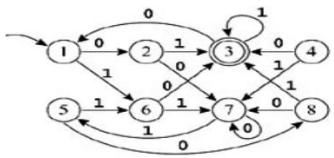
Ch1 绪论

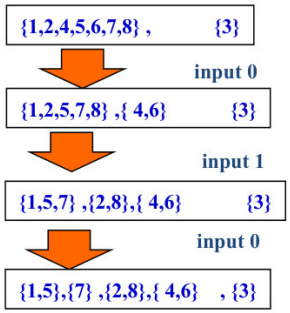
词法分析
将源文件分解为一个个独立的单词符号
语法分析
分析程序的短语结构
语义动作
建立每个短语对应的抽象语法树
语义分析
确定每个短语的含义，建立变量和其声明的关联，检查表达式的类型，翻译每个短语
栈帧布局
按机器要求的方式将变量、函数参数等分配于活跃记录(即栈帧)内
翻译
生成中间表示树(IR 树),这是一种与任何特定程序设计语言和目标机体系结构无关的表示
规范化
提取表达式中的副作用，整理条件分支，以方便下一阶段的处理
指令选择
将 IR 树结点组合成与目标机指令相对应的块
控制流分析
分析指令的顺序并建立控制流图，此图表示程序执行时可能流经的所有控制流
数据流分析
收集程序变量的数据流信息，例如，活跃分析(livenessanalysis)计算每一个变量仍需使用其值的地点(即它的活跃点)
寄存器分配
为程序中的每一个变量和临时数据选择一个寄存器，不在同一点活跃的两个变量可以共享同一个寄存器
代码流出
用机器寄存器替代每一条机器指令中出现的临时变量名

Ch2 词法分析

a	一个表示字符本身的原始字符
ε	空字符串
	空字符串的另一种写法
M | N	可选，在 M 和 N 之间选择
M · N	联结，M 之后跟随 N
MN	联结的另一种写法
M*	重复（0 次或 0 次以上）
M+	重复（1 次或 1 次以上）
M?	选择，M 的 0 次或 1 次出现
[a－zA－Z]	字符集。
.	句点表示除换行符之外的任意单个字符
" a . + * "	引号，引号中的字符串表示文字字符串本身

词法分析器消除二义性的规则:
最长匹配: 初始输入子串中,取可与任何正则表达式匹配的那个最长字符串作为下一个单词
规则优先: 对于一个特定的最长初始子串，第一个与之匹配的正则表达式决定了这个子串的单词类型。也就是说，正则表达式规则的书写顺序有意义。
DFA ,NAFA 绘图:箭头指向起始状态.接收状态双圆.
DFA 的最小化,例题见图:
根据输入和转移状态来分类，如果输入相同，转移的状态又是等价状态,即它们也等价.





Ch3 语法分析

1 推导:最左推导,即推导的中间符一直是最左边的那个(左侧除了终止符没有别的中间符)
最右推导:同最左.
2 语法分析树,即各个符号的推导过程依次连接形成树.
判断是不是 ambigous 的方式是能够有多个 parse tree. 或者多个最右推导.而不是多个推导(注意相同的语法树按不同顺序推导也是多种推导,但最右推导一定只有一种)
3 消除文法的二义性.
引入新的非终结符.,消除左递归.
引入优先级,通过更加紧密的约束提升优先级.(将乘除变为新增非终结符生成的表达式,即词法分析时会先将乘除法进行规约计算然后才计算加减,如此实现优先级)

二义性文法常常可以转换为无二义性的文法。一般通过文法转换来消除二义性，但是有些语言的二义性是固有的

预测分析,递归下降
只适合每个子表达式的第一个终结符号能够为产生式的选择提供足够信息的文法.
计算 first 和 follow 集合
Nullable: 若 X 可以导出空串，那么 nullable(X) 为真。
First: FIRST(y)是可从 y 推导出的字符串的开头终结符的集合。
Follow: FOLLOW(X)是可直接跟随于 X 之后的终结符集合。也就是说，如果存在着任一推导包含 Xt,则 t∈FOLLOW(X)。当推导包含 XYZt,其中 Y 和 Z 都推导出∈时,也有 t∈FOLLOW(X)
构造预测分析表
为了构造这张表，对每个 T∈FIRST(γ)，在表的第 X 行第 T 列，填入产生式 X→γ。 此外,如果 y 是可为空的，则对每个 T∈FOLLOW(X),在表的第 x 行第 T 列,也填入该产生式。
预测分析表 略

预测分析表不包含多重定义的想,说明文法就是 LL(1)的.

对于任何 k，不存在任何有二义性的文法是 LL(k)文法.

所有分析表和状态图见图

4 消除左递归
左递归:即定义 A 的推导式的右边第一个出现的就是 A.
右递归:同左递归反.
E → E+T E → T 转变为
E → TE′ E′ → +TE′ E′ →

5 提取左因子
对于以相同的符号开始的两个产生式可能出现问题.

$$S \rightarrow \text{if } E \text{ then } S \text{ else } S$$
$$S \rightarrow \text{if } E \text{ then } S$$

转变为

$$S \rightarrow \text{if } E \text{ then } S\ X$$
$$X \rightarrow$$
$$X \rightarrow \text{else } S$$

6 错误恢复
LR 分析
LL(k)分析技术的一个弱点是，它在仅仅看到右部的前 k 个单词时就必须预测要使用的是哪一个产生式。另-.种更有效的分析方法是 LR(k)分析,它可以将这种判断推迟至已看到与正在考虑的这个产生式的整个右部对应的输入单词以后(多于 k 个单词)。
LR(k)代表从左至右分析、最右推导、超前查看 k 个单词.
移进:将第一个单词压至栈顶
规约:使用文法 X→ ABC,从栈顶依次弹出 ABC,然后将 X 压入栈.
LR 分析表
Sn 移进到状态 n; gn 转换到状态 n
Rk 使用产生式(规则)k 进行规约
a 即接受.
遇到终结符使用移进,遇到中间符使用转换,到产生式末尾使用规约.
到文件末尾使用接受

错误,空项,停止分析,报告错误.

