编译原理复习大纲

Pre

1. 高级语言-汇编语言-机器语言关系
2. 编译：高级语言->汇编语言；高级语言->机器语言，即将高级语言（源语言）翻译为汇编或机器语言（目标语言）的过程
3. 源代码到机器代码的过程：
    - 预处理（宏定义处理，展开）
    - 编译
    - 汇编（生成可重定位的机器码）
    - 链接与加载（库、外部地址的链接，程序在内存的装载）
4. 编译过程：（1-4前端-源语言， 6-7后端-目标语言，234合称语法制导翻译）
    1. 词法分析——确定每个单词类型
    2. 语法分析——识别短语，构造语法分析树（标准合法）
    3. 语义分析——收集属性（用于运算），语义检查（逻辑合法）
    4. 生成中间代码
    5. 机器无关代码优化
    6. 生成目标代码
    7. 机器相关代码优化
5. 三地址码
    - 每个指令最多由三个操作数构成（类似于汇编）

词法分析

基本概念

1. 字母表（$\sum$）——串同理：
    1. 乘积（笛卡尔积）
    2. 幂运算（每位展开，长度为n），$\sum^0 = e$
    3. 正闭包（不包含空的n次幂求并）
    4. 克林闭包（正闭包+空）
2. 文法
    1. $G(V_t, V_N, P, S)$
        1. V_t：终结符（小写字母，斜体，小写希腊字母，空）
        2. V_N：非终结符（大写）
        3. P：产生式$\alpha \to \beta$（左部，右部）
        4. S：开始符号（特殊非终结符）
    2. 推导（带入过程），$ab =>^n cb$（n也可为+, *）
    3. 规约（推导逆过程）
    4. 分类

1. 0型（无限制PSG）
    - 左部至少包含一个非终结符
2. 1型（上下文有关CSG）
    - $aAb->aBb$（B不为空，CSG不含有空产生式），需要被夹起来
3. 2型（上下文无关CFG）
    - 左部为终结符
4. 3型（正则RG）
    - 右（左）线性文法：$A->bC, A->b(A->Cb, A->b)$

3. CFG分析树——每一个句型对应一棵语法分析树（并不唯一）
    1. 根节点为文法开始符号
    2. 内部节点为一个产生式（父亲为左部，儿子为右部）
    3. 不需要展开至终结符
    4. 叶子节点从左到右为树的产出（边缘）——一部分则成为原句型的一个短语
    5. 直接短语：分析树中，某个高度为2的子树的边缘
4. 正则表达式（利用$|, *, ()$连接）——正则文法与正则集合相互对应
5. 有穷自动机（FA）
    1. 构成：离散的输入输出+当前内部状态
    2. 一个初始状态+多个终止状态，状态间用有向边连接
    3. FA定义的语言：输入串从start到某个end终止状态停止，则称串被接受，所有被接受的串的集合即为FA定义的语言
    4. 最长子串匹配原则

正则到DFA

1. 正则到NFA
    - 从start开始，向每个新状态连有向边，需要注意，克林闭包为自环
2. NFA到DFA
    - DFA中的每个状态，为NFA中若干个状态的额合集（同一个状态，经过一条边能到达的所有状态合并为一个集合）
    - 对于边权为空的边，等价与所有被空相互链接的状态可以直接到达，因此也合并为一个状态

语法分析

自顶向下

- 对于给定串，从文法start（分析树根节点）推导（向叶子节点），构建语法分析树

最左（右）推导

- 每次选择当前句型的最左（右）侧非终结符进行推导（最大前（后）缀）
- 自顶向下采用最左推导：每次选择最左侧的非终结符进行替换

最左（右）规约

- 上述过程的反过程（从叶子向根）

匹配过程（递归下降分析）

1. 从start（S）开始，不断逐层展开其产生式，在匹配时，使用最左匹配原则（从左至右展开），直到当前待匹配位置被展开（出现）
2. 将匹配指针后移，继续重复1操作继续展开
3. 为保证最终叶子从左至右为待匹配举行，将未出现的位置展开为空（若不存在空产生式则该串不可被接受）

- 若同一非终结符由多个相同前缀的候选式时，则无法确定当前匹配位置与同一非终结符的哪个候选式匹配，此时就需尝试每个具有与匹配位置终结符相同前缀的候选式，此时就会存在回溯行为，具有回溯行为的分析器具有不确定性，虽能保证正确性，但分析效率不高

