CH2 词法分析

2.1 扫描处理

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 正则表达式

1.一些 notation

a	An ordinary character stands for itself.
ϵ	The empty string. Another way to write the empty string.
M | N	Alternation, choosing from *M* or *N*.
M · N	Concatenation, an *M* followed by an *N*.
MN	Another way to write concatenation.
M*	Repetition (zero or more times).
M+	Repetition, one or more times.
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

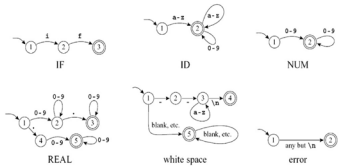
2.3 有穷自动机

1.DFA:M 由字母表Σ、状态集 S、转换函数 T:S×Σ→S、初始状态 $S_0 \in S$ 以及接受状态 A⊂S.

2.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.

3.NFA:M 由字母表Σ、状态集 S、转换函数 T:S×(Σ∪{ε})→P(S)、初始状态 S_0 及接受状态 A 的集合.

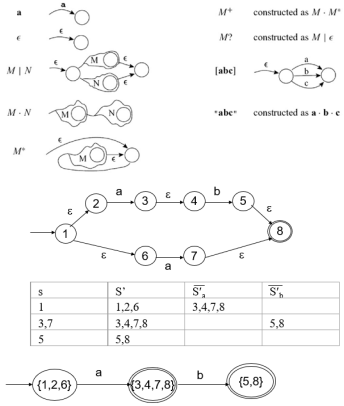
一些给出的基本 FA 图:



2.4 正则表达式到 DFA

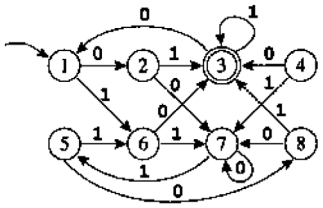
1.子集构造的过程:首先列出所有状态的ε闭包;然后将初始状态的ε闭包作为新的初始状态;然后计算在每个新状态下在各个字符上的转移的闭包作为新的状态,转移自然成为新的转移;包含原接受状态的所有新状态都是接受状.

PS:ε闭包首先包含自身.



s	S'	S'_a	S'_b
1	1,2,6	3,4,7,8	
3,7	3,4,7,8		5,8
5	5,8		

4.DFA 状态数最小化:一种正向思路:从 DFA 的终态出发**逆着找**,看出边、终点一致的就是等价态.以 2.6 为例:{4, 6} {2, 8} {1, 5}等价.



5. NFA-DFA 的步骤:给出一个表头如下:新状态编号 新加字符 新加旧状态 新状态代号;

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 分析数和抽象语法树

1.二义性(ambiguous):同一个串存在多个推导即多个分析树

2. 分析树(concrete sytax tree)是一个作了标记 labeled 的树,内部节点是非终结符,树叶是终结符;对一个内部节点运用推导时,推导结果从左到右依次成为该内部节点的子节点

3.最左推导和前序编号对应,最右推导后序

4.AST(syntax tree)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

3.3 Ambiguity 二义性

1.定义:带有两个不同的分析树的串的文法

2.解决方法①设置消歧规则 disambiguating rule,在每个二义性情况下指出哪个是对的.无需对文法进行修改,但是语法结构就不是单纯依赖文法了,还需要规则②修改文法.

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

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

6.相同优先级的运算符组叫 precedence cascade 优先级联

7.期中错题: A grammar is ambiguous if it has only one parse trees for all sentences.
Answer: T

自顶向下分析: LL(k)

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

递归下降:1.将一个非终结符 A 的文法规则看作将识别 A 的一个过程的定义.递归下降需要使用 EBNF;将可选[]翻译成 if,将重复{}翻译成 while 循环

LL(1)1. 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)**

2. LL(1)文法是无二义性的,对任意规则 $A \to \alpha 1 | \alpha 2$,First(α1)∩First(α2)为空,否则不是 LL(1).

3. 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.

