杭州电子科技大学
HANGZHOU DIANZI UNIVERSITY

《编译原理》实验指导书

适用专业：计算机各专业

编 制 人：谌志群、王小华

计 算 机 学 院

2009 年 6 月

前　言

“编译原理”是计算机类专业的一门重要专业基础课。设置该课程的目的在于系统地向学生讲述编译系统的结构、工作流程及编译程序各组成部分的设计原理和实现技术，使学生通过学习既掌握编译理论和方法方面的基本知识，也具有设计、实现、分析和维护编译程序的初步能力。

“编译原理”是一门理论性和实践性都很强的课程。进行上机实验的目的是使学生通过完成上机实验题目加深对课堂教学内容的理解，同时培养学生实际动手能力。针对我校学生的实际情况，本实验指导书设计了8个实验项目，在实验内容方面难易适中，在实验要求方面分成不同的层次，力争使学生经过一定的努力，都能够完成相应题目，收获成功的喜悦，从而激发起他们学习的兴趣和积极性。

在实验教学过程中，指导教师可根据不同专业的培养目标和不同类别学生的素质差异，灵活选择实验项目。实施方案建议如下：实验1、实验2为所有学生必做项目；计算机科学与技术专业学生必须完成实验3、4、5中的2项和实验6，实验7、8可选做；软件工程等工程类专业学生必须完成实验3、4、5 中的1项、实验6和实验7，实验8可选做；信息工程学院计算机科学与技术专业和软件工程专业的学生必须完成实验3、4、5中的1项，实验6、7、8可选做。

目　录

实验 1：PL/0 语言编译器分析实验

一、实验目的

通过阅读与解析一个实际编译器(PL/0语言编译器)的源代码，加深对编译阶段（包括词法分析、语法分析、语义分析、中间代码生成等）和编译系统软件结构的理解，并达到提高学生学习兴趣的目的。

二、实验要求

(1) 要求掌握基本的程序设计技巧（C语言）和阅读较大规模程序源代码的能力;

(2) 理解并掌握编译过程的逻辑阶段及各逻辑阶段的功能;

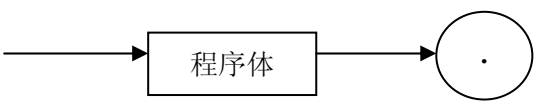
(3) 要求能把握整个系统（PL/0语言编译器）的体系结构，各功能模块的功能，各模块之间的接口;

(4) 要求能总结出实现编译过程各逻辑阶段功能采用的具体算法与技术。

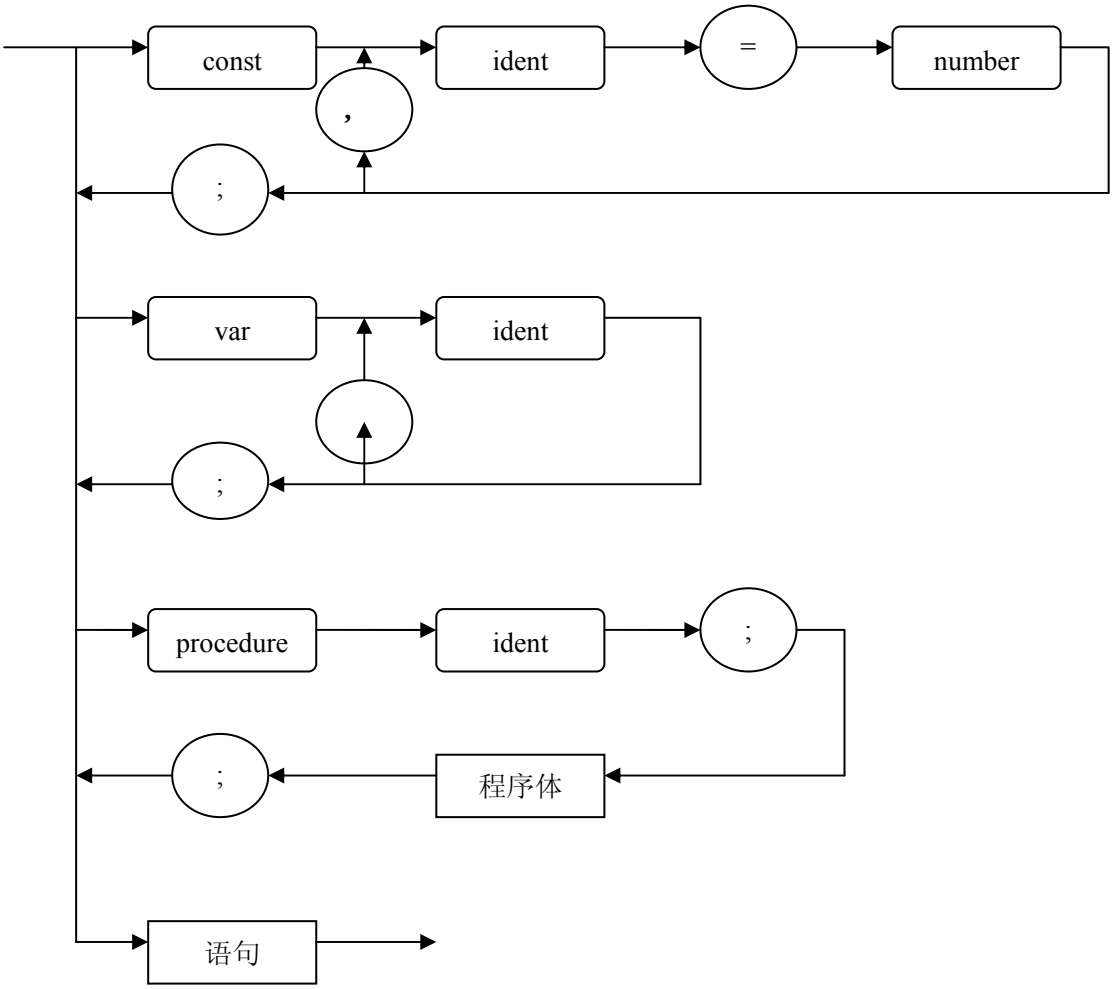
三、实验原理

PL/0语言编译器源代码见附录B，PL/0语言的语法图如下:

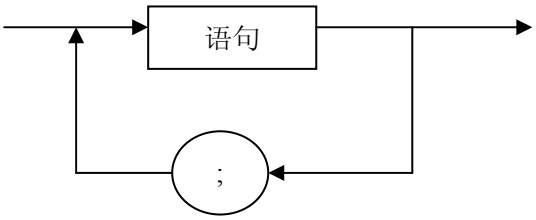
程序



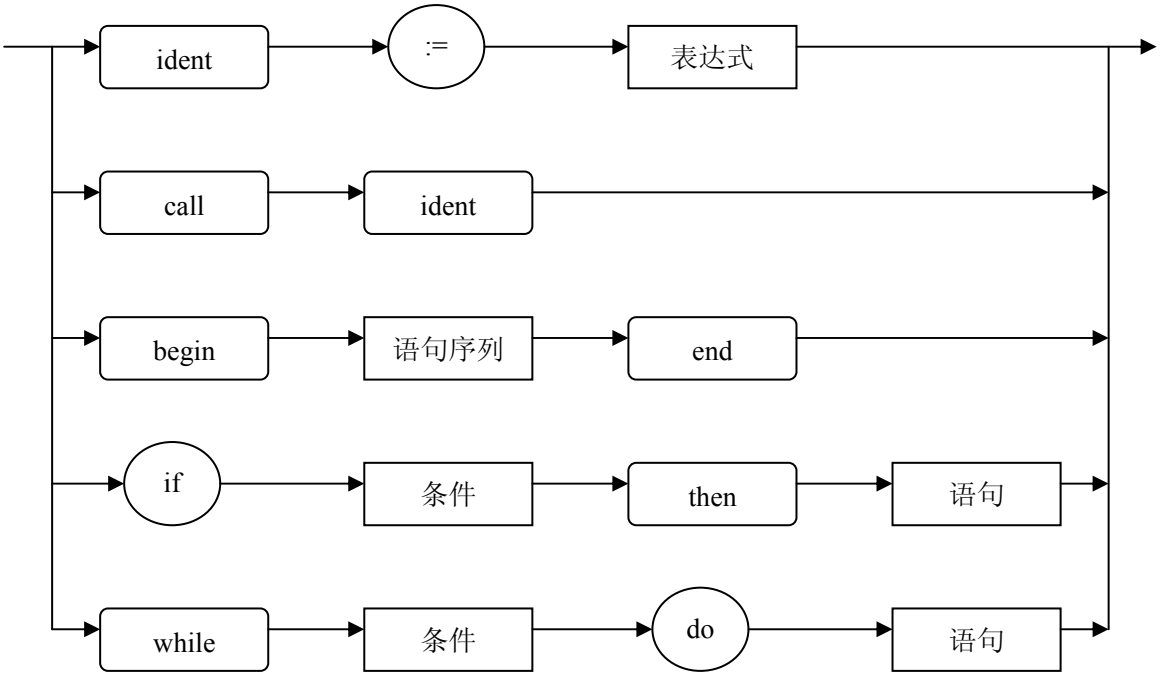
程序体



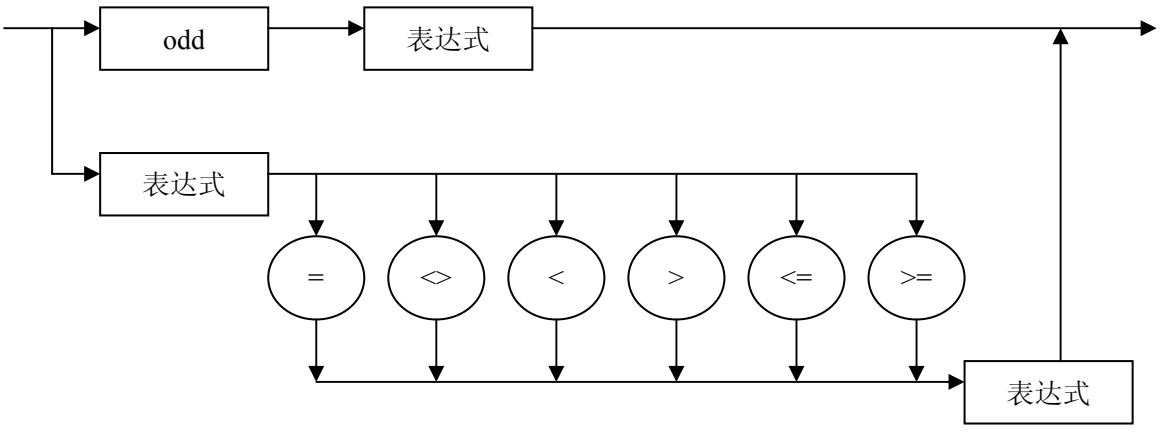
语句序列



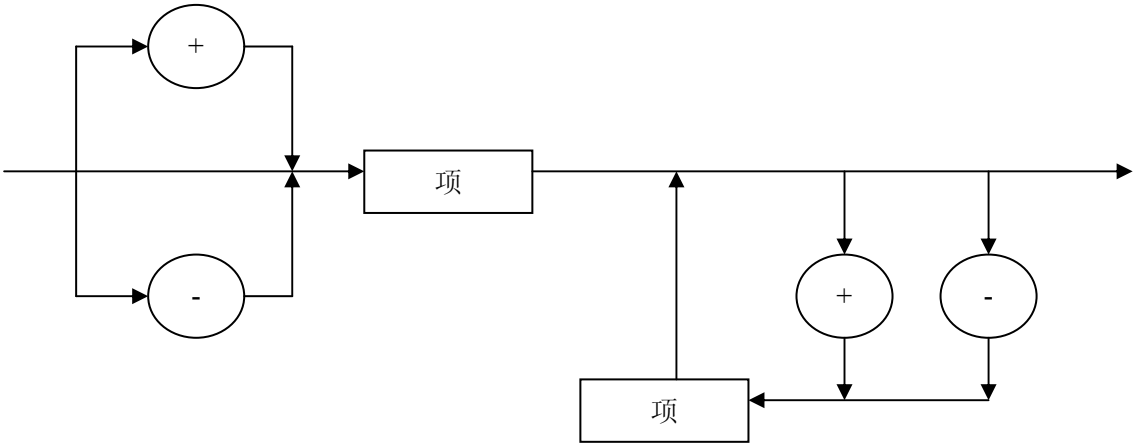
语句


ident
:=
表达式
call
ident
begin
语句序列
end
if
条件
then
语句
while
条件
do
语句

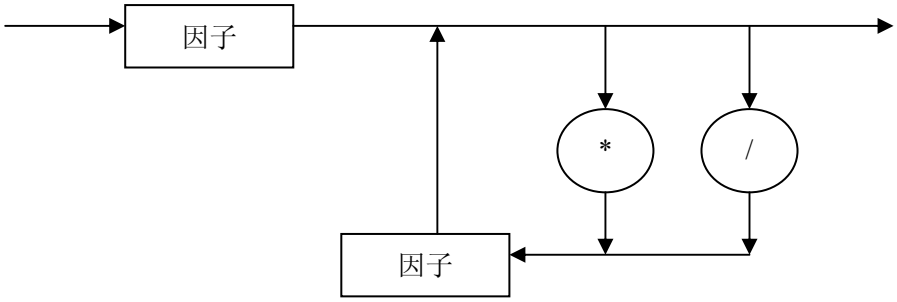
条件


odd
表达式
表达式
=
<>
<
>
<=
>=
表达式

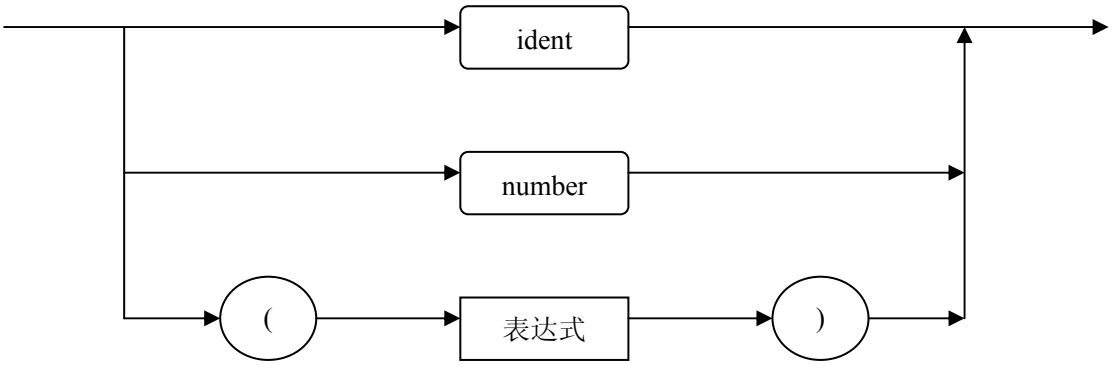
表达式



项



因子



四、实验步骤

(1) 根据PL/0语言的语法图，理解PL/0语言各级语法单位的结构，掌握PL/0语言合法程序的结构;

(2) 从总体上分析整个系统的体系结构、各功能模块的功能、各模块之间的调用关系、各模块之间的接口;

(3) 详细分析各子程序和函数的代码结构、程序流程、采用的主要算法及实现的功能;

(4) 撰写分析报告，主要内容包括系统结构框图、模块接口、主要算法、各模块程序流程图等。

实验 2：词法分析实验

一、实验目的

通过设计、开发一个高级语言的词法分析程序，加深对课堂教学内容（包括正规文法、正规表达式、有限自动机、NFA到DFA的转换、DFA的最小化）的理解，提高词法分析方法的实践能力。