存在的问题

1. 回溯降低效率——提取左公因子，将前缀相同的合并，定义一个新的产生式，仅包含当前待匹配字符（将文法进行改造，$S->abc|adb$改造为$S'->aS$，$S->bc|db$）
2. 左递归文法产生死循环
    - 直接左递归文法：形如$E->ET|T$，此时会不断展开$E->E....ET$
    - 间接左递归文法：推导一定步数后得到直接左递归文法的局面
    - 处理方式：令$E->TE'$，$E'->TE'|e$

预测分析

- 在递归下降分析时，向前看固定个数符号，以确定唯一非终结符进行展开

S_文法

- 每个产生式右部都由终结符开始
- 同一非终结符的每个候选式没有公共前缀
- 不包含空产生式

FIRST集

- 当前非终结符可以推导出的所有串首终结符的集合（左向右依赖）
- 构造过程：
    1. 若当前非终结符的产生式右部存在前缀为终结符，则其FIRST为该产生式右部的首终结符（首位）
    2. 若当前非终结符存在空产生式（或可推导出空），则将其加入FIRST集中
    3. 若当前非终结符的产生式其右部为非终结符，则将其右部的开头非终结符的FIRST加入当前非终结符FIRST中
    4. 在3基础上，若其右部刚刚加入的当前FIRST的非终结符的FIRST中含有空串，则继续加入其后字符的FIRST，直到当前字符为终结符或为非终结符且不可推导为空，若右部都由非终结符构成，且每个FIRST都含有空串，则最后将空串加入当前非终结符的FIRST中
    5. 不断重复以上，直到当前轮没有任何新终结符、空被加入任何FIRST中
- 求某串的FIRST同理于34

FOLLOW集

- 可能在某句型后紧跟的终结符的集合，定义`$`为某个句型的最右符号（右向左依赖）
- 构造过程：
    1. 对于start和能推导出空的非终结符，将`$`加入其中
    2. 对于右部形如AB，根据定义，则将A的FOLLOW（除空）加入B的FIRST
    3. 对于每个产生式，可以紧跟左部的符号，也应可以紧跟右部，因此将左部的FOLLOW加入右部末尾符号的FOLLOW
    4. 在3之上，若右部末尾符号可推导出空，则右部末尾符号之前的符号也可与左部末尾可跟符号一致，因此将左部FOLLOW加入右部末尾符号之前符号FOLLOW，直到当前符号为终结符或不可推导为空
- 求某串的FIRST同理于34

SELECT集

- 对于一产生式，可以使用其进行推导的当前输入符号的集合（当待匹配的符号在集合中时，就可以使用当前产生式），其逆向上对于当前待匹配符号，可以直接选用的产生式
- 构造过程：
    1. 若某产生式右部存在前缀为终结符构成的串，则其SELECT为该产生式右部的首字母
    2. 反之，则开头为非终结符，此产生式SELECT为该非终结符所能推导出的所有终结符（当其可推导为空时，根据SELECT定义，就需要继续往后看一位符号，直到当前符号为终结符或为非终结符且不能推导出空）
    3. 若为空产生式，则其SELECT为左部的FOLLOW集
- 利用FIRST集，求SELECT集：
    1. 产生式右部的FIRST不包含空，则产生式的SELECT就为右部可以推导出的所有串，即右部的FIRST
    2. 反之，则右部不仅包含右部FIRST还包含左部FOLLOW

Q_文法

- 每个产生式的右部为空，或存在前缀为终结符构成的串
- 具有相同左部的产生式的所有可选集，内有公共前缀
- 不允许右部前缀为非终结符

LL（1）文法

- 对于某两个具有相同左部的产生式：其右部（a，b）满足（具有相同左部的产生式，SELECT不相交）
    - a，b的FIRST之多有一个包含空（若都包含空，则其SELECT都包含右部FOLLOW）
    - 若a，b的FIRST均不包含空，则a，b的FIRST没有交集
    - 若有包含空，设a的FIRST包含空，则a的FIRST与共同左部的FOLLOW交集为空

LL（1）递归分析

1. 为每一个产生式的右部，定义递归下降分析函数
2. 每个递归下降分析函数，根据当前串的每个符号：
    1. 为终结符，对当前符号直接进行判断，若不符则报错
    2. 为非终结符，进入此非终结符的递归下降分析函数