4. 简单直接左递归:$A \to A\alpha | \beta$,αβ是 N,且β不以 A 开头.$A \to \beta A', A' \to \alpha A' | \varepsilon$

5. 普遍直接左递归:$A \to A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$
$A \to \beta_1 A' | \beta_2 A' | \dots | \beta_m A'$
$A \to \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \varepsilon$

6. 提取左因子:$A \to \alpha\beta | \alpha\gamma . A \to \alpha A', A' \to \beta | \gamma$

first follow sets:

1.Fisrt 定义:令 X 为一个 T 或 N 或ε,Fisrt(X)由 T 或ε组成.① 若 X 为 T 或ε,Fisrt(X)={X}②若 X 为 N,对于每个产生式 X→X_1 X_2 …X_n,First(X)都包含了 First(X_1)-{ε}.
若对于某个 i<n,所有的 Fisrt(X_1),…,First(X_i)都含有ε,则 First(X)也包括了 First(X_{i+1})-{ε}.若所有 Fisrt(X_1),…,First(X_n)都含有ε,则 First(X)也包含ε.

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

3.Follow 定义:若 A 是一个 N,那么 Follow(A)由 T 和$组成.①若 A 是$,直接进入 Follow(A) ②若存在产生式 B→αAγ,则 First(γ)-{ε}在 Follow(A)中 ③若存在产生式 B→αAγ,且ε在 First(γ)中,则 Follow(A) 包括 Follow(B)

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

4.First 关注点在产生式左边,Follow 在右边

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

5.LL(1)分析表 M[N,T]的构造算法:为每个非终结符 A 和产生式 A→α重复以下:
①对于 First(α)中的每个记号 a,都将 A→α添加到项目 M[A,a] ② 若 α 可空(nullable),则 Follow(A)中的每个元素 a (包括$),都将 A→α添加到 M[A,a]

例:下列文法构造 LL(1)

$Z \to d$ $Z \to XYZ$ $Y \to$ $Y \to c$ $X \to Y$ $X \to a$

1) 检查有无左递归左因子

2)nullable|First|Follow

	nullable	FIRST	FOLLOW
X	yes	a c	a c d
Y	yes	c	a c d
Z	no	a c d	

3) 按照算法填写规则进表

	a	c	d
X	$X \to a$ $X \to Y$	$X \to Y$	$X \to Y$
Y	$Y \to$	$Y \to$ $Y \to c$	$Y \to$
Z	$Z \to XYZ$	$Z \to XYZ$	$Z \to d$ $Z \to XYZ$

6.LL(1)判别:A grammar in BNF is LL(1) if the following conditions are satisfied. ① For every production Ai→ $\alpha_1 | \alpha_2 | \dots | \alpha_n$, $First(\alpha_i) \cap First(\alpha_j)$ is empty for all i and j, $1 \le i, j \le n, i \ne j$ ② For every non-terminal A such that First(A) contains ε, First(A)∩Follow(A)empty.

自底向上分析: LR(k)

Yacc 基于 LALR(1).

1.动作为①shift,将 T 从输入开头移到栈顶②reduce 使用产生式 A→α将栈顶的α规约成 A③accept 分析栈为开始符号,输入栈为空时的动作④error ⑤**转移(goto)**: 吃 non-terminal

2.加一个 S′:新的开始符号

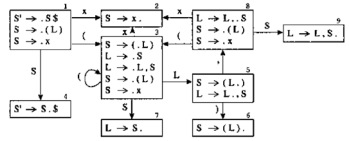
3.LR 分析器需要构造和使用一张 LR 分析表;而 LR 分析表的构造需要一个生成器.对于 start symbol 而言,需要新增 S′→S$规则来引入.

4.LR(0)的状态中包含带'.'的规则,'.'左边表示已读,右边表示未读;如果圆点'.'右边的第一个非终结符有规则,那么也写到这一状态中.

例如:如下文法构造 LR(0)

0 $S' \to S\$$
1 $S \to (L)$
2 $S \to x$
3 $L \to S$
4 $L \to L, S$

状态图如下:



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		

5.s-r conflict:表中同时出现了 sn 和 rn,即构成 s-r 冲突

6.r-r conflict:表中同时出现两个及以上的 rn,则构成 r-r

7.LR(0)文法不可能二义的

8.A grammar is LR(0) if and only if ①Each state is a shift state (a state containing only shift items) or a reduce state (containing a single complete item).

