Shandong University

School of Computer Science and Technology

PL/0 编译系统实践

Author:
刘建东 Jiandong Liu
Student Number & Class:
201700130011 & Elite Class

Supervisor:
王丽荣教授 Prof. Lirong Wang
Teaching Assistant:
袁娜 Na Yuan

The Experimental Report of Principle of Compiling

2020 年 6 月 8 日

摘　要

本编译系统基于 PL/0 语言文法实现，通过 Makefile 文件链接生成。编译系统一共由三部分组成，分别是词法分析器、语法分析器以及目标指令解释器。

词法分析器通过关键词表的定义从输入代码中精准识别关键字、标识符以及无符号整数，并使用合适的数据结构进行存储。

语法分析器采用递归下降子程序法通过一遍扫描同时完成语法分析、语法错误识别、语法树构建以及目标指令生成。其中目标指令是一种假想的栈式计算机的汇编语言，一共分为八类。

目标指令解释器模拟栈式存储空间，将每一个过程数据段分为了静态部分与动态部分。其中静态部分用于变量存放以及静态链、动态链、返回地址这三个联系单元的存储。

最终编译系统将链接这三大组成部分，实现 PL/0 语言程序的运行与单步调试。

关键词： PL/0 语言，编译原理，词法分析，语法分析，递归下降子程序法，目标指令解释

Abstract

This compiler system is based on the PL/0 language grammar and is generated by the Makefile method. The compiler system consists of three parts, namely lexical analyzer, grammar analyzer and target instruction interpreter.

The lexical analyzer accurately identifies keywords, identifiers, and unsigned integers from the input code through the definition of the keyword table, and uses appropriate data structures for storage.

The grammar analyzer uses the recursive descending subroutine method to complete grammar analysis, grammar error recognition, grammar tree construction, and target instruction generation through one scan. The target instruction is an imaginary stack computer assembly language, which is divided into eight categories.

The target instruction interpreter simulates a stack storage space and divides each process data segment into a static part and a dynamic part. The static part is used for variable storage and storage of the three contact units: static chain, dynamic chain, and return address.

The final compiler system will link these three components to realize the operation and single-step debugging of the PL/0 code.

Key Words: PL/0, principle of compiling, lexical analysis, grammatical analysis, recursive descending subroutine method, target instruction interpretation

目录

1 编译系统概述

1.1 PL/0 语言文法

PL/0 语言可以看成是 Pascal 语言的子集，::= 其目标程序为一种假想的栈式计算机的汇编语言，而其编译程序则是一个编译解释执行系统。

PL/0 语言文法的 BNF 表示

⟨ 程序⟩ ::= ⟨ 分程序⟩ .

⟨ 分程序⟩ ::= [⟨ 常量说明部分⟩][⟨ 变量说明部分⟩][⟨ 过程说明部分⟩]⟨ 语句⟩

⟨ 常量说明部分⟩ ::= const ⟨ 常量定义⟩{,⟨ 常量定义⟩};

⟨ 常量定义⟩ ::= ⟨ 标识符⟩ = ⟨ 无符号整数⟩

⟨ 无符号整数⟩ ::= ⟨ 数字⟩{⟨ 数字⟩}

⟨ 变量说明部分⟩ ::= var ⟨ 标识符⟩{,⟨ 标识符⟩};

⟨ 标识符⟩ ::= ⟨ 字母⟩{⟨ 字母⟩ | ⟨ 数字⟩}

⟨ 过程说明部分⟩ ::= ⟨ 过程首部⟩⟨ 分程序⟩;{⟨ 过程说明部分⟩}

⟨ 过程首部⟩ ::= procedure⟨ 标识符⟩;

⟨ 语句⟩ ::= ⟨ 赋值语句⟩ | ⟨ 条件语句⟩ | ⟨ 当型循环语句⟩ | ⟨ 过程调用语句⟩ | ⟨ 读语句⟩ | ⟨ 写语句⟩ | ⟨ 复合语句⟩ | ⟨ 空⟩

⟨ 赋值语句⟩ ::= ⟨ 标识符⟩ := ⟨ 表达式⟩

⟨ 复合语句⟩ ::= begin⟨ 语句⟩{;⟨ 语句⟩}end

⟨ 条件⟩ ::= ⟨ 表达式⟩⟨ 关系运算符⟩⟨ 表达式⟩ | odd⟨ 表达式⟩

⟨ 表达式⟩ ::= [+|-]⟨ 项⟩{⟨ 加减运算符⟩⟨ 项⟩}

⟨ 项⟩ ::= ⟨ 因子⟩{⟨ 乘除运算符⟩⟨ 因子⟩}

⟨ 因子⟩ ::= ⟨ 标识符⟩ | ⟨ 无符号整数⟩ | (⟨ 表达式⟩)

⟨ 加减运符⟩ ::= + | -

⟨ 乘除运算符⟩ ::= * | /

⟨ 关系运算符 ⟩ ::= = | <> | < | <= | > | >=

⟨ 条件语句⟩ ::= if ⟨ 条件⟩ then ⟨ 语句⟩

⟨过程调用语句⟩ ::= call ⟨标识符⟩

⟨当型循环语句⟩ ::= while ⟨条件⟩ do ⟨语句⟩

⟨读语句⟩ ::= read(⟨标识符⟩{,⟨标识符⟩})

