编译原理期末技能

正则表达式

- 简单分析
  - 多行注释： `/*"([^\*]|(\*)*[^\*/])*(\*)*"*/`
  - 相邻元素不同的串：
    no0-8⟶9
    no0-7⟶(8|ϵ)(no0-8 8)*(no0-8|ϵ)
    .....
    ans⟶(0|ϵ)(no0 0)*(no0|ϵ)
- 正则表达式 ⟶ NFA
- NFA ⟶ DFA
- DFA最小化

文法

- 证明文法G生成语言L：归纳法证明L(G)⊆L、L⊆L(G)
- 消除二义性
  - 没有规律可循，If 的例子：
    IF⟶**matched** | **open**
    matched⟶if **exp** then **matched** else **matched** | **other**
    open⟶if **exp** then **matched** else **open** | if **exp** then **IF**
- 特殊文法设计
  - 不符合AA形式的串：
    S⟶AB|BA
    A⟶1|0A0|0A1|1A0|1A1
    B⟶0|0B0|0B1|1B0|1B1

LL分析

- 消除左递归、提取左公因子
- 求First集、Follow集
- 填写LL分析表

- a∈First(A) 则 M[A,a]=A→a...
- ϵ ∈First(A)且b∈Follow(A) 则 M[A,b]=A→ ϵ

LR分析

- 识别句柄
  可以直接规约的、最左边的串
- 包含关系：LR(0)⊂SLR⊂LALR⊂LR(1)
  判断方法：
  - 构造LR(0)自动机，无冲突则为LR(0)文法，有冲突但可通过Follow集合解决则为SLR文法
  - 构造LR(1)自动机，无冲突则为LR(1)文法，合并同心集后仍无冲突则为LALR文法

LR(0)分析

- 构造LR(0)自动机
- 填写LR(0)分析表
  - 先给所有语句标号（包括增广句）、求所有First、Follow集
  - 状态0→状态2的转移条件为终结符a，则ACTION[0,a]里填状态转移S_2
  - 状态0→状态1的转移条件为非终结符S，则GOTO[0,S]里填状态号1
  - 状态2中有一条A→ ϵ(是4号语句)，则对于所有终结符b，ACTION[2,b]里填规约r_4
- 进行LR分析
  - 初始栈里为{0}，规约进度为{}，未输入串为{a_1, a_2, ...a_n}
  - 栈顶为0，未输入串的第一项为a。若ACTION[0,a]为移进S_2，则栈顶加入2，规约进度加入a，输入串去掉a
  - 栈顶为6，未输入串的第一项为$。若ACTION[6,$]为规约r_5，5号语句为A→aAB，则栈顶弹出3次，规约进度里替换aAB为A。观察现在的栈顶为2，则寻找GOTO[2,A]=4，则栈顶加入4
- 分析冲突
  表格的同一格内有S和r则为移进-规约冲突，同一格内有两个r则为规约-规约冲突

SLR分析

- 填写SLR分析表
  和LR(0)一样，但：
  - 状态2中有一条A→ ϵ(是4号语句)，则对于所有b∈Follow(A)，ACTION[2,b]里填规约r_4

LR(1)分析

- 构造LR(1)自动机
  现在，每一条语句后面都带有一些非终结符，称为搜索符号串
  - 初始状态下，生成项S'$\rightarrow$.S,$
  - 从项S'$\rightarrow$.AB，α 生成新项A'$\rightarrow$.d时，设置其搜索符号串为串“Bα”的First集的所有元素
- 填写LR(1)分析表
  和LR(0)一样，但：
  - 状态2中有一条A$\rightarrow \epsilon$，α (是4号语句)，则对于所有b$\in\alpha$，ACTION[2,b]里填规约r_4

LALR分析

- 同心集
  两个状态的语句集合相同，搜索符号串不同
- 构造LALR自动机
  和LR(1)一样，但要合并同心集

SDD

- 规则使用结束后执行操作
  - 综合属性：L$\rightarrow$En：L.val = E.val
  - 继承属性：E$\rightarrow$TE'：E'.inh = T.val, E.val = E'.syn
- 对于产生式A$\rightarrow$BCD的规则A.s=D.i, B.i=A.s+C.s, C.i=B.s, D.i=B.i+C.i提问：
  - 是否满足S属性要求？不满足，因为用到了继承属性
  - 是否满足L属性要求？不满足，因为B.i依赖在其右侧的C.s
  - 是否存在和这些规则一致的求值过程？不存在，因为依赖关系中有环
- 设计SDD