LL（1）非递归分析（表驱动）

- 下推自动机：利用一个栈记录当前的已经读入的状态内容（下推存储器）
- 预测分析表：
    - 针对每一个产生式的左部，对下一输入符号内容进行讨论
    - 当下一输入符号在当前产生式左部的SELECT中，则利用当前产生式进行推导
- 过程：
    1. 初始下，栈中包含开始符号与`$`
    2. 根据当前栈顶字符和下一输入字符，从预测分析表中选择对应的产生式进行推导（若不存在则报错），推导式倒序进栈
    3. 若栈顶字符和下一输入字符相同，则指针后移，栈顶出栈，表明当前字符匹配成功

错误分析

可能的错误

1. 当前栈顶为终结符，与下一输入符号不匹配
2. 当前栈顶为非终结符，下一输入符号，在预测分析表中为空

恐慌模式

- 同步词法单元：针对当前非终结符，若下一输入字符不能直接用于推导，但仍与当前非终结符存在后继关系，则应先报错，后将栈顶弹出，继续尝试与之后内容进行匹配。
- 将每个左部的FOLLOW集中元素，作为SYNCH
- 当出现错误时（先报错）
    1. 若当前表项为空，指针指向下一字符
    2. 若当前表项为SYNCH，则弹出栈顶，继续分析
    3. 若当前表项为终结符且与下一输入字符不匹配，则将栈顶弹出，继续分析

自底向上

- 对于给定串，从分析数叶子向顶部方向构建语法分析树，即将输入串向开始符号规约的过程。关键在于如何正确识别句柄

最左（右）规约

- 每次选择当前栈内串的最左（右）侧非终结符进行规约（最大前（后）缀）
- 自底向上采用最右规约，规约使用的句柄为当前分析树的最左直接短语

LR分析

基本概念

- L：从左向右扫描；R：反向构造一个最右推导序列，K：向前查看的字符数
- 状态：移进状态——待规约状态——规约状态
- LR分析器：栈内存两维信息（状态，符号）
- 预测分析表：
  - ACTION（针对下一状态为终结符）：S（移入），R（规约）
  - GOTO（针对当前栈顶为非终结符）：当前状态下，若栈顶为非终结符，则当前状态进行转移（每次规约结束后可能栈顶为非终结符，因此需要状态转移）
- 分析过程：
  1. 初始状态下，栈内为`0、$`，输入指针指向第一个字符
  2. 针对栈顶元素与下一输入字符，查表决定是规约还是移入操作
  3. 若2结束为规约操作，且栈顶为非终结符，则查询当前状态GOTO表，跳转至对应状态
  4. 反复操作直到下一操作AC

LR（0）——能规约就规约

- 增广文法：开一个新产生式，让$S'->S$，保证入口出口唯一
- 项目：
  - 定义：每一个右部长度为n的产生式可以生成n+1个项目，一个项目的后继项目，即将当前项目的点后移一个单位
  - 等价项目（项目的合并）：将等待的符号推导后得到相同结果的项目，将所有等价的项目合并为一个项目，即为一个项目集（闭包），对应与自动机的一个状态
- 构建LR（0）自动机
  1. 初始状态为，Start的所有等价项目（根据当前等待的字符进行展开）
  2. 每次根据上一状态的每一个项目的等待项，作为一条边连向下一状态，同时等待项目后移
  3. 对于当前状态中规约状态下的项目，当其下一输入为`$`时进行规约，规约后转移至当前产生式状态
- 构造LR（0）分析表
  1. 计算某个项目集的闭包：对于当前集合中的每个项目，根据每个项目的等待符号，利用产生式进行推导（类似于求传递闭包）
  2. 计算某个项目集移入某个字符X的后继项目集闭包：针对项目集中形如$A->a.Xb$的项目，将$A->aX.b$加入后继项目集闭包
  3. 利用12求一文法下的状态集（项集族）：
     1. 初始项目集为$S'->S$
     2. 对于当前项集中的每一个项目，利用上述2求其后继项目集闭包
  4. 根据DFA来建立对应的ACTION和GOTO表即可

SLR——仅对于在FOLLOW下进行规约

- 在LR（0）中出现的冲突中，由于没有结合上下文环境，在规约时会对错误的句柄进行规约，借助FOLLOW集可消除一些错误的规约情况
- 若项目集中满足：所有的移进项目的等待项与所有规约项目的FOLLOW没有交，则对于下一输入符号，若存在于等待项中，则移入，若存在于某个规约项目的FOLLOW集中则规约