⟨写语句⟩ ::= write(⟨表达式⟩{,⟨表达式⟩})

⟨字母⟩ ::= a | b | c ... x | y | z

⟨数字⟩ ::= 0 | 1 | 2 ... 7 | 8 | 9

PL/0 语言文法的 BNF 表示

表 1: BNF 符号意义

符号	意义
⟨⟩	必选项
[]	可选项
{}	可重复 0 至无数次的项
::=	被定义为

1.2 编译系统组成

PL/0 编译系统一共分为三个模块，即词法分析器、语法分析器、目标指令解释器。其中各模块功能如下：

1. 词法分析器
   - 词法分析，识别关键字、标识符以及无符号整数
2. 语法分析器
   - 语法分析（生成符号表、识别语法错误）
   - 生成语法树
   - 生成目标指令
3. 目标指令解释器
   - 执行目标指令
   - 处理指令运行错误

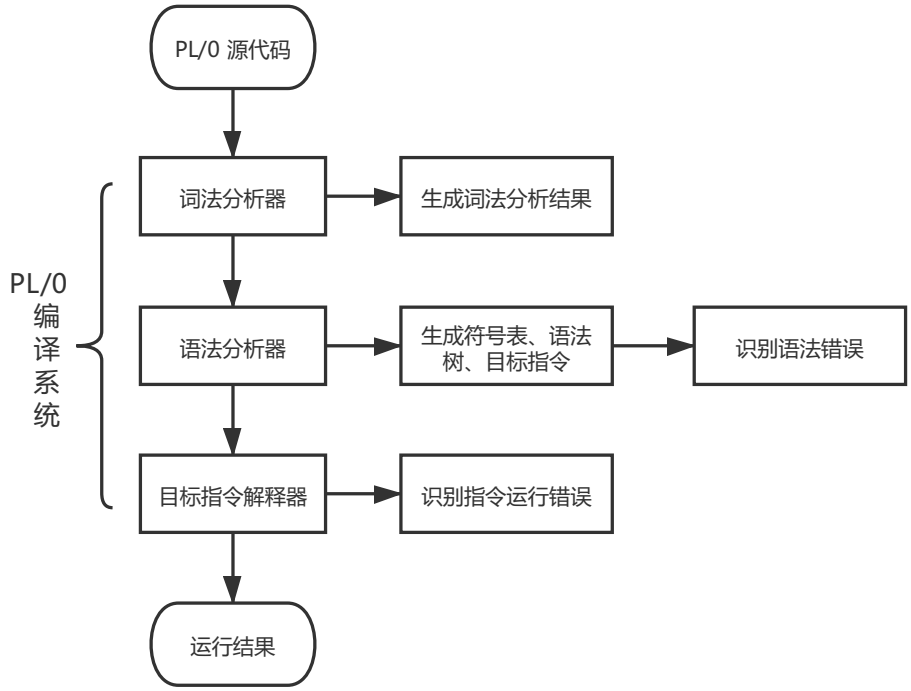

图 1: PL/0 编译系统流程图

1.3 编译系统代码解析

1.3.1 代码结构

编译系统一共由三部分组成，代码结构如下所示：

- Makefile
- 主函数（main.cpp）
- 词法分析器
  - lexical_analyzer.h
  - lexical_analyzer.cpp
- 语法分析器
  - syntax_analyzer.h
  - syntax_analyzer.cpp
  - syntax_tree.py
- 代码解释器
  - code_interpreter.h

· code_interpreter.cpp

其中语法分析器部分的 "syntax_tree.py" 用于将 "syntax_tree.cpp" 所得到的语法树结构转化成语法树。

1.3.2 Makefile

本编译系统的代码由 Makefile 编译链接生成可执行文件"compiler"，其中 Makefile 根据 MacOS 环境编写，可以直接在类 Unix 系统上直接运行，但由于编译器版本不同，在 Linux 上 make 生成可执行文件可能会报一些 warning，但并不影响正确性。

Makefile 文件内容如下所示：

```
# $@: 目标文件, $^: 所有依赖文件, $<: 第一个依赖文件
# 语法规则:
#        targets: prerequisites
#                 command
dest_dir = build
src_dir = src
obj_dir = $(dest_dir)/objects
bin_dir = $(dest_dir)/bin

CC = g++
CFLAGS = -Wall -O3 -std=c++14
CFILES = lexical_analyzer.cpp syntax_analyzer.cpp code_interpreter.cpp main.cpp
ofiles = $(CFILES:%.cpp=$(obj_dir)/%.o)

program = $(bin_dir)/compiler
$(program): $(ofiles)

$(bin_dir)/%:
        @echo ">>> Linking" $@ "<<<"
        @if [ ! -d $(bin_dir) ]; then mkdir -p $(bin_dir); fi;
        $(CC) -o $@ $^
        ln -sf $@ $(notdir $@)

$(obj_dir)/%.o: $(src_dir)/%.cpp
        @echo ">>> Compiling" $< "<<<"
        @if [ ! -d $(obj_dir) ]; then mkdir -p $(obj_dir); fi;
        $(CC) $(CFLAGS) -c $< -o $@

$(obj_dir)/%.o: $(src_dir)/*/%.cpp
        @echo ">>> Compiling" $< "<<<"
        @if [ ! -d $(obj_dir) ]; then mkdir -p $(obj_dir); fi;
        $(CC) $(CFLAGS) -c $< -o $@

.PHONY: clean
clean:
```

```
36          rm -rf $(dest_dir)
37          rm -f compiler
38          rm -f compiler.log
```