LR(0)分析器生成器 状态图
LR(0)分析器在遇到规约时,表中的所有终结符号都会填上规约动作,但是会导致冲突.
SLR 仅在 FOLLOW 集合指定的地方放置规约.
更详细地说,对于规约 A → xyz ,仅当 a 属于 FOLLOW(A)时,才在对应的状态和 a 处填上规约动作.
比 SLR 更强大的是 LR(1)分析算法。大多数用上下文无关文法描述其语法的程序设计语言都有一个 LR(1)文法。.
LR(1)的每个状态为转移前产生式的闭包.
即如果 点的位置位于一个中间符号,我们需要将该中间符号的所有产生式也填上,并再次闭包.
超前查看符可视为产生式的闭包时后面跟着的终结符,或 FOLLOW,相同的产生式可能有不同超前查看符.也是不同的项.

LALR(1) 分析表
由于不同超前查看符视作不同项,LR(1)有许多的状态,但是通过合并相同产生式的不同超前查看符,我们可以减少状态.
从而减小分析表大小
对于某些文法，LALR(1)表含有归约-归约冲突，因为仅超前查看符号不同的两个状态原来在接受同一字符时可能归约到不同的状态，而 LR(1)表中则没有这种冲突。不过实际中这种不同的影响很小，重要的是和 LR(1)表相比，LALR(1)分析表的状态要少得多，因此它需要的存储空间要少于 LR(1)表

冲突
Yacc 可以指出移进规约冲突(选择移进)和规约规约(选择文法中先出现的规则)冲突.

Ch4 抽象语法

Ch 5 语义分析

1 **编译器的语义分析(semantic analysis)阶段的任务是**:将变量的定义与它们的各个使用联系起来，检查每一个表达式是否有正确的类型,并将抽象语法转换成更简单的、适合于生成机器代码的表示.
符号表,表达式类型检查,声明的类型检查
2 符号表即环境,由一些绑定(binding)组成.

$$\text{绑定}\{g \mapsto \texttt{string}, a \mapsto \texttt{int}\}$$

3 对于符号表 X,Y 的运算 X+Y, 不可交换,且右侧符号表的相同符号覆盖左侧符号的绑定(即局部变量作用域的覆盖)
4 符号表的管理:有函数式的风格(即不同符号表切换时旧表保持不变,克隆一份后使用新表)和命令式风格(即直接使用一个表通过修改来形成符号表,另外维护一个"撤销栈",栈中保存撤销操作需要的所有信息,返回原环境时撤销即可)
5 命令式风格通常使用散列表实现,将左侧符号作为键值将右侧含义插入散列表中.散列表元素可以是散列链(栈),即如果存在覆盖,只需将新的含义加在链表头部(栈顶部),恢复时只需将其 pop 出来.
6 函数式符号表: 使用散列表会破坏原环境,复制的代价又太高.

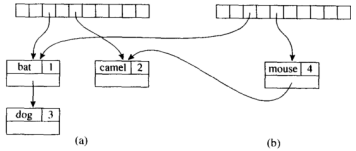


图 5-2 散列表

因此考虑使用二叉搜索树.在树的深度为 d 的地方添加新节点,必须创建 d 个节点.(节点根据字典顺序安排二叉树即 a<b<c...)

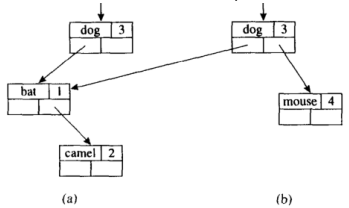


图 5-3 二叉搜索树

7 环境的创建和扩大是由程序中的声明导致.

Tiger 中声明只出现在 let 语句中.
8 对于递归的声明比非递归的声明麻烦许多,我们需要检查非法的递归类型声明,a->b->a,之类无有效"体"的声明.

Ch 6 活动记录

1 **嵌套函数定义**:即在一个函数内部再定义一个新函数,可能使用原函数的变量,这样就导致一个函数可能结束了,但是其变量未结束其生命周期.
如下 f() 返回自己定义的新函数 g(),g 中使用了 f 中的变量 x,而且调用不同 f,使用的 x 也不同.

```
fun f(x) =
  let fun g(y) = x+y
   in g
  end

val h = f(3)
val j = f(4)

val z = h(5)
val w = j(7)
```

(a) ML 语言编写的程序

嵌套函数和作为函数值返回函数导致函数内的局部变量需要的生命周期超过函数的生命周期.

2 栈帧
栈顶在低地址.后入先出.快速扩大收缩.
栈中用来存放一个函数的局部变量、参数、返回地址和其他临时变量的这片区域称为该函数的活动记录或栈帧.

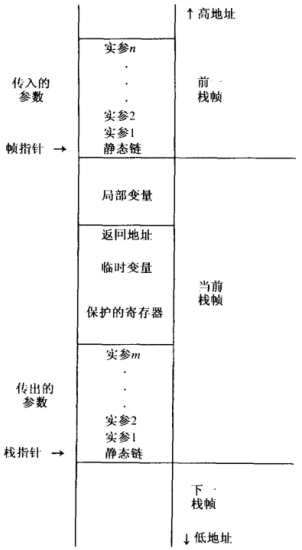


图 6-1 栈帧

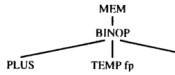
对于虎书,函数调用时的形参被视作在调用者的栈帧中..
返回地址实际上可能和传递的形参放在一起,作为"参数 0".即放在局部变量上面.
图中静态链实际上是动态链(翻译问题?),
动态链即上一个环境的 fp,而静态链据理解应该是定义时环境的 fp(对于 C 语言可能就是 main 环境,对于支持嵌套函数的自然大有不同)