LR（1）——考虑当前左部的后继符（后继符是FOLLOW的一个子集）

- 在不同的规约位置，左部会需要不同的后继符号
- 展望符：在LR（0）项目基础上，加入一个当前产生式的当前状态下必须紧跟的终结符（k定义其长度）
- 当规约时，需要当前产生式当前状态下的下k个元素等于当前状态的展望符时，才可进行规约
- LR（1）下的等价项目：
  - 形如$A->a.Bc,x$，则其等价项目为$B->.dyyy,FIRST(cx)$
- 构造过程与LR（0）类似，不同点在于需要继承展望符

语法制导翻译

基本定义

- 对于上下文无关文法（2型文法）,将语法分析+语义分析+中间代码生成一起完成
- 语义属性：
  - 为CFG中的每一个文法符号，设置若干语义属性，语义信息直接进行运算以完成语义分析
- 计算：
  - 对产生式定义语义规则，构建语义分析树后利用产生式相关的语义规则，计算树上各节点的语义属性
- 语法制导定义（SDD）
  - 为每一个文法符号赋予一些语义属性
  - 将每一个产生式与一组语义规则关联，用于计算产生式中各文法符号的属性（用形如$A.a$表示A的属性a）
- 语法制导翻译方案（SDT）
  - 在产生式中嵌入一些语义动作（计算函数），构成的新的形如产生式的文法
  - 语义动作位置决定了语义规则执行的次序（时刻）
- 文法符号的属性：
  - 综合属性：由语法分析树上的该点子节点属性，或本节点属性定义（终结符的属性即本身），值的来源于以该点为根的子树
  - 继承属性：由语法分析树上的该点的父亲，或兄弟，或自身的属性定义，值来源于以除去子节点的其他节点+自己
- 副作用：由特定的产生式，在规约时调用一些函数——不包含副作用的SDD称为属性文法
- 依赖图：
  - 对分析树中每一个节点的求值所依赖节点进行建图（向所依赖的节点连边）
  - 计算所有节点的属性的顺序应为TOP序，因此不应存在环形依赖
- S-SDD

    - 仅使用综合属性所定义的SDD——不存在继承依赖
- L-SDD
    - 可以存在继承依赖，但是所依赖的节点必须在当前节点的额左侧（依赖图中的边均为从左向右）

SDT过程

- 对于LR文法，若满足S-SDD，用自底向上分析
- 对于LL文法，若满足L-SDD，用自顶向下分析

S-SDD

- S-SDD转SDT：
    - 由于不存在继承依赖，将语义规则插入每条产生式末尾即可
- 实现过程（LR分析）
    - 在栈中同时记录当前栈顶文法符号的综合属性（可能有多个）
    - 将语义动作改写为具体的栈操作代码：
        - 规约前栈顶元素依次为当前规约的每个文法符号
        - 因此只需要将规约时所有出栈的文法符号的综合属性进行对应计算，最后得到的属性即为规约后产生式左侧文法符号的综合属性

L-SDD

- L-SDD转SDT：
    - 对于计算综合属性的动作，插入当前产生式的最右端
    - 对于某非终结符计算继承属性的动作，插入产生式右部紧靠当前非终结符之前出现的位置上（就是他之前）

非递归预测分析过程翻译

1. 扩展语法分析栈：
    - 对于继承属性，有应在其出现之前计算，对于综合属性，应在其所有子节点计算完成后最后计算，因此需要将每个文法符号的继承属性和综合属性同时记录
    - 将每个语义动作封装（改写为栈操作），同样记录与栈中
    - 即栈中依次记录（从栈顶向下）：语义动作指针——继承属性——综合属性
2. 每次入栈时，要求当前栈顶元素为非产生式，否则执行3出栈操作
3. 每次出栈时：
    - 出栈元素为综合属性
        - 应将其备份至指定的语义动作位置——综合记录出栈要将当前综合记录赋值给后面的语义动作
    - 变量展开（推导）时：
        - 若被展开的变量具有继承属性，则应将其备份至指定的语义动作位置——继承记录出都要将当前综合记录赋值给后面的语义动作
    - 出栈元素为语义动作：
        - 执行对应的栈操作代码

递归预测分析过程翻译

- 将每一个非终结符展开时，定义为进入一个函数，传入参数为所依赖的继承属性，其返回值为其综合属性
- 在函数内部，为当前产生式的每文法符号设置变量，用于计算最终的综合属性