这里的 r1 都是用 S'→S·规约,应该写成 accept

SLR:

1.SLR 算法定义:移进规则不变;规约时要求输入必须在属于 follow(A)的终结符的项中,而不是全放.就是比 LR(0)改进

$R \leftarrow \{\}$
for each state I in T
  for each item $A \to \alpha.$ in I
    for each token X in FOLLOW(A)
      $R \leftarrow R \cup \{(I, X, A \to \alpha)\}$

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 冲突使用消岐规则:优先移进;rr 冲突基本是设计出问题

LR(1) and LALR(1)

1.LR(1) items:[A→α·β,a]前面是 LR(0)项,后面是 lookahead token

2.LR(1)的起始状态[S'→·S,$]的闭包

3.LR(1)中如何获得规则状态的闭包?难点在找 look ahead symbol.使用如下算法:

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

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

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

4.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

5.LR(1)文法不可能二义性

6. LR(1)分析表示例

	x	*	=	S	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s11	s13				g9	g7
5				r2			
6	s8	s6				g10	g12
7				r3			
8			r4	r4			
9				r1			
10			r5	r5			
11				r4			
12			r3	r3			
13	s11	s13				g14	g7
14				r5			

8.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,目的在于减少表项.

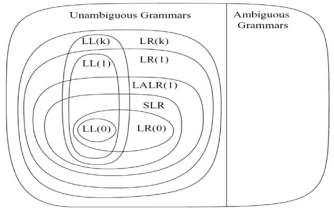
9.A grammar is an LALR(1) grammar if no parsing conflicts arise in the LALR(1) parsing algorithm.

10.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.

[A → α. a] [B → α. b] ⇒ [A → α. a/b] [B → α. a/b]

11.如果文法是 SLR(1),那么必然是 LALR(1).

12. 各类文法的层次如下:



13.悬挂 else(dangling else)的处理:悬挂 else 会导致 s-r 冲突,冲突文法如下:

S* → if *E* then *S* else *S
S* → if *E* then *S
***S* → other**

解决办法:引入新非终结符用于 if-else 的匹配问题.M 是匹配的 else;U 是未匹配的 else

S* → *M
S* → *U
M* → if *E* then *M* else *M
***M* → other**
U* → if *E* then *S
U* → if *E* then *M* else *U

CH5 语义分析

编译器完成的是 static semantic analysis

5.1 符号表(symbol table)

1.符号表,也成为**环境(environment),将标识符映射到类型和储存位置.**

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. **注意: 在 C/C++以及 Java 中,变量的作用域不可以交叉(scopes of vars cannot be intercrossed). 期中考.**

3.环境是由**绑定(binding)**组成的集合, 指标识符和含义之间的一种映射关系,用箭头表示.比如{g→string, a→int}

4. 符号表的有两种:命令式风格(imperative style)和函数式风格(functional style)

5.2 命令式(imperative)

1.散列表(bucket list)实现

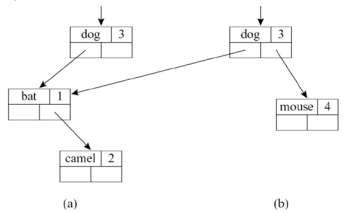
2.特点:破坏式更新:将变量名作为键值直接插入到现有散列表,外加变量的类型; 插入时将变量插入到对应 bucket 的链表头部,便于实现作用域的删除操作**以恢复环境(environment).**

5.3 函数式(functional)

1.实现方法:二叉搜索树(BST)

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),长效数据结构(persistent data structure);长效红黑树.

3. 例子:已知$m_1 = \{bat \to 1, camel \to 2, dog \to 3\}$,现在添加新绑定 mouse→4.需要创建复制 d 个 BST 节点(直到被插入的深度 d),剩余部分共用).然后插入新节点结果如下:



CH6 活动记录

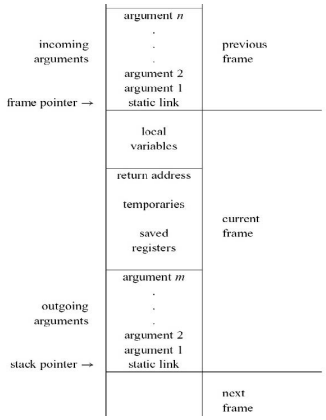
6.1 栈帧,也叫活动记录

1.**定义:**栈中存放函数的局部变量/参数/返回地址/临时变量的这片区域为该函数的活动记录(activation record)或栈帧(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.