在 MacOS 环境下执行 make 操作，可以得到如下所示的编译输出：

```
gene_liu@Gene-Liu Project % make
>>> Compiling src/lexical_analyzer.cpp <<<
g++ -Wall -O3 -std=c++14 -c src/lexical_analyzer.cpp -o build/objects/lexical_analyzer.o
>>> Compiling src/syntax_analyzer.cpp <<<
g++ -Wall -O3 -std=c++14 -c src/syntax_analyzer.cpp -o build/objects/syntax_analyzer.o
>>> Compiling src/code_interpreter.cpp <<<
g++ -Wall -O3 -std=c++14 -c src/code_interpreter.cpp -o build/objects/code_interpreter.o
>>> Compiling src/main.cpp <<<
g++ -Wall -O3 -std=c++14 -c src/main.cpp -o build/objects/main.o
>>> Linking build/bin/compiler <<<
g++ -o build/bin/compiler build/objects/lexical_analyzer.o build/objects/syntax_analyzer.o build/objects/code_interpreter.o build/objects/main.o
ln -sf build/bin/compiler compiler
```

图 2: PL/0 编译系统 Make 编译输出

2 词法分析器

2.1 词法分析

2.1.1 任务说明

在 PL/0 语言中，一共有三种类别的单词，分别是关键字、标识符和常量，此处的关键字也包括算符、界符。因此我们对 PL/0 语言中所有可能出现的单词进行归类编号，即可得到表2所示的单词编号。

表 2: PL/0 编译系统单词编号

编号	关键字	编号	关键字	编号	关键字	编号	关键字
0	.	8	begin	16	<	24	do
1	const	9	end	17	<=	25	read
2	,	10	odd	18	>	26	write
3	;	11	+	19	>=	27	(
4	=	12	-	20	if	28	)
5	var	13	*	21	then	29	标识符
6	procedure	14	/	22	call	30	常量
7	:=	15	<>	23	while		

设置完编号后，我们开始描述 PL/0 编译系统中词法分析所需要完成的任务。我们首先设置三个全局变量，分别为：

- SYM: 存放每个单词的类别，即单词编号

- ID: 存放用户所定义的标识符的值，即标识符字符串的机内表示
- NUM: 存放用户定义的数

定义完上述全局变量后，我们需要完成如下任务：

1. 滤掉单词间的空格。
2. 识别关键字。如果单词是关键字，则将对应的类别放在 SYM 中。
3. 识别标识符。如果单词是标识符，则将标识符对应的类别 29 放入 SYM，将其字符串值放入 ID。
4. 拼数。将数的类别 30 放入 SYM，将其本身的值放在 NUM 中。
5. 拼由两个字符组成的运算符，如: >=、<= 等，识别后将类别存放在 SYM 中。
6. 打印源程序，边读入字符边打印。

2.1.2 分析方法

根据上述定义的任务，我们不难发现词法分析的本质就是将输入的字符串识别成一个个 PL/0 编译系统中所定义的 31 类单词。

存储策略

首先我们需要考虑使用什么方式存储 SYM、ID、NUM，很明显 SYM、ID、NUM 具有大小不断增长的特性，因此有两种实现方法，一是链表，二是线性表。本 PL/0 编译系统采用线性表，即使用 STL 中的 vector 进行数据存储。

滤掉无效单词

观察 PL/0 输入程序，可以发现无效字符主要包括空格、回车以及所有 ASCII 码小于等于 32 的字符。

识别关键字 & 标识符

根据 PL/0 语言文法，关键字与标识符的开头字符均为字母，因此如果当前字符为字母，则不断读取下一个字符，直至下一个字符为无效字符。读取出字符串后，我们需要判断该字符串是否为关键字。

对于关键字的识别，本编译系统使用了 STL 中的 map 来存储上述 29 类关键字，可以在 O(logn) 内判断一个字符串是否是关键字。如果是关键字，则直接存入 SYM，如果不是，则为标识符，需要将字符串存入 ID。

识别数字

PL/0 语言一共有三大类别的单词，其中只有数字不但开头为数字，并且完全由数字组成。因此如果当前字符为数字，则不断读取下一个字符，直至下一个字符不为数字，然后将识别到的数字存入 NUM。

打印源程序

根据上述的读取策略，我们可以得到 SYM、ID、NUM 三个 vector 变量，因此我们设置一个下标遍历 SYM 数组，如果下标指向的值在 [0, 28] 范围内，则将其转换成对应的关键字，如果为 29，则从 ID 中读取对应的标识符，否则从 NUM 中读取对应的数字。随着下标的不断变大，依次输出对应元素即可完成打印源程序的任务。

2.1.3 运行测试

运行测试选取了大量 PL/0 语言程序进行词法分析结果验证，但此处由于篇幅的原因，只列举如下一份 PL/0 语言程序进行测试验证。

PL/0 测试程序

```
const a=10;
var d,e,f;
procedure p;
var g;
begin
  d:=a*2;
  e:=a/3;
  if d<=e then f:=d+e
end;
begin
  read(e,f);
  write(e,f,d);
  call p;
  while odd d do e:=-e+1
end.
```