利用LR分析实现翻译——这玩意之前自己居然发明过？

- 将含有继承属性的产生式进行修改：
  - 将该语法动作提出，构造新的空产生式，在其右部末尾添加该继承属性的语义动作
  - 新产生的左部具有的继承属性与语义动作需要的继承属性相同，其综合属性为原产生式所需计算的继承属性——原先依赖于一个语义动作，现在依赖于他之前的那个我们新加的文法符号的综合属性
- 修改后的文法符合S-SDD，因此可以建立LR自动机：
  - 由于新添加的产生式都为空产生式，因此一旦存在与某状态，则进行规约，实则是进行语义动作的计算
  - 对于语义动作所依赖的属性，不难发现其所依赖的属性就在当前栈顶所指向的位置往下取句柄长度个后对应的位置（逆向）——这个和之前自己发明的LR是一样的
  - 同样，我们也可以对每个语义动作生成对应固定的栈操作代码，这样就不用每次规约时先往前找对应句柄长度个再对号找依赖的属性了

中间代码生成

声明语句

类型表达式T（可套娃）

1. 数组构造符（array）
   - `array(nums, T)`
   - `int [i] --> array(i, int)`
   - `int [i][j] --> array(i, array(j, int))`
2. 指针构造符（pointer）
   - `pointer(T)`
3. 笛卡尔积构造符(X)
   - `T1 X T2`
4. 函数构造符（->）
   - `T1 X T2 ... -> Tk`，函数参数为箭头左侧，返回值为右侧、
5. 记录构造符（record）
   - `record((Name1 X T1) X (Name2 X T2)...)`，字段名X类型

翻译

- 声明语句作用
  - 标识符类型（T）
  - 类型宽度（width）
  - 起始位置的相对地址——编译时分配

- SDT:

```
1. P -> { offset = 0} D
2. D -> T id; {enter（id.lexeme, T.type, offset); offset = offset + T.width}D
3. D -> e
4. T -> B {t = B.type; w = B.width;}C {T.type = C.type; T.width = C.width;}
5. T -> |T1 {T.type = pointer(T1.type; T.width = 4;)}
6. B -> int {B.type = int; B.width = 4}
7. B -> real {B->type = real, B->width = 8}
8. C -> e {C.type = t, C.width = w;}
9. C -> [num]C1 {C.type = array(num, C1.type); C.width = num.val * C1.width}
```

  - B 为基本数据类型，给定了对应初始长度与类型
  - C为空则为变量，C不为空则是数组
  - T 为一条声明语句，也可为指针
  - D为声明序列
  - offset为下一可用地址
  - t，w从语法分析树中的B节点传递至C为空的节点——B是变量类型的申明
- 满足LL文法，因此可以使用表驱动自底向上来构建（求SELECT再推导）

基本赋值语句

- 基本语法

```
S -> id = E;
E -> E + E
E -> E * E
E -> E -> -E
E -> (E)
E -> id
```

翻译

- 赋值语句翻译的主要任务：
  - 借助中间变量，生成表达式求职的三地址码（基本操作原子）
- SDT：

```
1. S -> id = E {p = lookup(id.lexeme); if p = nil then error; S.code = E.code
|| gen(p = E.addr)}
2. E -> E1 + E2 {E.addr = newtemp(); E.code = E1.code || E2.code ||
gen(E.addr = E1.addr + E2.addr)}
3. E -> E1 + E2 {E.addr = newtemp(); E.code = E1.code || E2.code ||
gen(E.addr = E1.addr * E2.addr)}
4. E -> -E1 {E.addr = newtemp(); E.code = E1.code || gen(E.addr =
uminusE1.addr)}
5. E -> (E1) {E.addr = E1.addr, E.code = E1.code}
6. E -> id {E->addr = lookup(id.lexeme); if E.addr = nil then error; E.code =
'';}
```

  - lookup返回符号表中的记录，nil为空

    - newtemp生成新的临时变量
    - Code为三地址指令所串联的字符串，||表示将指令串联
    - gen生成对应的三地址指令
    - 不难发现上述Code的每次的复制是冗余的，可以每次再上次生成的Code上新增需要的三地址指令即可：
        - 因此可以将gen改为生成新的三地址指令后，将新的指令拼接至之前生成的指令之后
        - 此时就可以将所有的文法符号的Code属性删除
- 由于存在左递归，因此采用LR来翻译，构造LR自动机即可