期中错题:which of the following is commomly found in a stack frame? A. static vars B. sp pointer **C. saved frame** pointer D. global vars

3.栈帧的 layout: ①传入参数(incoming arguments) ②返回地址(由 call 指令创造) ③局部变量 ④传出参数(outgoing) ⑤静态链(static link).



4.参数传递: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**)

5.**返回地址:**call 指令地址的下一条指令地址.

6.栈帧内变量: 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)

7. 逃逸变量(elapsed 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.

8.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.

其他访问外层变量的方法：①嵌套层次显示表(display)：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(期中错过,管理的是静态链不是栈指针) ②λ提升(lambda 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. 中间表示(intermediate represent)：抽象的机器语言,链接前端和后端,解决了高级语言和目标机器汇编语言之间的转化(N*M → N+M)

2.基本概念：

①前端(front end)：词法分析|语法分析|语义分析|翻译成中间代码

②后端(back end)：IR 优化|翻译成机器语言.

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 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)：过程调用 ⑦ESEQ(s,e)：先计算语句 s 形成副作用,然后计算 e 违该表达式的值 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 ⑩ JUMP(e, labs)：跳转到 e 地址或者 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)：定会一名字后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 patch list)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**:a[i]表示为 MEM(+(MEM(e), BINOP(MUL, I, 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

8.1 规范树(canonical tree)

1.定义: 一颗不含 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, ….).

2.为什么? ①CJUMP 能够跳转到两个标号的任意一个,但实际的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

3.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

4.ESEQ 转化:①ESEQ(s1, ESEQ(s2,e))→ESEQ(SEQ(s1,s2), e) ② BINOP(op,ESEQ(s,e1), e2)→ESEQ(BINOP(op,e1, e2)) ③ MEM(ESEQ(s,e1))→ESEQ(s, MEM(e1)) ④ JUMP(ESEQ(s, e1))→SEQ(s,JUMP(e1)) ⑤ CJUMP(op,ESEQ(s,e1),e2,l1,l2)→SEQ(s, CJUMP(op, e1, e2, l1,l2)) ⑥ BINOP(op,e1, ESEQ(s,e2))→ESEQ(MOVE(TEMP t,e1), ESEQ(s,BINOP(op,TEMP t,e2))) ⑦ CJUMP(op,e1,ESEQ(s,e2),l1,l2)→SEQ(MOVE(TEMP t,e1), SEQ(s,CJUMP(op,TEMP t,e2,l1,l2))).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

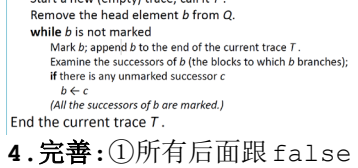
5.CALL 移到顶层:以 BINOP(op,CALL(),CALL(),….)为例, 第二个 CALL 会在 BINOP 执行前覆盖第一个 CALL 返回在 RV 寄存器里的值. 解决办法是使用 ESEQ 将返回值保存到一个新的临时变量里 :CALL(fun,args)→ESEQ(MOVE(TEMP t, CALL(fun, args)), TEMP t)

8.2 处理条件分支

1.基本块(basic block):取一列规范树,块的开始是 label,以跳转指令为结尾.即: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:从头到尾扫描语句序列,每次发现一个 LABEL 就开始一个新的基本块并结束上一个基本块;没发现一个 JUMP 或 CJUMP 就结束一个基本块(并开始下一个基本块).如果过程还遗留任何基本块不是 JUMP 或 CJUMP 结尾的,则在街边那块末尾增加一条转移到下一个基本快标号处的 JUMP;如果有任何基本块不是以 LABEL 开始的,则生成一个新的标号插入到基本块开始;在末尾添加 done LABEL, 将 JUMP(NAME done)放到最后一个基本快末尾.

3.轨迹(trace):程序执行期间可能连贯执行的语句序列.要寻找一组能够覆盖整个程序的轨迹集合,且每一个基本块仅出现在一条轨迹中.



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对气候跟随的既不是 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.