PL/0 编译系统词法分析结果

```
1  const      7  d          13 procedure  19 begin    25 ;
2  a          8  ,          14 p          20 d        26 e
3  =          9  e          15 ;          21 :=       27 :=
4  10         10 ,          16 var        22 a        28 a
5  ;          11 f          17 g          23 *        29 /
6  var        12 ;          18 ;          24 2        30 3
```

```
31 ;        40 +        49 f        58 d        67 do
32 if       41 e        50 )        59 )        68 e
33 d        42 end      51 ;        60 ;        69 :=
34 <=       43 ;        52 write    61 call     70 -
35 e        44 begin    53 (        62 p        71 e
36 then     45 read     54 e        63 ;        72 +
37 f        46 (        55 ,        64 while    73 1
38 :=       47 e        56 f        65 odd      74 end
39 d        48 ,        57 ,        66 d        75 .
```

2.2 完整代码

随着词法分析的完成，我们需要在"main.cpp" 中完成词法分析的调用过程，具体内容如下述代码所示:

```
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
#include "lexical_analyzer.h"

char indir[] = "data/input/PL0_code";
char outdir1[] = "data/output/PL0_lexicon.txt"; // lexical analysis result

int main(int argc, char * argv[]) {
    int len = strlen(indir);
    if(argc > 1) for(int i = 0; argv[1][i]; i++) indir[len++] = argv[1][i];
    indir[len++] = '.'; indir[len++] = 'i'; indir[len++] = 'n';

    // Output keywords
    // for(int i = 0; i < 29; i++)
    //     std::cout << "| " << i << " | " << keywords[i] << " |" << std::endl;

    // Lexical Analyzer - Part One
    if(!GETSYM(indir)) {std::cout << "Lexical Analyzer Error!" << std::endl; return 0;}
    else std::cout << "Lexical Analyzer Succeed!" << std::endl;
    SYM_OUTPUT(outdir1);
    std::cout << "The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: "
              << SYM.size() << '\n' << std::endl;

    return 0;
}
```

2.2.1 lexical_analyzer.h

```
#ifndef LEXICAL_ANALYZER_H
#define LEXICAL_ANALYZER_H
```

```
3
4  #include <string>
5  #include <vector>
6
7  extern std::string keywords[29];
8  extern const int keytype, tagtype, numtype;
9  extern std::vector<int> SYM;
10 extern std::vector<std::string> ID;
11 extern std::vector<long long> NUM;
12
13 void SYM_OUTPUT(char *outdir);
14 bool GETSYM(char *indir);
15
16 #endif
```

2.2.2 lexical_analyzer.cpp

```
1  #include <iostream>
2  #include <fstream>
3  #include <map>
4  #include "lexical_analyzer.h"
5
6  std::string keywords[29] = {".", "const", ",", ";", "=", "var",
7                              "procedure", ":=", "begin", "end", "odd",
8                              "+", "-", "*", "/", "<>", "<", "<=", ">",
9                              ">=", "if", "then", "call", "while", "do",
10                             "read", "write", "(", ")"};
11 const int keytype = 28, tagtype = keytype + 1, numtype = keytype + 2;
12
13 std::map<std::string, int> mp;
14 std::vector<int> SYM;
15 std::vector<std::string> ID;
16 std::vector<long long> NUM;
17
18 void SYM_OUTPUT(char *outdir) {
19     std::ofstream output(outdir);
20     int p2 = 0, p3 = 0, sz = SYM.size();
21     for(int p1 = 0; p1 < sz; p1++){
22         if(SYM[p1] <= keytype) output << keywords[SYM[p1]] << std::endl;
23         else if(SYM[p1] == tagtype) output << ID[p2++] << std::endl;
24         else output << NUM[p3++] << std::endl;
25     }
26     output.close();
27 }
28
29 bool GETSYM(char *indir) {
30     // init
31     std::ifstream input(indir);
```

```
32      mp.clear(); SYM.clear(); ID.clear(); NUM.clear();
33      for(int i = 0; i < 30; i++)
34          mp[keywords[i]] = i;
35      // lexical analysis
36      std::string line;
37      while(getline(input, line)){
38          for(int i = 0; line[i]; i++){
39              if(line[i] == ' ' || line[i] == '\n' || (int)line[i] <= 32){
40                  continue;
41              }
42              else if(line[i] >= '0' && line[i] <= '9'){
43                  // identify number
44                  long long number = line[i]-'0';
45                  while(line[i+1] && line[i+1] >= '0' && line[i+1] <= '9'){
46                      number = number * 10 + (int)(line[++i] - '0');
47                  }
48                  SYM.push_back(numtype);
49                  NUM.push_back(number);
50              }
51              else if(line[i] >= 'a' && line[i] <= 'z'){
52                  // identify variable
53                  std::string tmp = "\0";
54                  tmp += line[i];
55                  while(line[i+1] && ((line[i+1] >= '0' && line[i+1] <= '9')
56                                      || (line[i+1] >= 'a' && line[i+1] <= 'z')))
57                  {
58                      tmp += line[++i];
59                  }
60                  if(mp.find(tmp) != mp.end()) SYM.push_back(mp[tmp]);
61                  else SYM.push_back(tagtype), ID.push_back(tmp);
62              }
63              else{
64                  std::string tmp = "\0";
65                  tmp += line[i];
66                  while((mp.find(tmp+line[i+1]) != mp.end() || mp.find(tmp) == mp.end())
67                                      && line[i+1] && line[i+1] != ' '
68                                      && line[i+1] != '\n' && (int)line[i+1] != 13)
69                  {
70                      tmp += line[++i];
71                  }
72                  if(mp.find(tmp) != mp.end()) SYM.push_back(mp[tmp]);
73                  else {input.close(); return 0;}
74              }
75          }
76      }
77      input.close();
78      return 1;
79  }
```