SDT

- 规则使用到一半也能执行操作
- 先写出SDD，然后把SDD的每条操作塞到语句的特定位置，就得到了SDT
  $S \rightarrow if(C)S_1elseS_2$
  $S \rightarrow if($ {L_1=newlabel(), C.false=L_1}
  $C)$ {S_1.next=S.next}
  S_1else {}
  S_2 {S.code=C.code || S_1.code || label(L_1) || S_2.code}

- 后缀SDT
  将语句中所有非终结符依次入栈，翻译操作中将每个符号替换。例如S替换为stack[top-3]

递归下降翻译L属性SDT

- SDT构造完成后，将其写成代码
- 直接打印代码的方案
  void S(label next){
  label L_1
  if(当前输入=词法单元if){
  读取输入
  检查"（"是下一个输入符号，读取输入
  L_1=newlabel()
  C(L_1,S_1起始位置)
  检查"）"是下一个输入符号，读取输入
  S(next)
  检查"else"是下一个输入符号，读取输入
  print("label", L_1)
  S(next)
  } }
- 将代码储存在字符串的方案
  string S(label next){
  string Ccode,S1code,S2code
  label L_1
  if(当前输入=词法单元if){
  读取输入
  检查"（"是下一个输入符号，读取输入
  L_1=newlabel()
  Ccode=C(L_1,S_1起始位置)
  检查"）"是下一个输入符号，读取输入
  S1code=S(next)
  检查"else"是下一个输入符号，读取输入
  S2code=S(next)
  return (Ccode || S1code || label(L_1) || S2code);
  } }

自底向上翻译L属性SDT

- 所有元素依次入栈
  语句右侧所有非终结符C前都需要加一个无效果的非终结符P,C要储存自己的代码，P要储存C用到的所有标签
  $S \rightarrow if(PC)QS_1elseRS_2$
  $P \rightarrow \epsilon, Q \rightarrow \epsilon, R \rightarrow \epsilon,$
- 填写栈
  ? {S.next}
  if {}
  ({}
  P {C.false, C.true,L_1}
  C {C.code}
) {}
  Q {S1.next}
  S1 {S1.code}
  else {}
  R {S2.next}
  S2 {S2.code}
- 翻译为代码
  P:
  L_1=newlabel()
  C.false=L_1
  C.true=S1起始位置
  Q:
  S1.next=stack[top-7].next
  R:
  S2.next=stack[top-10].next
  S:
  tempCode=stack[top-6].code || stack[top-3] || label(L_1) || stack[top-7].code || stack[top].code
  top=top-10
  stack[top].code=tempCode

中间代码表示

- 有向无环图DAG：由抽象语法树AST提取公共子表达式得到

- 三地址代码
  遇到数组，需要用临时变量计算地址。数组运算符号[]、[]=
  - 常用的是四元式：(op,arg1,arg2,result)
    + b c t_1
    - 0 t_1 t_2
    * a t_2 t_3
  - 三元式：(op,arg1,arg2)
    (0) + b c
    (1) - 0 (0)
    (2) * a (1)
  - 间接三元式：instruction，(op,arg1,arg2)
    (0) + b c
    (1) - 0 (0)
    (2) * a (1)
    instruction：
    (33) (0)
    (34) (1)
    (35) (2)
- 记录的空间分配
  大小为2^t的类型需对齐到2^t的整数倍

中间代码生成

- 分支语句翻译
  会有一堆goto
  S$\rightarrow$for(S_1,B,S_2)S_3
  .....
  S.code=S_1.code || label(begin) || B.code || label(B.true) || S_3.code || label(S_3.next) || S_2.code || goto(begin)
- 回填
  为什么需要回填技术?
  有些语句的跳转位置在未翻译完成的代码块之后，故翻译该语句时无法确定其跳转位置，需要在其翻译完成后回填跳转位置，避免了第二趟处理
  将跳转位置的标签储存在一个表中，等到翻译完成后，将表中的所有标签都填上正确的位置。表之间可以合并。
  $S_7:if(B_1)S_1elseS_2$
  $S_7.nextlist = merge(B_1.falselist, S_1.nextlist, S_2.nextlist)$