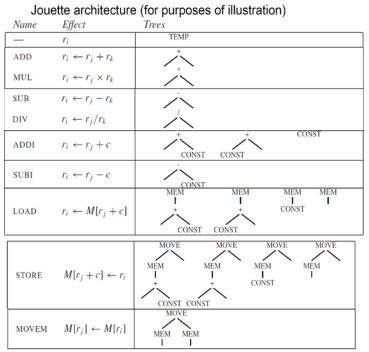
CH9 指令选择

9.1 树型到指令

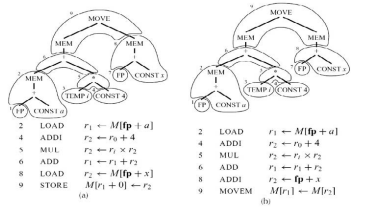
1.可以把一条机器指令表示成 IR 书的一段树枝,称为树型(tree pattern).

2.指令选择的任务就变成了使用树型的最小集合来覆盖(tiling)

3.使用 Jouette 体系结构,将树型映射成指令.指令和树型的映射如下表:



4.一棵树可以有多种 tiling 的方式,但是一定要按照给定的结构去 tiling:



5.最佳覆盖(optimum):瓦片的代价和可能是最小的覆盖,类似于全局最优.

6.最佳覆盖(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.

7.每一个最优覆盖同时也是最佳的,反之不然.

9.2 指令选择算法

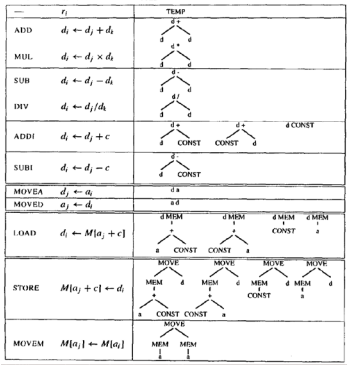
1.Maximal Munch 算法:①是一个最佳覆盖(optimal)算法 ②从树的根节点开始寻找适合他的最大瓦片,按照 Jouette 体系结构可能会覆

盖其他几个节点；对遗留的其他子树也进行相同操作 ③Maximal Munch 算法的 tiling 是从顶向下的，但是指令的生成是逆序的(很好理解，因为上层的覆盖指令需要下层的指令提供操作数，所以是逆序)．

2．动态规划：①可以找到最有覆盖，子问题是子树的覆盖 ②会给每个节点指定一个代价：可以覆盖该节点为根的字数的最优指令序列的指令代价之和．

3．树文法(Tree Grammar)：①动规的推广

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，但是动规可以很好处理．

下面是生成 load,move,moved 指令的树文法生成过程参考．

LOAD、MOVEA 和 MOVED：

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

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

1．RISC 机器特征：①32 个寄存器②仅有一类整数/指针寄存器③算数运算仅对寄存器进行操作④采用"三地址"指令(r1←r1+r2)⑤取指令和村指令只有 M[reg+const]模式⑥每条指令长度固定为 32 位⑦每一条指令产生一个结果或作用，无副作用

2．CISC 机器特征：①不多的几个寄存器(16,8,6)②寄存器分不同类型，某些操作只能在特定种类的寄存器上进行③算术运算可以通过不同的寻址模式访问寄存器和储存器④指令是"两地址"指令⑤有不同的寻址模式⑥有由变长操作码加变长寻址模式形成的变长指令⑦指令具有副作用(自增寻址方式)

3．CISC 机器的特点解决难题：①寄存器较少：不限制生成 TEMP 节点，假设寄存器分配能完成分配工作②寄存器分类：将操作数显示地传送到相应的寄存器中③两地址指令：增加一条额外的传送指令④算数运算可以访问存储器：指令选择阶段将每一个 TEMP 节点转化成一个寄存器引用．⑤若干种寻址模式：优点(破坏寄存器少；指令代码短)⑥变长指令：不管；⑦副作用指令：三种解决办法(a)忽略地址自增指令，希望其自动消失(b)在采取树型匹配的代码生成器的上下文中使用特别方式匹配方言(c)使用完全不同的指令算法，基于 DAG 样式．