3 语法分析器

PL/0 语法分析器使用递归下降子程序，采用一遍扫描的方法，同时完成语法分析（生成符号表、识别语法错误）、生成语法树、生成目标指令三大任务，但由于三大任务的要求以及实现方法、生成结果均不同，因此我们进行分开说明。

3.1 语法分析

3.1.1 任务说明

语法分析作为语法分析器三大任务之一，主要需要完成符号表的生成与语法错误的识别。

生成符号表

生成符号表就是对 PL/0 代码中每个过程（包括主程序）的说明对象造名字表，填写所在层次、标识符的属性和分配的相对地址。这里有两点需要注意：

1. 主程序是 0 层，在主程序中定义的过程是 1 层，随着嵌套深度增加而层次数增大。
2. PL/0 最多允许 3 层。
3. 标识符的属性不同则填写的信息不同。

符号表需要存放在一维数组 TABLE 中，TX 为指针，数组元素为结构体类型数据。LEV 给出层次，DX 给出每层的局部量的相对地址，每说明完一个变量后 DX 加 1。接下来我们以一段 PL/0 代码进行举例说明。

- 示例代码

```
const a = 35, b = 49;
var c, d, e;
procedure p;
  var g;
```

- 符号表

TX0 →

NAME: a	KIND: CONSTANT	VAL: 35	
NAME: b	KIND: CONSTANT	VAL: 49	
NAME: c	KIND: VARIABLE	LEVEL: LEV	ADR: DX
NAME: d	KIND: VARIABLE	LEVEL: LEV	ADR: DX+1
NAME: e	KIND: VAEIABLE	LEVEL: LEV	ADR: DX+2
NAME: p	KIND: PROCEDURE	LEVEL: LEV	ADR:
NAME: g	KIND: VARIABLE	LEVEL: LEV+1	ADR: DX
∘	∘	∘	∘
∘	∘	∘	∘
∘	∘	∘	∘

TX1 →

图 3: TABLE 示例

编译系统内置初值 DX=3，用于存放每个过程的静态链、动态链与返回地址。（此部分内容将在目标指令解释器部分进行详细说明）

识别语法错误

对于一份存在 PL/0 语法错误的代码，语法分析器需要指出其语法错误类型以及错误位置。

3.1.2 递归下降子程序

本编译系统中的语法分析器通过递归下降子程序完成，因此我们首先需要根据之前给出的 PL/0 语言文法的 BNF 表示来确定各个子程序。

```
/*———————————— Procedure Function ————————————*/
void Procedure();
void SubProcedure(int, int);

/*———————————— Specification Function ————————————*/
void ConstDefinition(int);
void ConstSpecification(int);
void VariableSpecification(int);
void ProcedureSpecification(int);

/*———————————— Statement Function ————————————*/
void Statement(int);
void AssignStatement(int);
void ConditionStatement(int);
void LoopStatement(int);
void CallStatement(int);
void ReadStatement(int);
void WriteStatement(int);
void CompoundStatement(int);

/*———————————— Other Functions ————————————*/
void Item(int);
void Factor(int);
void Condition(int);
void Expression(int);
```

确定完各个子程序名后，我们就可以根据 PL/0 语言文法的每一个 BNF 式子搭建出语法分析的大致框架。我们以如下这条 BNF 式子为例来说明子程序内部的书写方式。

⟨语句⟩::=⟨赋值语句⟩|⟨条件语句⟩|⟨当型循环语句⟩|⟨过程调用语句⟩|⟨读语句⟩|⟨写语句⟩|⟨复合语句⟩|⟨空⟩

```
void Statement() {
    if(SYM[p1] == tagtype) AssignStatement();
```

```
    else if(SYM[p1] == 20) ConditionStatement();
    else if(SYM[p1] == 23) LoopStatement();
    else if(SYM[p1] == 22) CallStatement();
    else if(SYM[p1] == 25) ReadStatement();
    else if(SYM[p1] == 26) WriteStatement();
    else if(SYM[p1] == 8) CompoundStatement();
}
```

其中 p1 表示当前所识别的单词，因此我们可以根据 p1 所指代的单词编号来确定具体是哪一条语句，并进入对应的子程序进行处理。

注意：此处的示例仅为递归下降子程序的大致框架，非最终版本。

3.1.3 生成符号表

存储方式

生成符号表的第一步是确定存储方式。不难发现，符号表属于一个三级结构，即“完整符号表”→“过程符号表”→“表项”，其中各过程符号表由其中的 procedure 所链接，因此每一个“procedure 表项”需要记录其所链接的“过程符号表”，而每一个“过程符号表”也需要记录其“前驱过程符号表”。