二、实验要求

(1) 深入理解、掌握有限自动机及其应用;

(2) 掌握根据语言的词法规则构造识别其单词的有限自动机的方法;

(3) 掌握NFA到DFA的等价变换方法、DFA最小化的方法;

(4) 掌握设计、编码、调试词法分析程序的技术与方法，具体实现S语言（见附录A）的词法分析程序。

三、实验原理

词法分析是编译过程的第一个阶段。它的任务是对输入的字符串形式的源程序按顺序进行扫描，根据源程序的词法规则识别具有独立意义的单词，并输出单词的序列。

有限自动机是描述程序设计语言单词结构的工具，而状态转换图是有限自动机的比较直观的描述方法。根据程序设计语言的词法规则构造描述该语言单词结构的有限自动机，获取识别各类单词的形式模型，进而通过编程模拟该形式模型的运行，可实现词

法分析程序。

四、实验步骤

(1) 根据S语言的词法规则，总结S语言的单词种类与各类单词的结构特征;

(2) 设计描述S语言各类单词结构的状态转换图（即有限自动机FA）;

(3) 对描述各类单词结构的状态转换图进行合并(将各状态转换图的初始状态合并为一个唯一的初态，然后对状态重新编号），构成一个能识别S语言所有单词的状态转换图（NFA）。

(4) 对能识别所有单词的NFA进行确定化操作，将其转换成等价的DFA;

(5) 对DFA进行最小化操作;

(6) 编写程序，模拟最小化DFA的运行，实现S语言的词法分析程序;

(7) 撰写实验报告。

实验 3：递归下降语法分析实验

一、实验目的

通过设计、开发一个高级语言的递归下降语法分析程序，实现对词法分析程序所提供的单词序列进行语法检查和结构分析，加深对相关课堂教学内容的理解，提高语法分析方法的实践能力。

二、实验要求

(1) 理解语法分析在编译程序中的作用，以及它与词法分析程序的关系；

(2) 掌握递归下降语法分析方法的主要原理；

(3) 理解递归下降分析法对文法的要求；

(4) 熟练掌握Select集合的求解方法；

(5) 熟练掌握文法变换方法（消除左递归和提取公因子）。

三、实验原理

递归下降分析法是语法分析中最易懂的一种方法，基本原理是：对每个非终结符号（分别代表一个语法单位）按其产生式结构构造相应语法分析子程序，以完成该非终结符号所对应的语法单位的分析和识别任务。其中终结符号产生匹配命令，而非终结符号则产生过程调用命令。因为文法可以递归，相应子程序也是递归的，所以称这种方法为递归子程序下降法或递归下降法。其中子程序的结构与产生式结构几乎是一致的。

假设一个文法中的非终结符号A的全部产生式为$A\rightarrow\alpha_1|\alpha_2|.....|\alpha_n$，则必须满足以下条件才能保证可以唯一的选择合适的产生式，才能采用递归下降分析法:

$Select(A\rightarrow\alpha_i)\cap Select(A\rightarrow\alpha_j)=\Phi$, 其中$i\neq j$

假设文法中有如下的产生式$A\rightarrow\beta_1|\beta_2|...|\beta_n$，则应按如下方法编写语法分析子程序。

```
procedure A( )
    begin   if  token∈Select(A→β1) then  θ(β1) else
            if  token∈Select(A→β2) then  θ(β2) else
                ……
            if  token∈Select(A→βn) then  θ(βn) else
              error()
    end
```

其中 $\beta_i=X_1X_2...X_n$，$\theta(\beta_i)=\theta'(X_1);\ \theta'(X_2);...;\ \theta'(X_n)$；如果$X_i\in V_N$，$\theta'(X_i)=X_i$；如果$X_i\in V_T$，$\theta'(X_i)=Match(X_i)$；如果$X_i=\varepsilon$，$\theta'(X_i)=skip$（空语句）。

四、实验步骤

(1) 根据S语言BNF形式的语法规则（见附件A），写出S语言的上下文无关文法;

(2) 求每个产生式的Select集:

Select(A→β)= First(β)，当 ε∉First(β)

= (First(β)-{ε})∪Follow(A)，当ε∈First(β)

(3) 判断是否满足递归下降法分析条件,若不满足用消除左递归和提取公因子等文法等价变换操作对文法进行变换,使其满足递归下降法的要求;

(4) 构造递归下降语法分析程序,对文法中的每个非终结符号按其产生式结构产生相应的语法分析子程序,完成相应的识别任务。其中终结符号产生匹配命令,非终结符号则产生调用命令。实际的语法分析工作从调用主程序（开始符号S对应的程序）开始，根据产生式递归调用各个分析子程序;

(5) 撰写实验报告。

实验 4：LL（1）语法分析实验

一、实验目的

通过设计、开发一个高级语言的LL（1）语法分析程序，实现对源程序的语法检查和结构分析，加深对相关课堂教学内容（包括自顶向下语法分析、First集、Follow集、Select集、文法等价变换）的理解，提高语法分析方法的实践能力。

二、实验要求

(1) 理解语法分析在编译程序中的作用，以及它与词法分析程序的关系；

(2) 掌握LL（1）语法分析方法的主要原理；

(3) 理解LL（1）分析器模型；

(4) 理解LL（1）语法分析方法对文法的要求；

(5) 熟练掌握Select集合的求解方法和LL(1)分析表的构造方法；

(6) 熟练掌握文法变换方法（消除左递归和提取公因子）。

三、实验原理

LL(1)分析法属于自顶向下分析方法，需解决的关键问题是在构建从文法开始符号到句子的推导序列时如何确定正确的产生式，即在LL(1)分析中，每当在符号栈的栈顶出现非终结符号时，要预测用哪个产生式的右部去替换该非终结符号。LL(1)分析方法对文法的要求与递归下降分析法一样。即对于文法中每个非终结

符号A的全部产生式$A\rightarrow\alpha_1|\alpha_2|.....|\alpha_n$ ，必须满足:

$Select(A\rightarrow\alpha_i)\cap Select(A\rightarrow\alpha_j)=\Phi$,其中$i\neq j$

满足以上条件的文法称为LL（1）文法，在求解Select集的基础上可进一步构建LL（1）分析表，有了LL（1）分析表可调用LL（1）分析器模型实现语言的语法分析。LL(1)分析表的作用是对当前非终结符号和输入符号确定应该选择用哪个产生式进行推导。它的行对应文法的非终结符号，列对应终结符号，表中的值有两种：一是产生式(或其编号)，一是错误处理动作。

LL(1)分析主要包括以下四个动作，其中X为符号栈栈顶元素，a为输入流当前字符:

a) 替换：当$X\in V_N$时选相应产生式的右部β去替换X。

b) 匹配：当$X\in V_T$时它与a进行匹配，其结果可能成功，也可能失败，如果成功则符号栈中将X退栈并将输入流指针向前移动一位，不成功则报错。

c) 成功：当格局为（空，空）时报告分析成功。

d) 报错：出错后，停止分析。

四、实验步骤

(1)根据S语言BNF形式的语法规则（见附件A），写出S语言的上下文无关文法;

(2)求每个产生式的Select集:

$Select(A\rightarrow\beta)=First(\beta)$，当 $\varepsilon\notin First(\beta)$

$=(First(\beta)-\{\varepsilon\})\cup Follow(A)$，当$\varepsilon\in First(\beta)$

(3) 判断文法是否为LL（1）的，若不是则用消除左递归和提取公因子等文法等价变换算法对文法进行变换，使其满足LL（1）文法的要求;

(4) 根据Select集构建LL（1）分析表;

(5) 调用LL（1）分析器模型，编程实现LL（1）分析程序;