CH10 活跃分析

10.0 定义

1．编译器需要分析程序的中间表示，以确定那些临时变量在同时被使用．如果一个变量的值在将来还需要使用，则变量是**活跃的(live)**，这种分析叫做活跃分析．

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

3．活跃范围：变量位索引的集合，变量在那几条边上活跃的边集合．

10.1 数据流方程的解

1．出边(out-edge)：节点引向后继节点的边；**入边(in-edge)：**由前继节点指向的边．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．

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．**公式 10.2** 如下：

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

2．活跃性计算的迭代方法：

```
for each n:
  in[n]←{};  out[n]←{};
repeat:
  for each n:
   in′[n]←in[n];
   out′[n]←out[n]
   in[n]:=use[n]∪(out[n]-def[n])
   out[n]:=∪in[s](s∈succ[n])
until  in′=in  &  out′=out
```

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 $O(N^2)$；repeat 循环的时间复杂度是 $O(N^4)$．由于活跃信息大部分稀疏，实际运行时间在 $O(N)$ 和 $O(N^2)$ 之间．

10.3 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．

10.4 最小不动点

1．数据流方程的解只是**保守的近似解**，只能保证生成的代码一定是正确的，但是产生代码的指令所使用的寄存器比实际需要的多．

2．定理：方程 10.2 有一个以上的解(in,out 计算公式方程)

3．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．数据流方程的解只是**保守的近似解**，只能保证生成的代码一定是正确的，但是产生代码的指令所使用的寄存器比实际需要的多．

2．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． **证明：**假设存在这样一个程序 H，我们会得出如下矛盾．从 H 构造函数 F，F(Y)=if H(X,Y) then (while true do()) else true．**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．

3．动态活跃：程序的某个执行从节点 n 到 a 的一个 use 之间没有经过 a 的任何 def，那么变量 a 在节点 n 时静态活跃的．

4．静态活跃：如果存在着一条从 n 到 a 的某个 use 的控制流路径，且此路径上没有 a 的任何 def，那么变量 a 在节点 n 静态活跃的．

10.6 冲突图

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

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．冲突表示：① 冲突矩阵：n*n 的矩阵，n 时临时变量的数目，(i,j)打叉表示 i,j 冲突．②冲突图：冲突矩阵的另一种表现形式．

4．绘制冲突图的办法：为新定值(def)添加冲突边的办法是 **(1)** 对变量 a 定值的非 MOVE 指令，以及在该指令节点 n 处，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,称之为"溢出"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.但是潜在溢出可能有节点颜色相同,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.重写后还需要重复上述所有流程,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:①Briggs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.**②George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,将节点分类为传送有关(move-related)和传送无关的(non-move-related).传送有关节点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:**每次一个地从图中删除低度数的传送无关节点,压入栈中. **③合并:**对简化的成果按照上述合并策略进行保守合并,并且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传送有关节点. **④冻结(freeze):**简化和合并都不能进行时,寻找一个低度数的传送有关节点,冻结这个节点所关联的那些传送指令,放弃合并的希望(把虚线画成直线)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):就是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.选择和合并阶段可以给普通临时变量分配与预着色相同的颜色,只要不相互冲突.

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.

5.溢出优先级计算公式: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:

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

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(优先溢出那些**不经常被使用的高度数**节点).

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(定义),同时重写原先的程序.

对def和use的修改如下:

①对于a的每一个use,都要新建一个临时变量a_i,从$M[a_{loc}]$位置读取a的值(因为a现在被存到内存中):$c \leftarrow a$变成$a_1 \leftarrow M[a_{loc}], c \leftarrow a_1$ ②对于a的每一次def,也要生成一个新的a_i,首先给a_i定值,然后a_i赋值到$M[a_{loc}]$:$a \leftarrow c$变成$a_2 \leftarrow c, M[a_{loc}] \leftarrow a_2$

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称之为垃圾(garbage).

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.

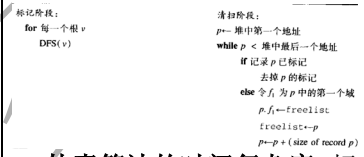
3.可达性(reachable): 程序变量和堆的记录构成一个有向图.每个程序变量是图中的一个根.如果存在一条从根节点出发到达n的有向路径,则称堆内存节点n是可到达的.