由此我们可以使用链表或数组模拟链表来存储“完整符号表”，本编译系统使用 vector 来模拟链表进行“完整符号表”的存储，具体结构体定义如下所示：

```
struct TableEntry {
    std::string NAME, KIND;
    int VAL, LEVEL, ADR, NXT;
    TableEntry() { ADR = NXT = -1; }
    TableEntry(std::string t1, int t2, int t3, int t4, int t5, int t6) {
        NAME = t1; KIND = t2; VAL = t3; LEVEL = t4; ADR = t5; NXT = t6;
    }
};

struct Table {
    int PreTable, PreEntry, VNUM;
    std::vector<TableEntry> table;
    std::map<std::string, std::pair<int, int> > mp1;
    std::map<std::string, int> mp2;
    Table() {
        PreTable = PreEntry = -1;
        VNUM = 0; // 记录变量个数
        table.clear();
        mp1.clear(); // 用于记录常量-(数值,0), 变量-(相对地址,1)
        mp2.clear(); // 用于记录过程-入口地址
    }
    Table(int t1, int t2) : PreTable(t1), PreEntry(t2) {}
};

```

```
25 extern std::vector<Table> TABLE;
```

注意：Table 结构体中的 mp1、mp2 用于后续的目标指令生成，与符号表的构建无关。

实现方法

仔细分析之前的任务说明，可以发现符号表构造的关键就在于识别关键字 "const" 、"var" 、 "procedure" 。其中 "const" 、 "var" 的识别只需往其对应的 Table 结构体中添加表项，而 "procedure" 的识别则需要创建新的 Table 结构体，并在整个 "procedure" 过程识别结束后，返回定义该 "procedure" 过程的 Table 结构体中，即回溯。

由此我们只需要在“常量/变量/过程说明部分”对应的子程序中增加代码即可实现符号表的构造，具体代码如下所示。

```
1  void ConstDefinition(int fa) {
2      /* ... */
3      if(SYM[p1] == tagtype) {
4          entry.NAME = ID[p2++];
5          entry.KIND = CONSTANT;
6          p1++;
7      }
8      /* ... */
9      if(SYM[p1] == numtype) {
10         entry.VAL = NUM[p3++];
11         entry.LEVEL = LEV;
12         /* ... */
13         TABLE[TX].mp1[entry.NAME] = std::make_pair(entry.VAL, 0);
14         TABLE[TX].table.push_back(entry);
15         p1++;
16     }
17     /* ... */
18 }
19
20 void RecognizeFlag(std::string flag, int fa, int op = 0) {
21     TableEntry entry;
22     entry.NAME = ID[p2++];
23     entry.KIND = flag;
24     entry.LEVEL = LEV;
25     entry.ADR = ADR++;
26     if(op){
27         /*
28             Recognize procedure
29             ...
30         */
31         TABLE[TX].mp2[entry.NAME] = CODE.size()+1;
32     }
33     else{
34         /*
35             Recognize variable
```

```
36                 ...
37             */
38             TABLE[TX].mp1[entry.NAME] = std::make_pair(entry.ADR, 1);
39             // the number of variable in TABLE[TX] ++
40             TABLE[TX].VNUM++;
41         }
42         TABLE[TX].table.push_back(entry);
43         p1++;
44         /* ... */
45     }
46
47     void VariableSpecification(int fa) {
48         /* ... */
49         RecognizeFlag(VARIABLE, id);
50
51         while(SYM[p1] == 2) {
52             p1++;
53             /* ... */
54             RecognizeFlag(VARIABLE, id);
55         }
56         /* ... */
57     }
58
59     void ProcedureSpecification(int fa) {
60         /* ... */
61         if(SYM[p1] == tagtype){
62             RecognizeFlag(PROCEDURE, id, 1);
63             TABLE.push_back(Table());
64             TABLE[TABLE.size()-1].PreTable = TX;
65             TABLE[TABLE.size()-1].PreEntry = TABLE[TX].table.size()-1;
66             TABLE[TX].table[TABLE[TX].table.size()-1].NXT = TABLE.size()-1;
67             TX = TABLE.size()-1, LEV++, ADR = DX;
68         }
69         /* ... */
70         SubProcedure(id);
71         /* ... */
72         LEV--;  // backtracking
73         int PreEntry = TABLE[TX].PreEntry;
74         TX = TABLE[TX].PreTable;
75         ADR = ADR - 1 + TABLE[TX].table[PreEntry].ADR;
76         TABLE[TX].table[PreEntry].ADR = ADR++;
77         /* ... */
78         while(SYM[p1] == 6) ProcedureSpecification(id);
79     }
```

注意：上述代码为删减后的版本，仅保留了构造符号表所涉及的部分。

3.1.4 识别语法错误

识别语法错误，主要包括两部分内容：

1. 指出出错位置
2. 标记语法错误类型

对于出错位置，我们可以直接使用出现语法错误时 p1 的值来指代，其中 p1 表示语法分析时执行的 SYM 下标。而针对错误类型，我们则需要在每一个递归下降子程序中标记上对应的错误类型。