静态链通常放在栈帧最底部,动态链放在栈顶.
fp 即上一个 sp,又或者是本栈帧的头部.调用函数时.sp-size 等于新 sp,然后将旧 fp 存入动态链,sp 复制到 fp 中.
调用者保护寄存器(caller-save)和被调用者保护寄存器(callee-save),参数传递可能有未失效的值在寄存器中需要保护.如何使用寄存器?
1 不调用其余过程称为叶过程.叶过程不必将参数保存至存储器中,常常可以不为它们分配栈帧.
2 有些优化编译器使用过程间寄存器分配,一次性分析程序中所有函数,给不同过程指派不同寄存器.
3 如果寄存器中保存的是失效变量,即不再使用,那么可以进行覆盖.
4 某些体系结构有寄存器窗口,使每次函数调用都分配一组新的寄存器,无需存储访问.
悬挂引用:函数形参的地址比函数本身的生命周期长.
任何取了地址的参数,我们需要在函数入口处将其放在存储器中.传递局部变量参数最好使用传地址的方法,传递实参而非形参.
Call 返回地址存储在寄存器中快,但是非叶过程需要将其存储在自己的栈帧中.
如果一个变量是传地址实参，或者它被取了地址(使用 C 语言中的&操作),或者内层的嵌套函数对其进行了访问，我们则称该变量是**逃逸**的(escape).
六种存储在栈帧中的情况:
1.作为传地址参数 2.被嵌套过程访问(不确定是否需要地址,当需要)3.变量值太大,无法使用单个寄存器(有些编译器可能使用多个寄存器存储)4.数组类型,使用地址访问元素 5.需要存放该变量的寄存器另有特殊用途,例如传参.6.局部变量和临时变量太多.
每当调用雨数 f 时，便传递给 f 一个指针，该指针指向静态包含 f 的那个函数，称这个指针为静态链(static link).
静态链(动态链)可以访问外层函数的栈帧,另外还有几种方法:
建立一个全局数组，该数组的位置 i 处包含一个指针，它指向最近一 次进入的,其静态嵌套深度是 i 的过程的栈帧，这个数组叫做**嵌套层次显示表(display).**
当 g 调用 f 时，g 中每-个实际被 f (或被嵌套在 f 内的任意函数)访问了的变量，都将作为额外的参数而传递给 f.这称之为**入-提升(lambda lifting)**

Ch7 翻译成中间代码

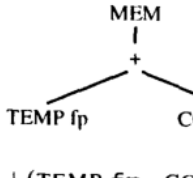
将抽象语法翻译成中间代码有利于可移植性.
1 中间表示(intermediate representation,IR) 是- '种抽象机器语言，它可以表示目标机的操作而不需太多地涉及机器相关的细节.而且它也独立于源语言的细节.编译器的前端(front end)进行词法分析,语法分析和语义分析，并且产生中间表示.编译器的后端(back end)对中间表示进行优化并将中间表示翻译成机器语言.
虎书的表达式树:
1 CONST(i)整型常数 i，用 C 语言写作 T Const(i).
2 NAME(n)符号常数 n (相当于汇编语言中的标号).用 C 语言写作 T. _Name(n).
3 TEMP(1)临时变量 t，抽象机器中的临时变量类似于真实机器中的寄存器，但抽象机器可以有无数个临时变量.

4 BINOP(o,e1 ,e2)对操作数 er、er 施加二元操作符 o 表示的操作,子表达式 e1 的计算先于 e2.
注意中间代码和源语言独立,所以无论源语言有什么操作符,中间代码都是一定的,可能需要将没有的操作运算进行转换.
5 MEM(e)开始于存储器地址 e 的 wordSize 个字节的内容.
6 CAL(f,I) 过程调用:以参数表 I 调用函数 f.子表达式 f 的计算先于参数的计算，参数的计算则从左到右.
7 ESEQ(s,e)先计算语句 s 以形成其副作用，然后计算 e 作为此表达式的结果.
语句执行副作用和控制流
8 MOVE(TEMPt,e) 计算 e 并将结果送入临时单元 t
9 MOVE(MEM(e),e) 计算 e，由它生成地址 a.然后计算 e,并将计算结果存储在从地址 a 开始的 wordSize 个字节的存储单元中.
10 EXP(e) 计算 e 但忽略结果.
11 JUMP(e,labs) 将控制转移到地址 e，目标地址 e 可以是文字标号
12 CJUMP(o,e1,e2,t,f)依次计算 e1.e2， 生成值 a、b,然后用关系操作符 o 比较 a 和 b.如果结果为 true,则跳转到 t;否则跳转到 f.
13 SEQ(s,s2)语句 s,之后跟随语句 s2。
14 LABEL(n)定义名字 n 的常数值为当前机器代码的地址。这类似于汇编语言中的标号定义。值 NAME(n)可能是转移或者调用等操作的目标。
对于在当前过程中声明的存放在栈帧中的简单变量 v.



MEM(BINOP(PLUS, TEMP fp, CONST k))

其中 k 是在栈帧内的位移，TEMP fp 是帧指针寄存器。
简写
BINOP(PLUS ,e_1 ,e_2)将简写为 +(e_1 ,e_2)



+(TEMP fp, CONST k)

函数调用
CALL(NAME l_f, [sl, e_1, e_2, ..., e_n])

Ch8 基本块和轨迹

如果树表达式不包含 ESEQ 和 CALL 节点,那么可以按照任意顺序执行它们.
常数可与任何语句交换,空语句可与任何表达式交换,其他都假定是不可交换的.
规范树:
1. 无 SEQ 或 ESEQ
2. 每一个 CALL 的父亲不是 EXP()就是 MOVE(TEMP t, ...)