数组赋值语句

- 数组引用翻译时，关键在于求出当前引用位置的下标（偏移地址）
- 数组元素寻址：
    - $add = base + w_k * idx_k + \ldots + w_m * idx_m)$
    - wi分别对应数组k维下的单位元素宽度
    - 在计算时，先计算每个乘号下的内容，当可以存在可以相加的两项时，计算加法，重复以上操作，最后再加上base
- SDT：

```
1. S -> id = E | L = E {gen(L.array[L.offset] = E.addr);}
2. E -> E1 + E2 | -E1 | (E1) | id | L {E.addr = newtemp(); gen(E.addr =
L.array[L.offset]);}
3. L -> id[E]{L.array = lookup(id.lexeme);
            if L.arrray == nil then error;
            L.type = L.array.type.elem;
            L.offset = newtemp();
            gen(L.offset = E.addr * L.type.Width)}
        | L1[E1] {L.array = L1.array;
            L.type = L.type.elem;
            t = newtemp();
            gen(t = E.addr * L.type.width; L.offset = newtemp();
            gen(L.offset = L1.offset + t)}
```

    - id读入对应的数组名称后，L.array即为该数组的起始位置，后获取其type与之后[E]生成三地址指令，并更新offset地址
    - 每次新节点的type由其儿子传递
- 由于满足SSDD，利用LL分析即可

控制流语句

- 对于不同分支，需要确定流向：
    - Next：表示当当前控制流语句结束后执行的下一代码位置
    - True：表示当控制流bool为真时转向的指令位置
    - False：与True同理
- 顺序语句的SDT

```
1. P -> {S.next = newlabel();} S (label(S.next))
2. S -> {S1.next = newlabel();} S1{(Label(S1.next); S2.next = S,next);} S2
3. S -> id = E; | L = E;
```

- newlabel为生成一个用于存放代码下标的临时变量（由于S未计算，其代码长度未知，因此S.next需要等其分析完毕）
- label将下一条三地址指令的额标号赋予传入的变量

- `if-else-then`的SDT

```
S -> if {B.true = newlabel(); B.false = newlabel();} B
    then {label(B.true; S1.next = S.next)} S1 {gen(goto S.next)}
    else {label(B.false); S2.next = S.next} S2 {gen(goto S.next)}
```

- `if-then` 的SDT

```
S -> if {B.true = newlabel(); B.false = S.next} B
    then {label(B.true); S1.next = S.next} S1
```

- `while-do`的DST

```
S -> while {S.begin = newlabel(); lable(S.begin) ;B.true = newlable();
B.false = S.next} B
    do {label(B.true); S1.next = S.begin() } S1 {gen(goto S.begin())}
```

- `switch-case-default`的DST
  - 将每个next指向下一个case的判断

```
Switch E {t = newtemp(); gen(t = E.addr);}
CaseV1:  {L1 = newlabel(), gen(if t != V1 goto L1);}
    S1 {next = newlabel(); gen(goto next);}
CaseV2:  {L2 = newlabel(), gen(if t != V2 goto L2);}
    S2 {gen(goto next);}
...
CaseVn:  {Ln = newlabel(), gen(if t != Vn goto Ln);}
    Sn {gen(goto next);}
Default: {label(Ln);}
    Sm {label(next);}
```

  - 将所有的判断集中与代码最后（写成一个test代码段）

```
Switch {t = newtemp(); gen(t = E.addr); test = newlabel(); gen(goto test)}
CaseV1: {L1 = newlabel(); label(L1); map(V1, L1);}
    S1 {next = newlabel(); gen(goto next)};
CaseV2: {L2 = newlabel(); label(L2); map(V2, L2);}
    S2 {gen(goto next)};
...
CaseVn: {Ln = newlabel(); label(Ln); map(Vn, Ln);}
    Sn {gen(goto next)};
Defaule: {Lm = newlable; label(lm);}
    Sm {{gen(goto next);
```

```
            label(test);
            gen(if t = V1, goto L1);
            ...
            gen(if t = Vn, goto Ln);
            gen(goto Lm);
            label(next);
        }
```

