栈和逆转指针；使用 scan 和 next 之间的区间作为 BFS 队列；实现简单.

⑥算法缺点：引用局部性差(没有把根相关的引用放到一起而是错开了，相邻的是不同变量引用.)

⑦算法代价：每一次回收$H/2-R$个字.摊还代价是每个分配字为$c_3R/(H/2-R)$条指令.



5.混合式算法(半深度优先搜索)：缓解 Cheney 空间局部性不好的缺点。摊还代价是每个分配字为$c_3R/(H/2-R)$条指令.当 H 远超 R 时代价接近于零，即没有固有的垃圾收集代价的下界，空间时间代价都很大.

13.5 编译器接口

1.快速分配：复制收集的分配空间使得分配的空间是连续的空间；区域的末端是 limit，next 指向下一个空闲单元.

2.分配大小为 N 的记录的步骤如下：①调用储存分配函数②测试 next+N<limit 是否成立(不成立则调用垃圾收集器)③将 next 复制到 result④清除 M[next]-M[next+N-1] 的内存⑤next:=next+N

```
function Forward(p)
  if p 指向 from-space
    then if p.f1 指向 to-space
           then return p.f1
           else Chase(p); return p.f1
    else return p

function Chase(p)
  repeat
    q ← next
    next ← next + 记录 p 的大小
    r ← nil
    for 记录 p 的每一个域 fi
      q.fi ← p.fi
      if q.fi 指向 from-space 且 q.fi.f1 不指向 to-space
      then r ← q.fi
    p.f1 ← q
    p ← r
  until p = nil
```

⑥从分配函数返回.A.将 result 传送到计算上有用的某个地方 B.将要用到的值储存到该记录.其中①⑥可以被内联拓展(inline expanding)消除；③可以与 A 结合被消除；④可以与 B 结合被消除；②⑤不可以被消除，但如果在同一基本块内有多个分配，则可以在多个分配间公用比较操作和自增操作，把 next 和 limit 放到寄存器里②⑤只需要 3 条指令.综上，分配记录的指令开销可以被减少到 4 条指令.

3.数据布局(data layout)描述：①收集器可以对任意类型记录进行操作②简单办法是让每个对象的第一个字指向特殊的类型(或类)的描述字记录(descripter)：包含对象的总大小以及每一个指针域的位置

4.指针映射(pointer map)：编译器需要向垃圾收集器标示出存放指针的临时变量和局部变量的位置。这些变量可能位于寄存器中或活动记录中。由于每条指令可能改变活跃临时变量的集合，指针映像在程序的每一点都是不同的。因此，一个较简单的方法是仅在那些可以启动垃圾收集的点(如 alloc 函数的调用点）描述指针映像。

5.导出指针(derived pointer)：导出指针涉及到从基指针（base pointer）派生出的指针。举例来说，对于表达式 a[i-2000]，内部会被编译器计算成 M[a-2000+i]。为避免重复计算，编译器会将一些计算提到循环外进行。如果循环中包含 alloc 操作且垃圾收集器在导出指针 t1 活跃时开始工作，收集器可能会混淆。因此，指针映像必须标识每一个导出指针，并指出其基指针，以确保收集器在重定位基指针时也能正确调整导出指针。**一个导出的指针将隐式地保持其基指针活跃.**

CH14&18 Dominators

必经结点：如果从 s 到结点 n 的所有有向边路径都经过结点 d，那么结点 d 是结点 n 的必经结点。●每个控制流图都一定有一个没有前驱的起始结点 s，这个结点是程序（或子程序）执行的假设开始点。●每一个结点都是自己的必经结点。●结点 n 可以有多个必经结点。

immediate dominator

在连通图中，假设 d 是 n 的必经结点，e 也是 n 的必经结点，则一定有 d e 的必经结点，或者 e 是 d 的必经结点.必经结点是有序的，不存在两个并列的必经结点。每个结点 n 都不会有超过一个的直接必经结点

loop-invariant 许多循环优化需要在紧挨着循环执行之前插入一些语句。循环不变量外提 loop-invariant hoisting：将一条语句从循环内移动到紧挨着的循环之前。(让它在循环外面先算好，而不是一次又一次地迭代)。将 d:t<-a xor b 外提到循环前置结点末尾的准则：

1.（对应(b)）如果在某些循环出口结点，t 是出口活跃的(之后还会被用到)，那么 d 是所有这样的循环出口结点的必经结点。

2.在循环中只有一个定义。

3.并且 t 不属于循环前置结点的出口活跃集合，人话就是在循环中 t 定义之前不能被使用。