参考最后一页树的等价形式消除 ESEQ.
将 CALL 移动到顶层
CALL 语句使用相同的结果返回寄存器
+(CALL(), CALL())
第二个调用将会在 PLUS 能够执行之前覆盖 RV 寄存器。
重写规则:
CALL(f, args) ==>
ESEQ(MOVE(TEMP t_{new}, CALL(f, $args$)), TEMP t_{new})

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

处理分支条件
基本块是语句组成的一个序列，控制只能从这个序列的开始处进入并从结尾处退出，即
●第一个语句是一个 LABEL。
●最后一个语句是 JUMP 或者 CJUMP。
●没有其他的 LABEL JUMP 或者 CJUMP。
将一长串语句序列划分成基本块相当简单。其方法是:从头至尾扫描语句序列，每发现一个 LABEL，就开始一个新的基本块(并结束前一个基本块);每发现一个 JUMP 或 CJUMP,就结束- -个基本块(并开始下一个基本块)。如果这个过程遗留有任何基本块不是以 JUMP 或 CJUMP 结束的，则在这个基本块的末尾增加一条转移到下一个基本块标号处的 JUMP。如果遗留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该基本块的开始。
轨迹集合:即基本块的跳转路径.注意轨迹集合要尽量少而且同一个基本块不要在不同轨迹出现.
完善调整基本块:
1 所有后面跟有 false 标号的 CJUMP 维持不变(许多 CJUMP 都是这种情况)。
2 对任何其后跟有 true 标号的 CJUMP，交换它们的 true 标号和 false 标号并将其条件更改成相反的条件。
3 对其后跟随的既不是它的 true 标号也不是它的 false 标号的 CJUMP(cond,a,b,1,,l).
生成一个新的标号 l}',并用如下三条语句重写该 CJUMP 语句，使它的 false 标号紧跟后面

CJUMP($cond$, a, b, l_t, l'_f)
LABEL l'_f
JUMP(NAME l_f)

Ch9 指令选择

1 中间表示语言每个节点只能表示一种操作,但是实际及其指令往往能完成多个操作.
2. jouette 体系结构参见最后的表
使用 瓦片(tile)来覆盖 IR 树
3 指令选择的目的是使用一组不重叠的瓦片来覆盖 IR 树
4 树的最好覆盖对应于代价最小的指令序列:即最短的指令序列，或者当指令的执行时间各不相同时，总执行时间最短的指令序列。
最佳覆盖:其中不存在两个相邻的瓦片可以连接成一个代价更小的瓦片覆盖.
最优覆盖:其瓦片的代价之和可能是最小的覆盖.
每个最优覆盖一定是最佳的,但是反之则不然.
5 最佳覆盖算法更加简单;
对于 CISC 指令集,由于其指令的瓦片可能相当大,最优覆盖和最佳覆盖之间可能有差别,虽然不是很大.
而对于 RISC 架构,瓦片很小且代价一致,其最有覆盖和最佳覆盖通常完全不存在差别.
6 maximal munch 算法
从根节点开始寻找覆盖节点最多的瓦片.对剩余未覆盖的每个子树依次寻找.
寻找最佳覆盖而非最优覆盖.
7 动态规划算法
寻找最优覆盖
动态规划算法给树中每个结点指定一个代价，这个代价是可以覆盖以该结点为根的子树的最优指令序列的指令代价之和。
8 快速覆盖: 两种算法都需要检查与该节点相匹配的所有瓦片.为节点 n 匹配一个瓦片,通过 switch-case 来考虑仅以该标号(MEM)为根的树型.
9 覆盖算法效率: 和其余处理步骤相比,快!!

Ch10 活跃分析

概念:
1 如果一个变量的值定义后(赋值后)将来还需要使用,称这个变量是活跃的.即从赋值后到下一次赋值之间,赋值到使用的语句路径都是活跃的,从最后一次使用到下一次赋值是不活跃的.
2 程序的控制流图:注意仅包含普通的表达式语句,跳转语句都转换成了箭头路线.条件判断也是表达式.
3 流图的每个节点都有入边和出边,每个赋值语句称为变量定值(define)，右边为变量使用(use),比较表达式涉及的变量都是 use.我们可以定义每个节点的 def(x)和 use(x)集合
4 活跃性。-一个变量在一条边上是活跃的是指,如果存在着-条从这条边通向该变量的一个 use 的有向路径，并且此路径不经过该变量的任何 def。如果-一个变量在一个结点的所有入边上均是活跃的，则该变量在这个结点是入口活跃的(live-in);如果一个变量在一个结点的所有出边上均是活跃的，则该变量在该结点是出口活跃的(live-out)。
5 活跃性计算:5.1 一个变量属于 use(x),即它在 x 节点为入口活跃 5.2 一个变量在 n 是入口活跃的,那么在 n 的所有前驱节点都是出口活跃的.5.3 一个节点是出口活跃的,且不属于 def(x),则该节点是入口活跃的.
数据流方程

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

6 对于活跃计算最好使用沿控制流边的反向计算,即从最后一个节点向前,由 out 计算 in
可参见作业
7 最小不动点:数据流方程的任何一个解都是保守的近似解,即保证不会将活跃变量视作死亡;但是会将不活跃变量视为活跃变量.
非最优但不会得到错误的结果.
在使用时我们需要使寄存器的个数尽可能少
8.静态活跃性和动态活跃性:对于判断 a^2+1>0,很容易判断只会有一条路径，但是没有编译器可以判断.
动态活跃。如果程序的某个执行从结点 n 到 a 的-一个使 用之间没有经过 a 的任何定值，那么变量 a 在结点 n 是动态活跃的。
静态活跃。如果存在着-条从 n 到 a 的某个使用的控制流路径,且此路径上没有 a 的任何定值,那么变量 u 在结点 n 是静态活跃的。
即任意和存在的区别.
如果 a 是动态活跃的,那么也必然是静态活跃.
9 冲突图:
(I)对于任何对变量 a 定值的非传送指令，以及在该指令处是出口活跃的变量 b1,..., bj，添加冲突边(a,bi).,.,(a,bj)。
(2)对于传送指令 a <- c,如果变量 b,...,b,在该指令处是出口活跃的，则对每一个不同于 c 的 b,添加冲突边(a,b),...,(a,b,)。
虚线指出传送指令,即将一个寄存器的值存到另一个寄存器.