(6) 设计、实现能根据分析结果（产生式序列）构建源程序分析树的方法;

(7) 撰写实验报告。

实验 5：LR 语法分析实验

一、实验目的

通过设计、开发一个高级语言的LR语法分析程序，实现对源程序的语法检查和结构分析，加深对相关课堂教学内容（包括自底向上语法分析、“移进-规约”分析法、LR分析器模型、LR分析表）的理解，提高语法分析方法的实践能力。

二、实验要求

(1) 理解语法分析在编译程序中的作用，以及它与词法分析程序的关系；

(2) 掌握自底向上语法分析和LR分析方法的主要原理；

(3) 理解LR分析器模型与运行过程；

(4) 掌握LR（0）项目集规范族的构建方法和识别文法所有活前缀的DFA的构建方法；

(5) 掌握LR分析表的构造方法。

三、实验原理

LR语法分析方法是一种自底向上的语法分析方法，是当前使用最广泛的无回溯的“移进-归约”方法。它根据栈中的符号串和向前查看的k(k⩾0)个输入符号，通过查LR分析表就能唯一确定分析器的动作是移进还是归约，以及用哪个产生式进行归约。该方法的优点：文法适用范围广、识别效率高、查错能力强、可自动构

造等。

一个LR分析器由一个输入串、一个栈和一个带有分析表的总控程序组成。栈中存放着由“历史”和“展望”材料抽象而来的各种“状态”。任何时候，栈顶的状态都代表了整个的历史和已推测出的展望。LR分析器的工作过程是由总控程序根据分析表（包括ACTION子表和GOTO子表），使得分析器从一种格局向另一种格局变化的过程。初始格局：（S_0，$a_1a_2\cdots a_n$\$），S_0为分析器的初态，\$为输入串的结束标记。

分析过程的每步结果可表示为：（$S_0X_1S_1\cdots X_mS_m$，$a_ia_{i+1}\cdots a_n$\$）。分析器的下一次动作是由栈顶状态S_m和当前输入符号a_i所唯一确定的，即：执行ACTION[S_m，a_i]规定的动作。经执行各种可能的动作后，分析器的格局可如下变化：

a) 若ACTION(S_m，a_i)=“移进S”，则分析器格局变为（$S_0X_1S_1\cdots X_mS_ma_iS$，$a_{i+1}\cdots a_n$\$）。

b) 若ACTION(S_m，a_i)=“归约A→β”，则分析器格局变为（$S_0X_1S_1\cdots X_{m-r}S_{m-r}AS$，$a_ia_{i+1}\cdots a_n$\$），其中S=GOTO(S_{m-r}，A)，$|\beta|=r$。

c) 若ACTION(S_m，a_i)=“接受”，则分析成功，正常停止。

d) 若ACTION(S_m，a_i)=“ERROR”，语法出错，进行出错处理。

四、实验步骤

(1)根据S语言BNF形式的语法规则（见附件A），写出S语言的上下

文无关文法;

(2) 求该文法的LR（0）项目集规范族;

(3) 构建识别该文法所有活前缀的DFA;

(4) 构建LR分析表;

(5) 调用LR分析器模型，编程实现LR分析程序;

(6) 设计、实现能根据分析结果（产生式序列）构建源程序分析树方法;

(7) 撰写实验报告。

实验 6：语义检查与中间代码生成实验

一、实验目的

通过设计、开发一个高级语言的语义检查和中间代码生成程序，加深对相关课堂教学内容，包括语法制导翻译技术、类型确定、类型检查、常见可执行语句（如赋值语句、条件语句、循环语句）翻译技术的理解。

二、实验要求

(1) 了解语义检查和中间代码生成的目的和意义;

(2) 掌握语义检查和中间代码生成的一般内容;

(3) 掌握语法制导翻译技术(特别是针对S-属性定义的自底向上翻译方法和针对L-属性定义的深度优先翻译方法）;

(4) 掌握根据翻译目标编制语义子程序的方法(可模仿教材中的语义规则进行设计）。

三、实验原理

语义分析是任何编译程序必不可少的一个阶段，在整个编译过程中，词法分析和语法分析是对源程序形式上的识别和处理，而语义分析程序是对源程序的语义做相应的处理工作。中间代码生成不是编译器的必须阶段，生成中间代码的目的是为了便于优化和移植。

语义分析和中间代码生成的主流技术是语法制导翻译技术。语

法制导翻译的基本思想: 为每个产生式配上一个语义子程序,(该子程序描述了一个产生式所对应的翻译工作。这些工作包括: 生成中间代码，查填有关的符号表，检查和报错，修改编译程序某些工作变量的值等）。在语法分析过程中，每当一个产生式用于推导（自顶向下分析）或归约（自底向上分析）时，就调用该产生式所对应的语义子程序，以完成既定的翻译任务。

四、实验步骤

(1) 熟悉S语言的语义，了解语义分析阶段需要进行哪些语义检查和需要对哪些可执行语句进行翻译;

(2) 分析S语言的文法，必要的话可对文法进行改写（如增加标记非终结符号并为其添加产生式）;

(3) 为相应产生式编写语义子程序(模仿教材中的语义规则进行设计)，实现对运算及运算分量进行类型检查(运算的合法性与运算分量类型的一致性或相容性)、变量定义的唯一性检查等语义检查工作;

(4) 理解掌握变量的中间代码、表达式的中间代码、语句的中间代码的结构（见教材）;

(5) 编写语义子程序（模仿教材中的语义规则进行设计），实现S语言中各类语句（赋值语句、条件语句、循环语句）的翻译工作，将源程序翻译为中间代码。

(6) 撰写实验报告。

实验 7：编译器集成实验

一、实验目的

通过将一个高级语言的各个编译子程序（词法分析程序、语法分析程序、语义分析程序、中间代码生成程序）集成为一个完整的编译器，提高对完整编译程序体系结构的认识，提高大型软件的分析、设计、开发、调试能力。

二、实验要求

(1) 已自行开发词法分析程序、语法分析程序、语义分析程序、中间代码生成程序;

(2) 掌握软件界面设计、开发技术;

(3) 理解编译各阶段功能及各阶段之间的接口;

(4) 掌握大型软件功能模块的集成、调试技术。

三、实验原理

集成工作环境已成为大多数软件产品的必要组成部分，也是实用编译器所必须的，现在主流的编译系统都采用了集成开发环境，能将程序的编辑、编译、查错、调试集成在统一的界面中完成。

一个完整的编译器应包括词法分析程序、语法分析程序、语义分析程序、中间代码生成程序、代码优化程序和目标代码生成程序，为避免陷入目标机器的技术细节，降低实验难度，本实验不涉及代码优化和目标代码生成。

四、实验步骤

(1) 开发编译器集成环境，包括主界面设计、各个功能窗口设计、菜单项设计等;

(2) 设计实现一个程序编辑器，可提供源程序的基本编辑工作，具有一般文本编辑器的功能，包括插入、删除、拷贝、黏贴、剪切等。

(3) 为各功能模块设计统一调用接口,将各功能模块集成到同一环境中;

(4) 撰写实验报告。

实验 8：S 语言扩充实验

一、实验目的

通过对一个高级语言的数据类型、语句类型及其它语言结构的扩充，并实现其编译程序，进一步提高编译器设计的实践能力。

二、实验要求

(1) 深刻理解主流编程语言（面向过程的程序设计语言）的语言结构与特征;

(2) 掌握通过增加产生式对高级语言进行扩充的技术;

(3) 掌握各编译子程序的设计与实现技术。

三、实验原理

S语言是一个面向实验教学的、具有某些高级程序设计语言特点的模型语言，数据类型简单（只有整数类型）、语句类型有限（只有赋值语句、条件语句、当循环语句、复合语句等几种）、没有定义子程序和函数、不允许递归调用。这样一种语言没有目前主流程序设计语言的一些重要特征。可通过实验，对S语言进行扩充，实现一些高级语言中的常见语言结构。