13.2 标记-清扫式算法

1.算法原理:①标记阶段:使用DFS标记所有可达节点 **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.

伪代码如下(最朴素的算法):

```
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)
```



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$(c_1R + c_2H)/(c_1R + c_2H)$

3.朴素DFS改进:使用显式栈节约空间,可能生长到H大小个字.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 是一个指针并且记录 x 没有标记
        标记 x
        t←1
        stack[t]←x
        while t > 0
            x←stack[t]; t←t-1
            for 记录 x 的每一个域 f_i
                if x.f_i是一个指针并且记录 x.f_i没有标记
                    标记 x.f_i
                    t←t+1;  stack[t]←x.f_i
```

4.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).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.f_i
        if y 是一个指针并且记录 y 没有标记
          x.f_i ← 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.f_i;  x.f_i ← 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,然后把剩余的(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便可以做到.计数的域和每一个记录储存在一起.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:(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(对对交互式程序或实时程序好)(2)递归减少的做法,递归减少动作只需要在分配器中进行.

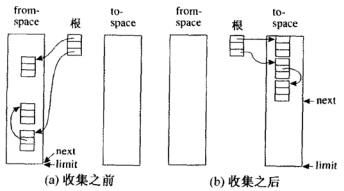
2.优缺点:①优点:简单易于实现②缺点:(1)无法回收成环的垃圾.(2)增加引用计数所需的操作代价很大.

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

1.算法概述:①将堆分为两块区域 from-space 和 to-space;当前的内存放到 from-space 中;②在 to-space 构造一个同构的副本,副本空间利用上更紧凑:占据连续的不含碎片的储存单元③ from-space 和 to-space 互换.

2.图例



3.算法流程: ①收集初始化:初始化指针 next 指向 to-space 的开始.每当 from-space 发现一个可到达记录,便把它复制到 to-space 的 next 所指位置,同时式 next 增加该记录的大小②转递(forwarding):使一个指向 from-space 的指针 p 转而指向 to-space(1)p 指向的使 from-space 中已经复制的记录,则 p.fi 是指明副本在何处的特殊的转递指针(2)p 指向 from-space 中一个尚未复制过的记录,则将它复制到 next 所指的位置;同时将转递指针赋给 p.f1(3)p 不是指针或者指向 from-space 以外的指针,对 p 不做任何事情.

4.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
```

5.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 循环回事 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.

6.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.时间复杂度也是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.

```
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
```

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] ⑤next:=next+N⑥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):①编译器必须能给收集器标出存放指针的临时变量和局部变量(寄存器中 or 活动记录)②由于每条指令都可能使活跃临时变量集合发生改变,故指针映像在程序的每一点都是不同的.③因此一个较简单的办法使仅在那些可以开始新的垃圾收集的点才描述指针映像;这些点才是 alloc 函数的调用点;每个函数的调用点也必须描述指针映像.④指针映像最好用返回地址作为键值:(1)为了找到所有根,收集器从栈顶向下扫描(2)每一个返回地址的键值对应一个指针映像的登记项,登记项描述下一个栈帧(3)每个栈帧内收集器从站真的指针开始标记(4)callee-save 的寄存器需要特殊处理:f 调用 g,g 调用 h,h 知道自己保存了 callee-save 的寄存器但是不知道那些使指针.所以 g 的指针影响必须指出在调 h 时他的 callee-save 的寄存器那些是指针那些从 f 继承.

5.导出指针(derived pointer):①对于表达式 a[i-2000],内部被编译器计算成为 M[a-2000+i]:对应 t1:=a-2000;t2:=t1+i;t3:=M[t2]

②为避免重复计算,会把 t1:=a-2000;提到循环外计算.③若循环中包含 alloc 且收集器在 t1 活跃时开始工作,那么收集器可能会被搞糊涂④此处定义 t1 是由基指针(base)a 导出的(derived).pointer map 必须标识每一个导出指针(derived pointer)并指出其导出的基指针.⑤就是收集器在把 a 重定位到地址 a'时,也必须调整 t1 到 t1+a'-a;把导出的指针和其基类相互绑定:只要基类活跃,导出指针也必须活跃.**一个导出的指针将隐式地保持其基指针活跃.**