Ch11 寄存器分配

1 我们这个编译器在 Translate.Canon 和 Codegen 阶段均假定有无限个寄存器可以用于存放临时变量,同时假定 MOVE 指令没有代价。寄存器分配器的任务就是将大量的临时变量分配到计算机实际具有的少量机器寄存器中.
前面分析后我们可以得到一个冲突图,将分配问题转换成图着色问题.
2 四个处理阶段:构造,简化,溢出,选择.
构造冲突图,计算每个程序点同时活跃的变量集合.
简化:(简单的启发式图着色算法)
于栈(或递归)的图着色算法:这个算法重复地删除度数小于 K 的结点(并将它压人栈中)。每简化掉一个结点都会减少其他结点的度数，从而产生更多的简化机会。(K 即代表寄存器个数,相邻的结点即无法共用寄存器的变量,相邻结点数小于 K,说明可以同时用寄存器存储)
溢出:再简化过程中某一图只包含高度数的结点,即度数>=K,此时启发式算法已经无用,因此我们标记一个结点为溢出的节点(即该变量需要存储在存储器而不是寄存器.我们对溢出结果作乐观估计,希望不会与余留的结点发生冲突)然后将该结点从图中删除并继续简化.
(注意即使结点的邻居数>K 也不一定会有实际溢出,因为邻居不相邻的话是可以相同颜色的)
选择:即从栈中弹出结点并赋予颜色.
此时我们可能遇到添加结点的所有邻居的颜色已经达到 K 种,即此时我们遇到了实际溢出.
此时我们不指定颜色,而是继续执行选择来识别其余实际溢出.
重新开始:一旦我们不能为某些结点赋予颜色,即遇到了实际溢出,我们需要对程序改写,将溢出的变量存储到存储器中.这样将一个溢出的变量转变为几个小范围活跃的新变量.然后再次执行该算法直到无溢出.

3 合并: 利用冲突图很容易删除冗余的传送指令。如果在冲突图中，一条传送指令的源操作数和目的操作数对应的结点之间不存在边，那么可以删除这条传送指令。它的源操作数结点和目的操作数结点可以合并(coalesce)成新的结点，这个新结点的边是被合并的这两个结点的边的并集。
合并引入的新结点受到的限制比原来的单一结点大得多.因为邻居是原结点之和.因此盲目合并不可取.
安全合并策略:
Briggs:如果合并后的结点的高度数邻居的个数小于 K 个,则可以合并.
George:结点 a 和 b 可以合并的条件是:对于 a 的每一个邻居 t，或者 t 与 b 已有冲突，或者 t 是低度数(度<K)的结点。通过下述推理可以证明这种合并是安全的。令 s 为原图中结点 a 的度<K 的邻结点组成的集合。若不进行合并，简化可以移去 s 中的所有结点，得到一个变小了的图 G1。如果进行合并，则简化也可以移去 S 内的所有结点，得到图 G2。但是，G2 是 G1 的子图(结点 G2 中的 ab 对应于 G,中的 b),因此它至少会比 G1 更容易着色。
合并不成功仍然是安全的.
带有合并的图着色.
构造冲突图.并将结点分为传送有关的或传送无关的.
简化:每次删除一个与传送无关的低度数结点.
合并(coalesce):对简化图实行保守的合并.
冻结(freeze):如果简化和合并都不能进行,那么寻找一个低度数的传送相关结点(即将其视为传送无关),然后继续进行简化和合并.
溢出:如果最后还是没能继续简化,对一个高度数结点进行溢出标记并将其压栈.继续简化
选择:弹出结点选择颜色..

当存在溢出时,重新进行一次构造和简化,需要忽略所有的合并再重新进行.
4 合并溢出:存在多个溢出变量时,分裂变量有可能导致进一步溢出以及其余情况. 溢出的偶对中有很多从不会同时活跃，通过合并便有可能对这些结点着色，事实，上，对栈帧单元的数量并没有固定的限制,考虑较激进的溢出策略:
(1)使用活跃信息构造被溢出结点的冲突图。
(2) 如果传送指令关联的一对溢出结点不相冲突，合并它们。
(3)使用简化和选择对图着色。在着色过程中不会有(进-一步的)溢出;替代地，简化阶段只是挑选度数最低的结点，选择阶段则取第一个可用的颜色，它不对颜色数量预先设定任何限制。
(4)这些颜色对应于被溢出变量在活动记录中的存储单元。
5 **预着色的结点**:机器寄存器,特殊作用寄存器?即传递参数一类的寄存器等.
预着色寄存器的变量可以通过保守合并和非预着色的结点合并.
我们不能简化预着色结点(指派颜色),预着色结点的颜色固定,即寄存器是固定的.度数为无限大.
6 如果一个变量不是跨过程活跃的,那么它往往会被分配到调用者保护寄存器中.
7 图着色的实现:

Ch13 垃圾收集

1 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)。垃圾占据的存储空间应当被回收，以便分配给新的记录，这一过程叫做垃圾收集(garbage collection)。垃圾收集不是由编译器完成的，而是由运行时系统完成的,运行时系统是与已编译好的代码连接在- -起的一些支持程序。
2 我们要求编译器保证所有活跃变量都是可到达的,并尽可能减少哪些可到达但是非活跃的记录数量.
MARK -AND -SWEEP COLLECTION 标记-清扫式收集:使用深度优先搜索,标记出所有可到达的点.查找到未标记的点进行清扫,用一个链表链接在一起,(空闲表),清扫阶段还要清除所有已标记节点的标记,等待下一次垃圾收集.