四、实验步骤

(1) 数据类型扩充实验，可增加实数类型、字符串类型、数组类型、指针类型等;

(2) 语句扩充实验，可增加FOR循环语句、CASE分支语句等;

(3) 定义子程序或函数结构，并实现它;

(4) 定义程序的递归调用，并实现递归调用的处理;

(5) 撰写实验报告。

附录 A：S 语言语法的 BNF 表示

(1) <程序>→[<常量说明>][<变量说明>]<语句>

(2) <常量说明>→Const <常量定义>{, <常量定义>};

(3) <常量定义>→<标识符>=<无符号整数>

(4) <无符号整数>→<数字>{<数字>}

(5) <字母>→a|b|c| … |z

(6) <数字>→0|1|2| … |9

(7) <标识符>→<字母>{<字母>|<数字>}

(8) <变量说明>→Var <标识符>{, <标识符>};

(9) <语句>→<赋值语句>|<条件语句>|<当循环语句>|<复合语句>|ε

(10) <赋值语句>→<标识符>=<表达式>;

(11) <表达式>→[+|-]<项>{<加法运算符><项>}

(12) <项>→<因子>{<乘法运算符><因子>}

(13) <因子>→<标识符>|<无符号整数>| ‘(’ <表达式> ‘)’

(14) <加法运算符>→+|-

(15) <乘法运算符>→*|/

(16) <条件语句>→if <条件> then <语句>| if <条件> then <语句> else <语句>

(17) <条件>→<表达式><关系运算符><表达式>

(18) <关系运算符>→==|<=|<|>|>=|<>

(19) <当循环语句>→while <条件> do <语句>

(20) <复合语句>→begin <语句>{; <语句>} end

注：产生式中<、>括起的部分表示一个非终结符号，[、]括起的部分表示可选项，{、}括起的部分表示可重复，符号 | 表示“或”。

附录 B：PL/0 语言编译器源代码

PL/0 语言编译器源程序包括如下 C 程序文件：PL0.h、PL0.c、set.h 和 set.c。

```
/************  PL0.h  *************/

#include <stdio.h>

#define NRW         11     // number of reserved words
#define TXMAX       500    // length of identifier table
#define MAXNUMLEN   14     // maximum number of digits in numbers
#define NSYM        10     // maximum number of symbols in array ssym and csym
#define MAXIDLEN    10     // length of identifiers
#define MAXADDRESS 32767   // maximum address
#define MAXLEVEL    32     // maximum depth of nesting block
#define CXMAX       500    // size of code array
#define MAXSYM      30     // maximum number of symbols
#define STACKSIZE   1000   // maximum storage

enum symtype
{
    SYM_NULL,
    SYM_IDENTIFIER,
    SYM_NUMBER,
    SYM_PLUS,
    SYM_MINUS,
    SYM_TIMES,
    SYM_SLASH,
    SYM_ODD,
    SYM_EQU,
    SYM_NEQ,
    SYM_LES,
    SYM_LEQ,
    SYM_GTR,
    SYM_GEQ,
    SYM_LPAREN,
    SYM_RPAREN,
    SYM_COMMA,
    SYM_SEMICOLON,
    SYM_PERIOD,
    SYM_BECOMES,
    SYM_BEGIN,
```

```
    SYM_END,
    SYM_IF,
    SYM_THEN,
    SYM_WHILE,
    SYM_DO,
    SYM_CALL,
    SYM_CONST,
    SYM_VAR,
    SYM_PROCEDURE
};

enum idtype
{
    ID_CONSTANT, ID_VARIABLE, ID_PROCEDURE
};

enum opcode
{
    LIT, OPR, LOD, STO, CAL, INT, JMP, JPC
};

enum oprcode
{
    OPR_RET, OPR_NEG, OPR_ADD, OPR_MIN,
    OPR_MUL, OPR_DIV, OPR_ODD, OPR_EQU,
    OPR_NEQ, OPR_LES, OPR_LEQ, OPR_GTR,
    OPR_GEQ
};

typedef struct
{
    int f; // function code
    int l; // level
    int a; // displacement address
} instruction;

//////////////////////////////////////////////////////////////////////
char* err_msg[] =
{
/*  0 */    "",
/*  1 */    "Found ':=' when expecting '='.",
/*  2 */    "There must be a number to follow '='.",
/*  3 */    "There must be an '=' to follow the identifier.",
/*  4 */    "There must be an identifier to follow 'const', 'var', or 'procedure'.",
```

```
/*  5 */    "Missing ',' or ';'.",
/*  6 */    "Incorrect procedure name.",
/*  7 */    "Statement expected.",
/*  8 */    "Follow the statement is an incorrect symbol.",
/*  9 */    "'.' expected.",
/* 10 */    "';' expected.",
/* 11 */    "Undeclared identifier.",
/* 12 */    "Illegal assignment.",
/* 13 */    "':=' expected.",
/* 14 */    "There must be an identifier to follow the 'call'.",
/* 15 */    "A constant or variable can not be called.",
/* 16 */    "'then' expected.",
/* 17 */    "';' or 'end' expected.",
/* 18 */    "'do' expected.",
/* 19 */    "Incorrect symbol.",
/* 20 */    "Relative operators expected.",
/* 21 */    "Procedure identifier can not be in an expression.",
/* 22 */    "Missing ')'.",
/* 23 */    "The symbol can not be followed by a factor.",
/* 24 */    "The symbol can not be as the beginning of an expression.",
/* 25 */    "The number is too great.",
/* 26 */    "",
/* 27 */    "",
/* 28 */    "",
/* 29 */    "",
/* 30 */    "",
/* 31 */    "",
/* 32 */    "There are too many levels."
};

////////////////////////////////////////////////////////////////////
char ch;         // last character read
int  sym;        // last symbol read
char id[MAXIDLEN + 1]; // last identifier read
int  num;        // last number read
int  cc;         // character count
int  ll;         // line length
int  kk;
int  err;
int  cx;         // index of current instruction to be generated.
int  level = 0;
int  tx = 0;
char line[80];
instruction code[CXMAX];
```

```
char* word[NRW + 1] =
{
    "", /* place holder */
    "begin", "call", "const", "do", "end","if",
    "odd", "procedure", "then", "var", "while"
};

int wsym[NRW + 1] =
{
    SYM_NULL, SYM_BEGIN, SYM_CALL, SYM_CONST, SYM_DO, SYM_END,
    SYM_IF, SYM_ODD, SYM_PROCEDURE, SYM_THEN, SYM_VAR, SYM_WHILE
};

int ssym[NSYM + 1] =
{
    SYM_NULL, SYM_PLUS, SYM_MINUS, SYM_TIMES, SYM_SLASH,
    SYM_LPAREN, SYM_RPAREN, SYM_EQU, SYM_COMMA, SYM_PERIOD, SYM_SEMICOLON
};

char csym[NSYM + 1] =
{
    ' ', '+', '-', '*', '/', '(', ')', '=', ',', '.', ';'
};

#define MAXINS   8

char* mnemonic[MAXINS] =
{
    "LIT", "OPR", "LOD", "STO", "CAL", "INT", "JMP", "JPC"
};

typedef struct
{
    char name[MAXIDLEN + 1];
    int  kind;
    int  value;
} comtab;

comtab table[TXMAX];

typedef struct
{
    char  name[MAXIDLEN + 1];
```

```
    int   kind;
    short level;
    short address;
} mask;

FILE* infile;

// EOF PL0.h


/************  SET.h  *************/

#ifndef SET_H
#define SET_H

typedef struct snode
{
    int elem;
    struct snode* next;
} snode, *symset;

symset phi, declbegsys, statbegsys, facbegsys, relset;

symset createset(int data, .../* SYM_NULL */);
void destroyset(symset s);
symset uniteset(symset s1, symset s2);
int inset(int elem, symset s);

#endif
// EOF set.h


/************  SET.c  *************/

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include "set.h"

symset uniteset(symset s1, symset s2)
{
    symset s;
    snode* p;
```

```
    s = p = (snode*) malloc(sizeof(snode));
    while (s1 && s2)
    {
        p->next = (snode*) malloc(sizeof(snode));
        p = p->next;
        if (s1->elem < s2->elem)
        {
            p->elem = s1->elem;
            s1 = s1->next;
        }
        else
        {
            p->elem = s2->elem;
            s2 = s2->next;
        }
    }

    while (s1)
    {
        p->next = (snode*) malloc(sizeof(snode));
        p = p->next;
        p->elem = s1->elem;
        s1 = s1->next;

    }

    while (s2)
    {
        p->next = (snode*) malloc(sizeof(snode));
        p = p->next;
        p->elem = s2->elem;
        s2 = s2->next;
    }

    p->next = NULL;

    return s;
} // uniteset

void setinsert(symset s, int elem)
{
    snode* p = s;
    snode* q;
```

```
    while (p->next && p->next->elem < elem)
    {
        p = p->next;
    }

    q = (snode*) malloc(sizeof(snode));
    q->elem = elem;
    q->next = p->next;
    p->next = q;
} // setinsert

symset createset(int elem, .../* SYM_NULL */)
{
    va_list list;
    symset s;

    s = (snode*) malloc(sizeof(snode));
    s->next = NULL;

    va_start(list, elem);
    while (elem)
    {
        setinsert(s, elem);
        elem = va_arg(list, int);
    }
    va_end(list);
    return s;
} // createset

void destroyset(symset s)
{
    snode* p;

    while (s)
    {
        p = s;
        s = s->next;
        free(p);
    }
} // destroyset

int inset(int elem, symset s)
{
    s = s->next;
```

```
    while (s && s->elem < elem)
        s = s->next;