由此我们定义了如下两部分代码用于输出错误位置以及错误类型。

```
#define Judge(i, NAME) if(SYM[p1] == i) {p1++;} \
                       else {return Error(NAME);}

void Error(std::string error) {
    std::cout << "error: expected " << error << " at the position "
    std::cout << p1 << "." << std::endl;
    std::cout << "Syntax Analyzer Error!" << std::endl;
    exit(0);
}
```

接下来我们以常量说明部分对应的 BNF 式子来进一步说明错误类型的识别方法。

⟨ 常量说明部分⟩ ::= const ⟨ 常量定义⟩{,⟨ 常量定义⟩};

```
void ConstSpecification(int fa) {
    Judge(1, "'const' in const declaration");
    ConstDefinition(id);
    /* ... */

    while(SYM[p1] == 2){
        p1++;
        ConstDefinition(id);
        /* ... */
    }
    Judge(3, "';' in const declaration");
}
```

观察上述代码可以发现，我们只需对关键字进行错误类型说明，其它子程序同理。

3.1.5 运行测试

PL/0 测试程序

```
var x, y, z, q, r, n, f;

procedure multiply;
```

```
var a, b;
begin
  a := x;
  b := y;
  z := x*y
end;

procedure divide;
var w;
begin
  r := x;
  q := 0;
  w := y;
  while w <= r do w := 2 * w;
  while w > y do
  begin
    q := 2 * q;
    w := w / 2;
    if w <= r then
    begin
      r := r - w;
      q := q + 1
    end
  end
end;

procedure gcd;
var  g;
begin
  f := x;
  g := y;
  while f <> g do
  begin
    if f < g then g := g - f;
    if g < f then f := f - g
  end;
  z := f
end;

procedure fact;
begin
  if n > 1 then
  begin
    f := n * f;
    n := n - 1;
    call fact
  end
end;

begin
  read(x);read(y); call multiply;write(z);
  read(x);read(y); call divide; write(q); write(r);
  read(x);read(y); call gcd; write(z);
  read(n); f := 1; call fact; write(f)
end.
```

对应符号表

```
------------------------------------------------------------------------------------------------------------
TX: 0         NAME: x                 KIND: VARIABLE              LEVEL: 0              ADR: 3
              NAME: y                 KIND: VARIABLE              LEVEL: 0              ADR: 4
              NAME: z                 KIND: VARIABLE              LEVEL: 0              ADR: 5
              NAME: q                 KIND: VARIABLE              LEVEL: 0              ADR: 6
              NAME: r                 KIND: VARIABLE              LEVEL: 0              ADR: 7
              NAME: n                 KIND: VARIABLE              LEVEL: 0              ADR: 8
              NAME: f                 KIND: VARIABLE              LEVEL: 0              ADR: 9
              NAME: multiply          KIND: PROCEDURE             LEVEL: 0              ADR:
------------------------------------------------------------------------------------------------------------
TX: 1         NAME: a                 KIND: VARIABLE              LEVEL: 1              ADR: 3
              NAME: b                 KIND: VARIABLE              LEVEL: 1              ADR: 4
------------------------------------------------------------------------------------------------------------
TX: 0         NAME: divide            KIND: PROCEDURE             LEVEL: 0              ADR:
------------------------------------------------------------------------------------------------------------
TX: 2         NAME: w                 KIND: VARIABLE              LEVEL: 1              ADR: 3
------------------------------------------------------------------------------------------------------------
TX: 0         NAME: gcd               KIND: PROCEDURE             LEVEL: 0              ADR:
------------------------------------------------------------------------------------------------------------
TX: 3         NAME: g                 KIND: VARIABLE              LEVEL: 1              ADR: 3
------------------------------------------------------------------------------------------------------------
TX: 0         NAME: fact              KIND: PROCEDURE             LEVEL: 0              ADR:
------------------------------------------------------------------------------------------------------------
TX: 4
------------------------------------------------------------------------------------------------------------
```

图 4: PL/0 测试程序对应符号表

识别语法错误测试

我们将上述 PL/0 测试程序中的第一行改为如下形式，即缺少了末尾的 ";" 。

```
var x, y, z, q, r, n, f
```

运行编译系统，可得到如下报错信息：

```
gene_liu@Gene-Liu Project % ./compiler
-------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 260

-------------------------------- PART TWO ---------------------------------
error: expected ';' in variable declaration at the position 14.
Syntax Analyzer Error!
```

图 5: PL/0 语法错误测试 1

继续测试，我们将上述 PL/0 测试程序中的第三行改为如下形式，依然是缺少了末尾的 ";" 。

```
procedure multiply
```

运行编译系统，可以发现报错信息发生了变化：

```
gene_liu@Gene-Liu Project % ./compiler
-------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 261

-------------------------------- PART TWO ---------------------------------
error: expected ';' in procedure declaration at the position 17.
Syntax Analyzer Error!
```

图 6: PL/0 语法错误测试 2

3.2 语法树

3.2.1 画图方法

语法树画图一共由两部分组成：

1. 语法分析时得到语法树结构
2. 使用 python 读入树结构并画出语法树

语法树结构

首先我们需要定义一下存图方式。计算机中常见的存图方法有邻接矩阵、邻接表，空间复杂度分别是 $O(n^2)$ 和 $O(n+m)$。由于语法树的点数较多，因此选择使用 C++ 中的 *vector* 数据结构实现邻接表存图。

```
std::vector<std::string> label;
std::vector<std::vector<int> > G;
```