标记阶段:
for 每一个根 v
DFS(v)

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_1 为 p 中的第一个域
$p.f_1$←freelist
freelist←p
p←p + (size of record p)

代价:深度优先搜索的代价和它可以标记的点数有关,垃圾清扫的时间和堆的大小有关.
there are R words of reachable data in a heap of size H. Then the cost of one garbage collection is $c_1R + c_2H$ for some constants c_1 and c_2;
the *amortized cost* of collection by dividing the *time spent collecting* by the *amount of garbage reclaimed*:

$$\frac{c1R+c2H}{H-R}$$

使用显式栈的深度优先搜索 **Using an explicit stack..参见图**

指针逆转: Pointer reversal.
指针逆转。在记录域 x.f 的内容被压人栈后，算法 13-3 将不再查看原来的 x.f。这意味着我们可以使用 x f,来存储栈自身的一个元素!这种极聪明的思想称作指针逆转(pointer reversal) ,因为它能使 x. f 反指向这样一个记录:从该记录可到达 x。之后，当从栈中弹出 x. f,的内容时，再将域 x.f 恢复为它原来的值。
使用指针逆转的深度优先搜索.

```
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
```

每个记录有一个 down 的域用于记录该记录多少个域被处理.
变量 t 用于指明栈顶,栈内的每个记录 x 都是标记了的记录,如果 I == down[x]则 x.fi 是链接下面一个节点的栈链.当对栈执行弹出操作时,x.fi 恢复为原来的值.
注意该参数是从程序变量中取的,不是按堆栈的顺序来.

空闲表数组:
使用若干空闲表组成的数组,让 freelist[i]为所有大小为 i 的记录组成的链表.如果要取的大小为空链表,那么可以从较大的链表中分配一个.
外部碎片和内部碎片.

REFERENCE COUNTS 引用计数

当某个记录 r 的计数减少到 0,那么将其添加到空闲表,并将其指向的所有记录数减少.
问题(1)无法识别环状相互引用.代价比较大.
两种方法解决环:程序员使用数据结构时显式解开所有环,另一种将引用计数和标记清扫结合.

COPYING COLLECTION

堆中的可到达部分是一个有向图，堆中的记录是图中的结点，指针是图中的边，每- - 个程.序变量在图中是一个根。复制式垃圾收集(copying garbage cllction) 遍历这个图(堆中称为 from-space 的部分)，并在堆的新区域(称为 to-space) 建立一个同构的副本。副本 to-space 是紧凑的，它占据连续的、不含碎片的存储单元(即在可到达数据之间没有零散分布的空闲记录)。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，加上以前是可到达的图一.起)便成为不可到达的。
复制式收集没有碎片问题.
见图

收集的初始化。为了开始一次新的收集，初始化指针 next 使其指向 to-space 的开始。每当在 from-space 发现一个可到达记录，便将它复制到 to-space 的 next 所指的位置，同时使 next 增加该记录的大小。

转递:即将一个指向 from-space 的指针转而指向 to-space.
(1) p 指向已复制的记录,p.f1 是一个指明副本在何处的特殊的转递指针,通过指针指向 to-space 可以识别该类转递指针.
(2) p 指向未复制的记录,将其复制到 next 指向的区域,同时将转递指针赋给 p.f1.此时因为已经有了复制记录,我们可以覆写原记录的 f1 域.
(3) p 不是指针或者指向 from-space 外的指针,不进行操作.

转递指针

```
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
```

根据情况将 p 指向的记录复制到 to-space 内
返回 to-space 内的地址.

Cheney 算法

```
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
```

遍历所有记录的所有域,将内容全部复制到 to-space 中并修改指针地址.

Yacc Lex

1 Yacc can not use ambiguous grammars. 错误
2 Scopes of the variables are intercrossed sometimes. 变量的定义域?错误
3 Code generation depends on detailed information about the target architecture, and doesn't care the characteristics of the source language. 要看实现方式,如果没有中间代码,和源语言就有关了？错误

4 The best choice of data structure of the symbol table is HASH table. 符号表的数据结构,即哈希表？老旧版本?

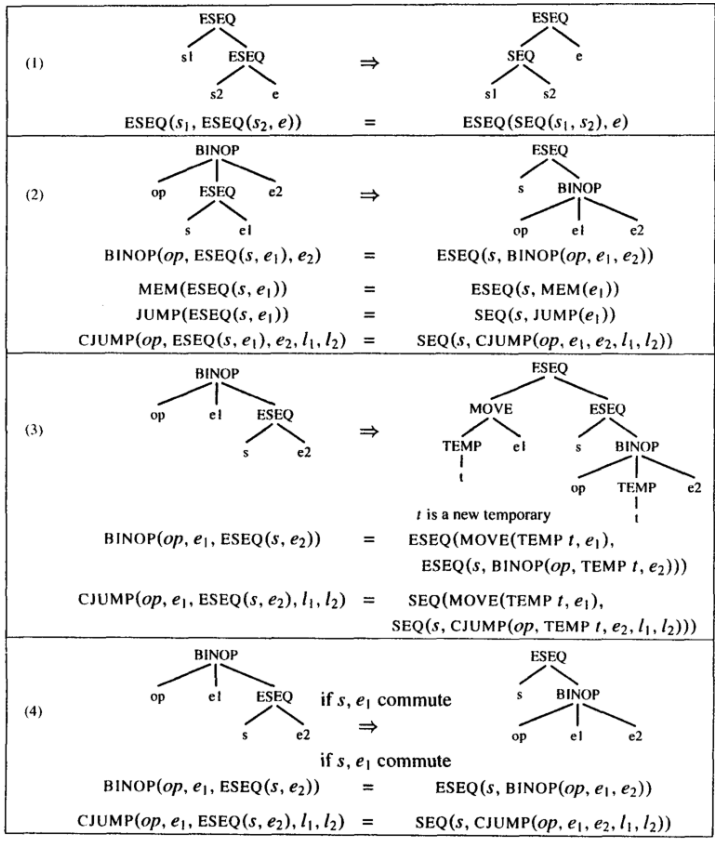
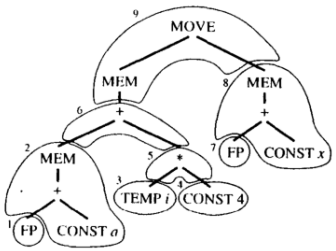