    if (s && s->elem == elem)
        return 1;
    else
        return 0;
} // inset

// EOF set.c


/************  PL0.c  *************/

// pl0 compiler source code

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "set.h"
#include "pl0.h"

////////////////////////////////////////////////////////////////////
// print error message.
void error(n)
{
    int i;

    printf("      ");
    for (i = 1; i <= cc - 1; i++)
        printf(" ");
    printf("^\n");
    printf("Error %3d: %s\n", n, err_msg[n]);
    err++;
} // error

////////////////////////////////////////////////////////////////////
void getch(void)
{
    if (cc == ll)
    {
        if (feof(infile))
        {
```

```
            printf("\nPROGRAM INCOMPLETE\n");
            exit(1);
        }
        ll = cc = 0;
        printf("%5d  ", cx);
        while (!feof(infile) && (ch = getc(infile))!='\n')
        {
            printf("%c", ch);
            line[++ll] = ch;
        } // while
        printf("\n");
        line[++ll] = ' ';
    }
    ch = line[++cc];
} // getch

//////////////////////////////////////////////////////////////////////
// gets a symbol from input stream.
void getsym(void)
{
    int i, k;
    char a[MAXIDLEN + 1];

    while (ch == ' ')
        getch();

    if (isalpha(ch))
    { // symbol is a reserved word or an identifier.
        k = 0;
        do
        {
            if (k < MAXIDLEN)
                a[k++] = ch;
            getch();
        }
        while (isalpha(ch) || isdigit(ch));
        a[k] = 0;
        strcpy(id, a);
        word[0] = id;
        i = NRW;
        while (strcmp(id, word[i--]));
        if (++i)
            sym = wsym[i]; // symbol is a reserved word
        else
```

```
            sym = SYM_IDENTIFIER;   // symbol is an identifier
    }
    else if (isdigit(ch))
    { // symbol is a number.
        k = num = 0;
        sym = SYM_NUMBER;
        do
        {
            num = num * 10 + ch - '0';
            k++;
            getch();
        }
        while (isdigit(ch));
        if (k > MAXNUMLEN)
            error(25);     // The number is too great.
    }
    else if (ch == ':')
    {
        getch();
        if (ch == '=')
        {
            sym = SYM_BECOMES; // :=
            getch();
        }
        else
        {
            sym = SYM_NULL;       // illegal?
        }
    }
    else if (ch == '>')
    {
        getch();
        if (ch == '=')
        {
            sym = SYM_GEQ;     // >=
            getch();
        }
        else
        {
            sym = SYM_GTR;     // >
        }
    }
    else if (ch == '<')
    {
```

```
        getch();
        if (ch == '=')
        {
            sym = SYM_LEQ;     // <=
            getch();
        }
        else if (ch == '>')
        {
            sym = SYM_NEQ;     // <>
            getch();
        }
        else
        {
            sym = SYM_LES;     // <
        }
    }
    else
    { // other tokens
        i = NSYM;
        csym[0] = ch;
        while (csym[i--] != ch);
        if (++i)
        {
            sym = ssym[i];
            getch();
        }
        else
        {
            printf("Fatal Error: Unknown character.\n");
            exit(1);
        }
    }
} // getsym

//////////////////////////////////////////////////////////////////////
// generates (assembles) an instruction.
void gen(int x, int y, int z)
{
    if (cx > CXMAX)
    {
        printf("Fatal Error: Program too long.\n");
        exit(1);
    }
    code[cx].f = x;
```

```
    code[cx].l = y;
    code[cx++].a = z;
} // gen

//////////////////////////////////////////////////////////////////////
// tests if error occurs and skips all symbols that do not belongs to s1 or s2.
void test(symset s1, symset s2, int n)
{
    symset s;

    if (! inset(sym, s1))
    {
        error(n);
        s = uniteset(s1, s2);
        while(! inset(sym, s))
            getsym();
        destroyset(s);
    }
} // test

//////////////////////////////////////////////////////////////////////
int dx;  // data allocation index

// enter object(constant, variable or procedre) into table.
void enter(int kind)
{
    mask* mk;

    tx++;
    strcpy(table[tx].name, id);
    table[tx].kind = kind;
    switch (kind)
    {
    case ID_CONSTANT:
        if (num > MAXADDRESS)
        {
            error(25); // The number is too great.
            num = 0;
        }
        table[tx].value = num;
        break;
    case ID_VARIABLE:
        mk = (mask*) &table[tx];
        mk->level = level;
```

```
        mk->address = dx++;
        break;
    case ID_PROCEDURE:
        mk = (mask*) &table[tx];
        mk->level = level;
        break;
    } // switch
} // enter

//////////////////////////////////////////////////////////////////////
// locates identifier in symbol table.
int position(char* id)
{
    int i;
    strcpy(table[0].name, id);
    i = tx + 1;
    while (strcmp(table[--i].name, id) != 0);
    return i;
} // position

//////////////////////////////////////////////////////////////////////
void constdeclaration()
{
    if (sym == SYM_IDENTIFIER)
    {
        getsym();
        if (sym == SYM_EQU || sym == SYM_BECOMES)
        {
            if (sym == SYM_BECOMES)
                error(1); // Found ':=' when expecting '='.
            getsym();
            if (sym == SYM_NUMBER)
            {
                enter(ID_CONSTANT);
                getsym();
            }
            else
            {
                error(2); // There must be a number to follow '='.
            }
        }
        else
        {
            error(3); // There must be an '=' to follow the identifier.
```

```
        }
    }
    error(4); // There must be an identifier to follow 'const', 'var', or
'procedure'.
} // constdeclaration

////////////////////////////////////////////////////////////////////
void vardeclaration(void)
{
    if (sym == SYM_IDENTIFIER)
    {
        enter(ID_VARIABLE);
        getsym();
    }
    else
    {
        error(4); // There must be an identifier to follow 'const', 'var', or
'procedure'.
    }
} // vardeclaration

////////////////////////////////////////////////////////////////////
void listcode(int from, int to)
{
    int i;

    printf("\n");
    for (i = from; i < to; i++)
    {
        printf("%5d %s\t%d\t%d\n", i, mnemonic[code[i].f], code[i].l, code[i].a);
    }
    printf("\n");
} // listcode

////////////////////////////////////////////////////////////////////
void factor(symset fsys)
{
    void expression();
    int i;
    symset set;

    test(facbegsys, fsys, 24); // The symbol can not be as the beginning of an
expression.
```

```
while (inset(sym, facbegsys))
{
    if (sym == SYM_IDENTIFIER)
    {
        if ((i = position(id)) == 0)
        {
            error(11); // Undeclared identifier.
        }
        else
        {
            switch (table[i].kind)
            {
                mask* mk;
            case ID_CONSTANT:
                gen(LIT, 0, table[i].value);
                break;
            case ID_VARIABLE:
                mk = (mask*) &table[i];
                gen(LOD, level - mk->level, mk->address);
                break;
            case ID_PROCEDURE:
                error(21); // Procedure identifier can not be in an expression.
                break;
            } // switch