栈式存储管理

- 活动树
  函数调用即为分支，非常简单
- 活动记录
  参数、函数调用、临时变量
- 变量作用域
  经典大坑：传值 OR 传指针?
  int f(int x, int* y, int** z)
  { **z+=1; *y+=2; x+=3; return x+*y+**z; }
  a→b, b→c, int c=4, 则调用f(c,b,a)返回值为21，而非30
  因为函数中的x是临时变量，运行到x+=3时x仍然为4，这一步的3只加到了x身上
- 填写访问链
  嵌套深度：main函数为1，在嵌套深度为i的过程a中定义了过程b，则b的嵌套深度为i+1
  当嵌套深度为m的过程p调用嵌套深度为n的过程q时，如果：
  - m<n，则q的访问链指向p
  - m≥n，则追踪p的访问链m-n+1步到r，q的访问链指向r

垃圾回收

- 引用计数：有多少个元素指向它
  引用计数为0时，删除该元素
- 遍历法：DFS遍历一遍，没遍历到的就删除

寄存器分配

- 转化为三地址代码
  对于语句x = y op z，需翻译为：
  LD L_1, y
  LD L_2, z
  op L_3, L_1, L_2
  (若x不是临时变量) ST x, L_3
- 基本块划分和流图构造
  确定首语句。首语句到下一个首语句之前为一个基本块。以下是首语句：
  - 第一条三地址代码
  - 跳转语句的目标语句
  - 被跳转语句的下一条语句

- 活跃变量分析
  - 代码块B从其下一个代码块获取活跃变量集合α
  - 从B的最后一条语句开始向前扫描，扫描到语句x = y op z时，先从α中去掉x，再将y和z加入α
  - 所有语句的最大的活跃变量集合大小为k，则至少需要k个寄存器才不会溢出
- 寄存器冲突图
  进行活跃寄存器分析（和活跃变量分析步骤一样），画出冲突图，尝试x-着色，求出x的最小值。则至少需要x个寄存器才不会溢出
- 寄存器分配过程分析
  以每条源代码为单位，写出其对应的三地址代码和一个表，表头包含所有寄存器、变量和临时变量
  - 寄存器表项：所放的变量名
  - 变量表项：被放入的寄存器名（无则不填）、本体名称
  - 临时变量表项：被放入的寄存器名（无则不填）
- 溢出
  - 尝试去除一个节点使得图可以(x-1)-着色。被去掉的寄存器需放在栈上
  - 使用之前，将其Load出来
    LD L_1, 4(sp)
    SUB R_1, R_2, R_1
  - 如果它是运行结果，使用完后还需要将其Store回去
    ADD R_1, R_2, R_1
    ST 4(sp), R_1
  - 两个操作都会使用一个新的寄存器，这会形成新的寄存器冲突图，该图可能仍然无法(x-1)-着色

代码优化

- 简单优化
  - 公共子表达式消除：两个表达式一样，却赋值到不同的临时变量t_1、t_2，则所有t_2可改为t_1
  - 复写传播：x = y 为复写语句，此后若y不再被使用，则所有y可改为x
  - 死代码消除：某语句的运行结果不会被使用，则可删除
  - 常量传播：某个变量的值可以确定，则所有该变量可改为该值
  - 代码外提：某语句无论循环多少次都不会改变结果，则可以提到循环外面
  - 归纳变量消除：某变量的每次循环变化量固定，则可以用加减法运算替换乘法运算
- 窥孔优化
  用更高效的指令替换窗口中的指令序列

数据流分析

- def 和 use
  对于语句x = y op z，其def集合为{x}，use集合为{y,z}
  对于代码块B，需从下到上分析：对于每条语句a，defB+=defa再-=usea，useB-=defa再+=usea，最终得到了B的def和use集合。
- IN 和 OUT
  EXIT的IN集合为ϕ，ENTRY的OUT集合为ϕ
  从下向上分析，每个块B的OUT集合为其后继所有块的IN集合的并集，IN集合为OUT-=defB再+=useB。遇到循环时，需不断迭代直到IN和OUT不再变化。