图 8-1 树的等价形式（同时参见习题 8.1）

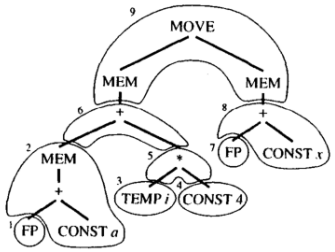
指令名	作用	树型
—	r_i	TEMP
ADD	$r_i \leftarrow r_j + r_k$	+
MUL	$r_i \leftarrow r_j \times r_k$	*
SUB	$r_i \leftarrow r_j - r_k$	-
DIV	$r_i \leftarrow r_j / r_k$	/
ADDI	$r_i \leftarrow r_j + c$	+ CONST; + CONST CONST
SUBI	$r_i \leftarrow r_j - c$	- CONST
LOAD	$r_i \leftarrow M[r_j + c]$	MEM + CONST; MEM + CONST CONST; MEM CONST; MEM
STORE	$M[r_j + c] \leftarrow r_i$	MOVE MEM + CONST CONST, MOVE MEM + CONST CONST, MOVE MEM CONST, MOVE MEM
MOVEM	$M[r_j] \leftarrow M[r_i]$	MOVE MEM MEM

图 9-1 算术和存储器存取指令。$M[x]$ 是地址为 x 的存储单元



(a)

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



(b)

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

指令名	作用	树型
—	r_i	TEMP
ADD	$d_i \leftarrow d_j + d_k$	d + d d
MUL	$d_i \leftarrow d_j \times d_k$	d * d d
SUB	$d_i \leftarrow d_j - d_k$	d - d d
DIV	$d_i \leftarrow d_j / d_k$	d / d d
ADDI	$d_i \leftarrow d_j + c$	d + d CONST; d + CONST d; d CONST
SUBI	$d_i \leftarrow d_j - c$	d - d CONST
MOVEA	$d_j \leftarrow a_i$	d a
MOVED	$a_j \leftarrow d_i$	a d
LOAD	$d_i \leftarrow M[a_j + c]$	d MEM + a CONST; d MEM + CONST a; d MEM CONST; d MEM a
STORE	$M[a_j + c] \leftarrow d_i$	MOVE MEM + a CONST d, MOVE MEM + CONST a d, MOVE MEM CONST d, MOVE MEM a d
MOVEM	$M[a_j] \leftarrow M[a_i]$	MOVE MEM a MEM a

图 9-3 “患有精神分裂症的” Jouette 体系结构

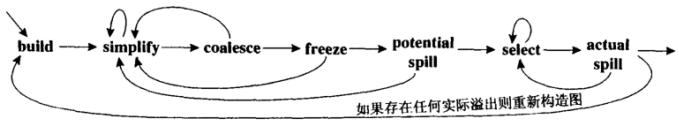


图 11-4 带合并的图着色

算法 13-3 使用显式栈的深度优先搜索

```
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
```