        }
        getsym();
    }
    else if (sym == SYM_NUMBER)
    {
        if (num > MAXADDRESS)
        {
            error(25); // The number is too great.
            num = 0;
        }
        gen(LIT, 0, num);
        getsym();
    }
    else if (sym == SYM_LPAREN)
    {
        getsym();
        set = uniteset(createset(SYM_RPAREN, SYM_NULL), fsys);
        expression(set);
        destroyset(set);
        if (sym == SYM_RPAREN)
```

```
            {
                getsym();
            }
            else
            {
                error(22); // Missing ')'.
            }
        }
        test(fsys, createset(SYM_LPAREN, SYM_NULL), 23);
    } // while
} // factor

//////////////////////////////////////////////////////////////////////
void term(symset fsys)
{
    int mulop;
    symset set;

    set = uniteset(fsys, createset(SYM_TIMES, SYM_SLASH, SYM_NULL));
    factor(set);
    while (sym == SYM_TIMES || sym == SYM_SLASH)
    {
        mulop = sym;
        getsym();
        factor(set);
        if (mulop == SYM_TIMES)
        {
            gen(OPR, 0, OPR_MUL);
        }
        else
        {
            gen(OPR, 0, OPR_DIV);
        }
    } // while
    destroyset(set);
} // term

//////////////////////////////////////////////////////////////////////
void expression(symset fsys)
{
    int addop;
    symset set;

    set = uniteset(fsys, createset(SYM_PLUS, SYM_MINUS, SYM_NULL));
```

```
    if (sym == SYM_PLUS || sym == SYM_MINUS)
    {
        addop = sym;
        getsym();
        term(set);
        if (addop == SYM_MINUS)
        {
            gen(OPR, 0, OPR_NEG);
        }
    }
    else
    {
        term(set);
    }

    while (sym == SYM_PLUS || sym == SYM_MINUS)
    {
        addop = sym;
        getsym();
        term(set);
        if (addop == SYM_PLUS)
        {
            gen(OPR, 0, OPR_ADD);
        }
        else
        {
            gen(OPR, 0, OPR_MIN);
        }
    } // while

    destroyset(set);
} // expression

////////////////////////////////////////////////////////////////////////
void condition(symset fsys)
{
    int relop;
    symset set;

    if (sym == SYM_ODD)
    {
        getsym();
        expression(fsys);
        gen(OPR, 0, 6);
```

```
    }
    else
    {
        set = uniteset(relset, fsys);
        expression(set);
        destroyset(set);
        if (! inset(sym, relset))
        {
            error(20);
        }
        else
        {
            relop = sym;
            getsym();
            expression(fsys);
            switch (relop)
            {
            case SYM_EQU:
                gen(OPR, 0, OPR_EQU);
                break;
            case SYM_NEQ:
                gen(OPR, 0, OPR_NEQ);
                break;
            case SYM_LES:
                gen(OPR, 0, OPR_LES);
                break;
            case SYM_GEQ:
                gen(OPR, 0, OPR_GEQ);
                break;
            case SYM_GTR:
                gen(OPR, 0, OPR_GTR);
                break;
            case SYM_LEQ:
                gen(OPR, 0, OPR_LEQ);
                break;
            } // switch
        } // else
    } // else
} // condition

////////////////////////////////////////////////////////////////////
void statement(symset fsys)
{
    int i, cx1, cx2;
```

```
    symset set1, set;

    if (sym == SYM_IDENTIFIER)
    { // variable assignment
        mask* mk;
        if (! (i = position(id)))
        {
            error(11); // Undeclared identifier.
        }
        else if (table[i].kind != ID_VARIABLE)
        {
            error(12); // Illegal assignment.
            i = 0;
        }
        getsym();
        if (sym == SYM_BECOMES)
        {
            getsym();
        }
        else
        {
            error(13); // ':=' expected.
        }
        expression(fsys);
        mk = (mask*) &table[i];
        if (i)
        {
            gen(STO, level - mk->level, mk->address);
        }
    }
    else if (sym == SYM_CALL)
    { // procedure call
        getsym();
        if (sym != SYM_IDENTIFIER)
        {
            error(14); // There must be an identifier to follow the 'call'.
        }
        else
        {
            if (! (i = position(id)))
            {
                error(11); // Undeclared identifier.
            }
            else if (table[i].kind == ID_PROCEDURE)
```

```
            {
                mask* mk;
                mk = (mask*) &table[i];
                gen(CAL, level - mk->level, mk->address);
            }
            else
            {
                error(15); // A constant or variable can not be called.
            }
            getsym();
        }
    }
    else if (sym == SYM_IF)
    { // if statement
        getsym();
        set1 = createset(SYM_THEN, SYM_DO, SYM_NULL);
        set = uniteset(set1, fsys);
        condition(set);
        destroyset(set1);
        destroyset(set);
        if (sym == SYM_THEN)
        {
            getsym();
        }
        else
        {
            error(16); // 'then' expected.
        }
        cx1 = cx;
        gen(JPC, 0, 0);
        statement(fsys);
        code[cx1].a = cx;
    }
    else if (sym == SYM_BEGIN)
    { // block
        getsym();
        set1 = createset(SYM_SEMICOLON, SYM_END, SYM_NULL);
        set = uniteset(set1, fsys);
        statement(set);
        while (sym == SYM_SEMICOLON || inset(sym, statbegsys))
        {
            if (sym == SYM_SEMICOLON)
            {
                getsym();
```

```
            }
            else
            {
                error(10);
            }
            statement(set);
        } // while
        destroyset(set1);
        destroyset(set);
        if (sym == SYM_END)
        {
            getsym();
        }
        else
        {
            error(17); // ';' or 'end' expected.
        }
    }
    else if (sym == SYM_WHILE)
    { // while statement
        cx1 = cx;
        getsym();
        set1 = createset(SYM_DO, SYM_NULL);
        set = uniteset(set1, fsys);
        condition(set);
        destroyset(set1);
        destroyset(set);
        cx2 = cx;
        gen(JPC, 0, 0);
        if (sym == SYM_DO)
        {
            getsym();
        }
        else
        {
            error(18); // 'do' expected.
        }
        statement(fsys);
        gen(JMP, 0, cx1);
        code[cx2].a = cx;
    }
    test(fsys, phi, 19);
} // statement
```

```
////////////////////////////////////////////////////////////////////////
void block(symset fsys)
{
    int cx0; // initial code index
    mask* mk;
    int block_dx;
    int savedTx;
    symset set1, set;

    dx = 3;
    block_dx = dx;
    mk = (mask*) &table[tx];
    mk->address = cx;
    gen(JMP, 0, 0);
    if (level > MAXLEVEL)
    {
        error(32); // There are too many levels.
    }
    do
    {
        if (sym == SYM_CONST)
        { // constant declarations
            getsym();
            do
            {
                constdeclaration();
                while (sym == SYM_COMMA)
                {
                    getsym();
                    constdeclaration();
                }
                if (sym == SYM_SEMICOLON)
                {
                    getsym();
                }
                else
                {
                    error(5); // Missing ',' or ';'.