BOOL表达式

- 逻辑运算：`not, and, or`（优先级递减），`E relop E, true, false`
- 逻辑运算符在翻译时，被翻译为跳转指令，并非出现在代码中（true进入，false逃过）
- SDT

```
1. B -> E1 relop E2 {gen(if E1.addr relop E2.addr goto B.true); gen(goto
B.false)}
2. B -> true {gen(goto B.true)}
3. B -> false {gen(goto B.false)}
4. B -> ({B1.true = B.true, B1.false = B.false}) B1
5. B -> not {B1.true = B.false; B1.false = B.true} B1
6. B -> {B1.true = B.true; B1. false = newlabel();}B1 or {label(B1.false);
B2.true = B1.true; B2.false = B.false;} B2
7. B -> {B1.true = newlable(); B1.false = B.false} B1 and {lable(B1.true);
B2.true = B.true; B2.false = B.false;} B2
```

过程调用翻译

- 即函数调用
- 利用队列将所有传入的形参入队，最后生成param t语句，call语句为函数地址+参数个数（向上数n条语句为参数定义）

```
S -> call id(Elist)
{
    cnt = 0;
    for t in q do
    {
        gen(param t);
        n = n + 1;
    }
    gen(call id.addr, n);
}
Elist -> E
{
    q init为空，将E.addr入队
}
Elist -> Elist, E
{
    将E.addr入队
}
```

语句回填

- 在生成跳转指令时，使用了newlabel来预留位置，将目标地址标号作为继承属性传递至跳转指令生成位置，最后再将需要跳转的位置的对应标号与具体地址绑定
- 回填：
  - 在生成跳转指令时，先不指定跳转指令的目标标号，将指令放入由跳转指令组成的列表（同列表的指令标号相同），作为综合属性传递，待正确标号确定后，再统一回填
- makelist（i）
  - 创建一个新列表，加入i，i为跳转指令的标号，返回列表指针
- merge（p1， p2）
  - 将p1，p2列表合并，返回新列表
- backpatch（p，i）
  - 将目标标号i回填p列表的指令中

BOOL表达式回填

- True（false）list：
  - 表示所有包含需要跳转至B为True（false）时控制流该转向的指令标号

回填操作

1. `B -> E1 relop E2`

```
{
    B.truelist = makelist(nextquad);
    B.falselist = makelist(nextquad + 1);
    gen(if E1.addr relop E2.addr goto_);
    gen(goto_);
}
```

2. `B -> true`

```
{
    B.truelist = makelist(nextquad);
    gen(goto_);
}
```

3. `B -> false`

```
{
    B.falselist = makelist(nextquad);
    gen(goto_);
}
```

4. `B -> (B1)`

```
{
    B.truelist = B1.truelist
    B.falselist = B1.falselist
}
```

5. `B -> not B1`

```
{
    B.truelist = B1.falselist
    B.falselist = B1.truelist
}
```

6. `B -> B1 or B2`

```
B -> B1 or MB2
{
    backpatch(B1.falselist, M.code);
    B.truelist = merge(B1.truelist, B2.truelist);
    B.falselist = B2.falselist
}
M -> e {M.code = nextquad;}
```

7. `B -> B1 and B2`

```
B -> B1 and MB2
{
    backpatch(B1.truelist, M.code);
    B.falselist = merge(B1.falselist, B2.falselist);
    B.truelist = B2,truelist
}
M -> e {M.code = nextquad}
```

控制流指令回调

- nextlist:
  - 表示包含该语句顺序紧跟再其后的控制流该转向的指令标号
- `S -> if B then S1`:

```
S -> if B then MS1
{
    backpatch(B.truelist, M.code);
    S.nextlist = merge(B.falselist, S1.nextlist)
}
```

- `S -> if B then S1 else S2`:

```
S -> if B then M1 S1 N else M2 S2
{
    backpatch(B.truelist, M1.code);
    backpatch(B.falselist, M2.code);
    S1.nextlist = makelist(nextquad);
    S.nextlist = merge(merge(S1.nextlist, S2.nextlist), N.nextlist);
}
N -> e {N.nextlist = makelist(nextquad); gen(goto_);}
```

- `S -> while B do S1`

```
S -> while M1 B do M2 S1
{
    backpatch(S1.nextlist, M1.code);
    backpatch(B.truelist, M2.code);
    S.nextlist = B.falselist;
    gen(goto M.code);
}
```

- `S -> S1 S2`

```
S -> S1 MS2
{
    backpatch(S1.nextlist, M.code);
    S.nextlist = S2.nextlist;
}
```

- `S -> id = E ; | L = E; {S.nextlist = null}`