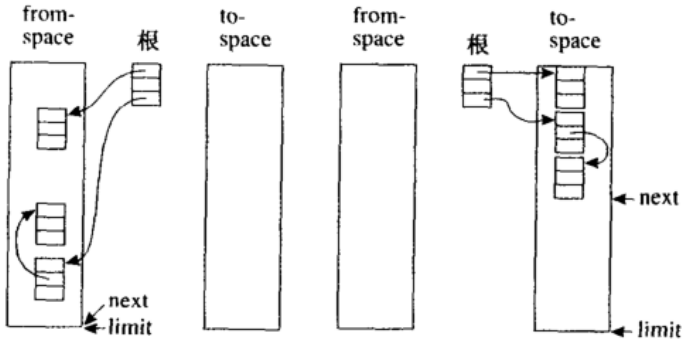


图 13-3 复制式收集

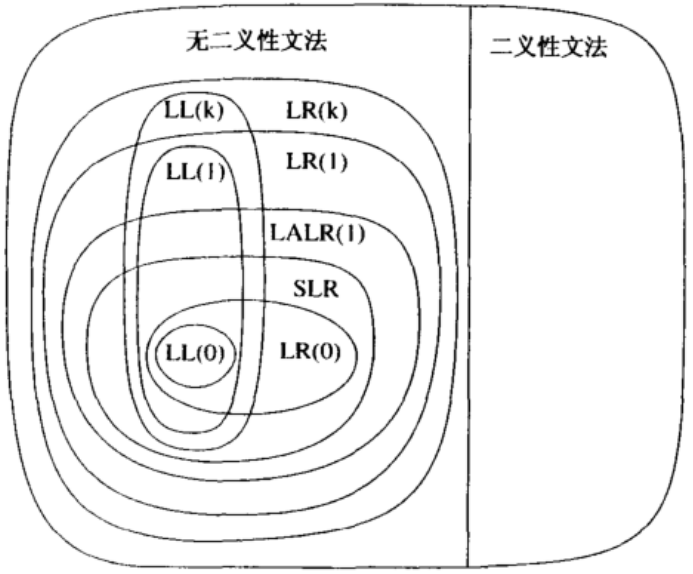


图 3-12 各类文法的层次

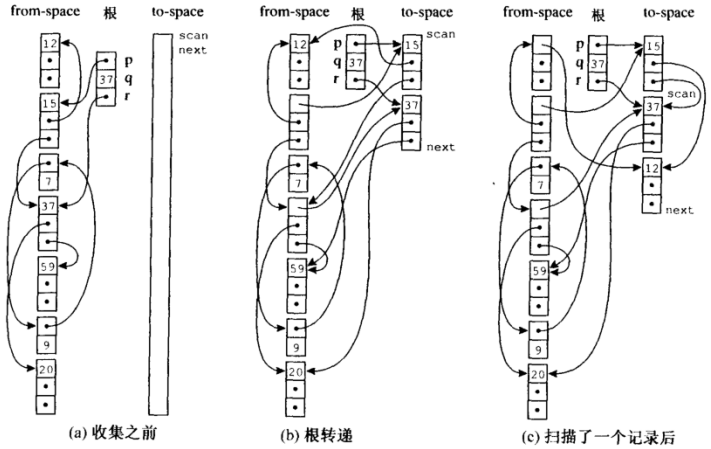


图 13-4　宽度优先复制式收集

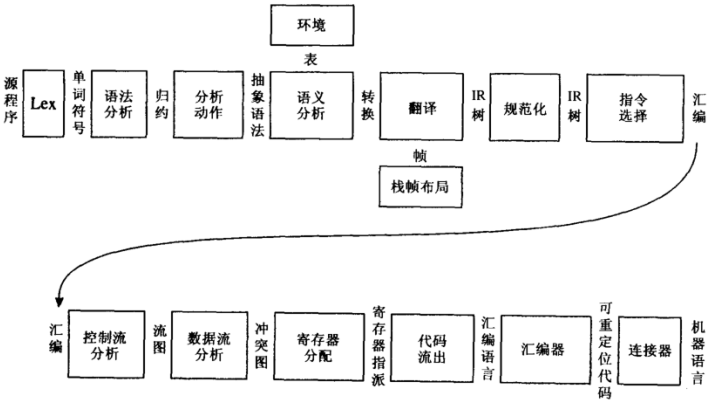


图 1-1　编译器的各个阶段以及它们之间的接口

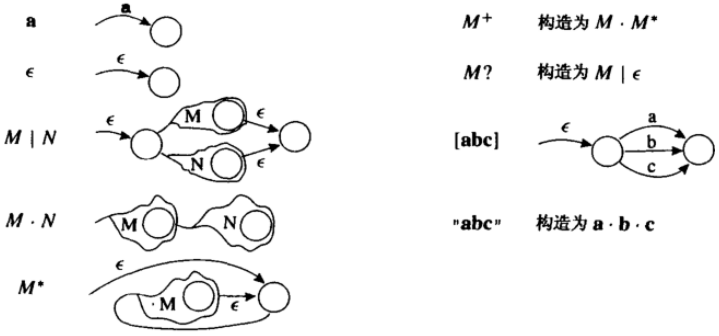


图 2-6　正则表达式至 NFA 的转换

LR(0)例子

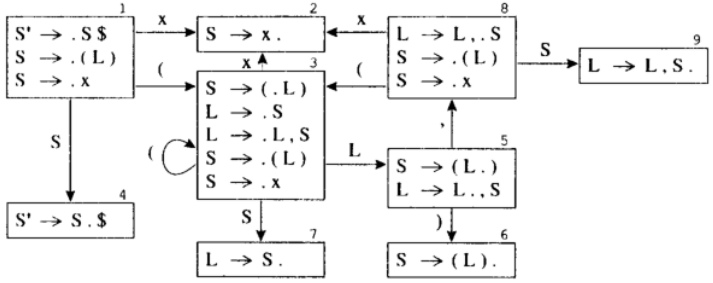


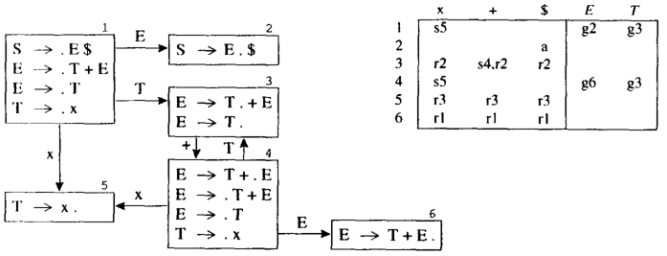
图 3-8　文法 3-8 的 LR(0)状态

分析表

	(	)	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		

文法 3-9

0 $S \rightarrow E\$$	2 $E \rightarrow T$
1 $E \rightarrow T + E$	3 $T \rightarrow x$



	x	+	$	E	T
1	s5			g2	g3
2			a		
3	r2	s4,r2	r2		
4	s5			g6	g3
5	r3	r3	r3		
6	r1	r1	r1		

图 3-9　文法 3-9 的 LR(0)状态和语法分析表

SLR 分析器

仅在 follow 集合指定的位置放置规约

	x	+	$	E	T
1	s5			g2	g3
2			a		
3		s4	r2		
4	s5			g6	g3
5		r3	r3		
6			r1		

图 3-10　文法 3-9 的 SLR 分析表

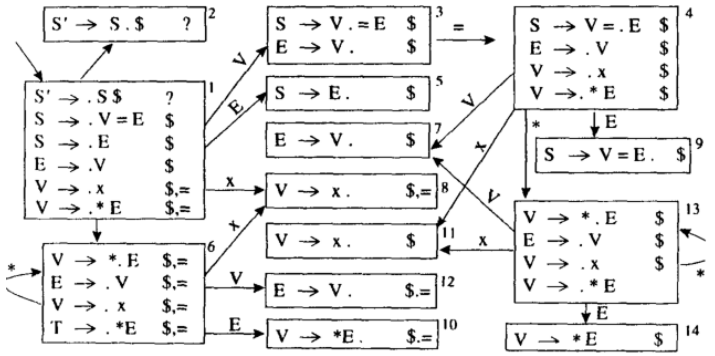


图 3-11　文法 3-10 的 LR(1)状态

	x	*	=	$	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			

(a) LR(1)

	x	*	=	$	S	E	V
1	s8	s6			g2	g5	g3
2				a			
3			s4	r3			
4	s8	s6				g9	g7
5				r2			
6	s8	s6				g10	g7
7			r3	r3			
8			r4	r4			
9				r1			
10			r5	r5			

(b) LALR(1)