                }
            }
            while (sym == SYM_IDENTIFIER);
        } // if

        if (sym == SYM_VAR)
```

```
        { // variable declarations
            getsym();
            do
            {
                vardeclaration();
                while (sym == SYM_COMMA)
                {
                    getsym();
                    vardeclaration();
                }
                if (sym == SYM_SEMICOLON)
                {
                    getsym();
                }
                else
                {
                    error(5); // Missing ',' or ';'.
                }
            }
            while (sym == SYM_IDENTIFIER);
//          block = dx;
        } // if

        while (sym == SYM_PROCEDURE)
        { // procedure declarations
            getsym();
            if (sym == SYM_IDENTIFIER)
            {
                enter(ID_PROCEDURE);
                getsym();
            }
            else
            {
                error(4); // There must be an identifier to follow 'const', 'var',
or 'procedure'.
            }


            if (sym == SYM_SEMICOLON)
            {
                getsym();
            }
            else
            {
```

```
            error(5); // Missing ',' or ';'.
        }

        level++;
        savedTx = tx;
        set1 = createset(SYM_SEMICOLON, SYM_NULL);
        set = uniteset(set1, fsys);
        block(set);
        destroyset(set1);
        destroyset(set);
        tx = savedTx;
        level--;

        if (sym == SYM_SEMICOLON)
        {
            getsym();
            set1 = createset(SYM_IDENTIFIER, SYM_PROCEDURE, SYM_NULL);
            set = uniteset(statbegsys, set1);
            test(set, fsys, 6);
            destroyset(set1);
            destroyset(set);
        }
        else
        {
            error(5); // Missing ',' or ';'.
        }
    } // while
    set1 = createset(SYM_IDENTIFIER, SYM_NULL);
    set = uniteset(statbegsys, set1);
    test(set, declbegsys, 7);
    destroyset(set1);
    destroyset(set);
}
while (inset(sym, declbegsys));

code[mk->address].a = cx;
mk->address = cx;
cx0 = cx;
gen(INT, 0, block_dx);
set1 = createset(SYM_SEMICOLON, SYM_END, SYM_NULL);
set = uniteset(set1, fsys);
statement(set);
destroyset(set1);
destroyset(set);
```

```
    gen(OPR, 0, OPR_RET); // return
    test(fsys, phi, 8); // test for error: Follow the statement is an incorrect
symbol.
    listcode(cx0, cx);
} // block

//////////////////////////////////////////////////////////////////////
int base(int stack[], int currentLevel, int levelDiff)
{
    int b = currentLevel;

    while (levelDiff--)
        b = stack[b];
    return b;
} // base

//////////////////////////////////////////////////////////////////////
// interprets and executes codes.
void interpret()
{
    int pc;        // program counter
    int stack[STACKSIZE];
    int top;       // top of stack
    int b;         // program, base, and top-stack register
    instruction i; // instruction register

    printf("Begin executing PL/0 program.\n");

    pc = 0;
    b = 1;
    top = 3;
    stack[1] = stack[2] = stack[3] = 0;
    do
    {
        i = code[pc++];
        switch (i.f)
        {
        case LIT:
            stack[++top] = i.a;
            break;
        case OPR:
            switch (i.a) // operator
            {
            case OPR_RET:
```

```
        top = b - 1;
        pc = stack[top + 3];
        b = stack[top + 2];
        break;
    case OPR_NEG:
        stack[top] = -stack[top];
        break;
    case OPR_ADD:
        top--;
        stack[top] += stack[top + 1];
        break;
    case OPR_MIN:
        top--;
        stack[top] -= stack[top + 1];
        break;
    case OPR_MUL:
        top--;
        stack[top] *= stack[top + 1];
        break;
    case OPR_DIV:
        top--;
        if (stack[top + 1] == 0)
        {
            fprintf(stderr, "Runtime Error: Divided by zero.\n");
            fprintf(stderr, "Program terminated.\n");
            continue;
        }
        stack[top] /= stack[top + 1];
        break;
    case OPR_ODD:
        stack[top] %= 2;
        break;
    case OPR_EQU:
        top--;
        stack[top] = stack[top] == stack[top + 1];
        break;
    case OPR_NEQ:
        top--;
        stack[top] = stack[top] != stack[top + 1];
    case OPR_LES:
        top--;
        stack[top] = stack[top] < stack[top + 1];
        break;
    case OPR_GEQ:
```

```
                top--;
                stack[top] = stack[top] >= stack[top + 1];
            case OPR_GTR:
                top--;
                stack[top] = stack[top] > stack[top + 1];
                break;
            case OPR_LEQ:
                top--;
                stack[top] = stack[top] <= stack[top + 1];
            } // switch
            break;
        case LOD:
            stack[++top] = stack[base(stack, b, i.l) + i.a];
            break;
        case STO:
            stack[base(stack, b, i.l) + i.a] = stack[top];
            printf("%d\n", stack[top]);
            top--;
            break;
        case CAL:
            stack[top + 1] = base(stack, b, i.l);
            // generate new block mark
            stack[top + 2] = b;
            stack[top + 3] = pc;
            b = top + 1;
            pc = i.a;
            break;
        case INT:
            top += i.a;
            break;
        case JMP:
            pc = i.a;
            break;
        case JPC:
            if (stack[top] == 0)
                pc = i.a;
            top--;
            break;
        } // switch
    }
    while (pc);

    printf("End executing PL/0 program.\n");
} // interpret
```

```
//////////////////////////////////////////////////////////////////////
void main ()
{
    FILE* hbin;
    char s[80];
    int i;
    symset set, set1, set2;

    printf("Please input source file name: "); // get file name to be compiled
    scanf("%s", s);
    if ((infile = fopen(s, "r")) == NULL)
    {
        printf("File %s can't be opened.\n", s);
        exit(1);
    }

    phi = createset(SYM_NULL);
    relset = createset(SYM_EQU, SYM_NEQ, SYM_LES, SYM_LEQ, SYM_GTR, SYM_GEQ,
SYM_NULL);

    // create begin symbol sets
    declbegsys = createset(SYM_CONST, SYM_VAR, SYM_PROCEDURE, SYM_NULL);
    statbegsys = createset(SYM_BEGIN, SYM_CALL, SYM_IF, SYM_WHILE, SYM_NULL);
    facbegsys = createset(SYM_IDENTIFIER, SYM_NUMBER, SYM_LPAREN, SYM_NULL);

    err = cc = cx = ll = 0; // initialize global variables
    ch = ' ';
    kk = MAXIDLEN;

    getsym();

    set1 = createset(SYM_PERIOD, SYM_NULL);
    set2 = uniteset(declbegsys, statbegsys);
    set = uniteset(set1, set2);
    block(set);
    destroyset(set1);
    destroyset(set2);
    destroyset(set);
    destroyset(phi);
    destroyset(relset);
    destroyset(declbegsys);
    destroyset(statbegsys);
    destroyset(facbegsys);
```

```
    if (sym != SYM_PERIOD)
        error(9); // '.' expected.
    if (err == 0)
    {
        hbin = fopen("hbin.txt", "w");
        for (i = 0; i < cx; i++)
            fwrite(&code[i], sizeof(instruction), 1, hbin);
        fclose(hbin);
    }
    if (err == 0)
        interpret();
    else
        printf("There are %d error(s) in PL/0 program.\n", err);
    listcode(0, cx);
} // main    END OF PL0.c
```