在上述的存图结构中，*label* 表示每一个节点的名称，*G* 存取了每一个节点的邻接节点，空间复杂度为 $O(n+m)$。

接下来我们需要考虑如何在递归下降子程序中构建出语法树。由于语法树中每一个节点只会有一个父节点，因此可以在每一个子程序中记录父节点的编号，用于将当前节点保存为父节点的邻接点。

```
std::vector<int> empty_vec;

int Graph_init(std::string name, int fa){
    // Graph Construct
    label.push_back(name);
    G.push_back(empty_vec);
    int id = label.size()-1;
    G[fa].push_back(id);
    return id;
}
```

上述代码为生成一个名称为 "name" 的新节点，且该节点的父节点为 *fa*。接下来我们可以用该函数进行语法树的构建，我们以 *Procedure*() 和 *SubProcedure*(*int*) 函数进行举例。

由于 *Procedure*() 是整个语法分析的入口函数，因此我们不需要在该函数中传入父节点参数，即往 *G* 中加入了第一个叫“程序”的节点，并令 *id* 为当前节点编号，且在运行 *SubProcedure*(*int*) 函数时，传入了 *id* 作为该函数的父节点。

```
void Procedure() {
    // Graph Construct
    label.push_back("程序");
    G.push_back(empty_vec);
    int id = label.size()-1;
    /* ... */
    SubProcedure(id, 1);
    if(SYM[p1] == 0 && p1 == sz) Graph_init(keywords[0], id);
    else Error("'.' in the end of main procedure");
}
```

接下来我们再看看 *Subprocedure*(*int*) 函数。进入该函数后的第一件事就是创建一个新的名叫“分程序”的节点，并将父节点连向当前节点。之后依次识别常量、变量、过程、语句即可。

```
void SubProcedure(int fa) {
    int id = Graph_init("分程序", fa);
    if(SYM[p1] == 1) ConstSpecification(id);
    if(SYM[p1] == 5) VariableSpecification(id);
    if(SYM[p1] == 6) ProcedureSpecification(id);
```

```
    /* ... */
    Statement(id);
}
```

最后，我们需要将得到的图结构输出到文件，具体函数如下所示。

```
void GRAPH_OUTPUT(char *outdir) {
    std::ofstream output(outdir);
    int n = label.size();
    output << n << std::endl;
    for(int i = 0; i < n; i++)
        output << label[i] << " \n"[i==n-1];
    for(int i = 0; i < n; i++){
        int sz = G[i].size();
        if(sz == 0) output << std::endl;
        for(int j = 0; j < sz; j++)
            output << G[i][j] << " \n"[j==sz-1];
    }
    output.close();
}
```

python 画图

得到上述生成的图结构后，我们就可以读取图结构并用 *python* 中的 *graphvis* 库来画出语法树，一共分为两个步骤。

读出树结构

首先是第一步，我们从文件中读出树结构。下述代码中 n 表示树中节点个数，*lable* 表示每个节点的名称，G 存储图结构。

```
def data_acquire(path):
    # read file
    fo = open(path, "r+")
    data = fo.read().split("\n")

    # acquire data
    n = int(data[0])
    lable = data[1].split(" ")

    # modify data format
    for i in range(0, len(lable)):
        if lable[i] == "<>":
            lable[i] = "\<\>"

    G = []
    for row in range(2, len(data)):
        G.append(data[row].split(" "))
    return n, lable, G
```

dfs 遍历语法树

接下来是第二步，我们 dfs 遍历语法树，将树中的信息输入到画图库中，用于后续的语法树生成。在下述代码中，x 表示当前节点，fa 表示父节点。

```
def dfs(x, fa, lable, G, graph):
    graph.node(str(x), lable[x])
    if fa != -1:
        graph.edge(str(fa), str(x))
    for y in G[x]:
        if y == "":
            break
        dfs(int(y), x, lable, G, graph)
```

3.2.2 运行测试

此部分测试我们仍然以 3.1.5 中所给出的 PL/0 测试程序为例，给出其对应的语法树结构以及语法树图。

语法树结构

```
digraph G {
  node [color=white style=filled]
  edge [color=grey style=filled]
  bgcolor=white
  rankdir=LR
  style=filled
  0 [label="程序"]
  1 [label="分程序"]
  0 -> 1
  2 [label="变量说明部分"]
  1 -> 2
  3 [label=var]
  2 -> 3
  4 [label=x]
  2 -> 4
  5 [label=","]
  2 -> 5
  6 [label=y]
  2 -> 6
  7 [label=","]
  2 -> 7
  8 [label=z]
  ...
  506 [label="写语句"]
  505 -> 506
  507 [label=write]
  506 -> 507
  508 [label="("]
  506 -> 508
  509 [label="表达式"]
  506 -> 509
  510 [label="项"]
  509 -> 510
  511 [label="因子"]
  510 -> 511
  512 [label=f]
  511 -> 512
  513 [label=")"]
  506 -> 513
  514 [label=end]
  385 -> 514
  515 [label="."]
  0 -> 515
}
```

语法树图

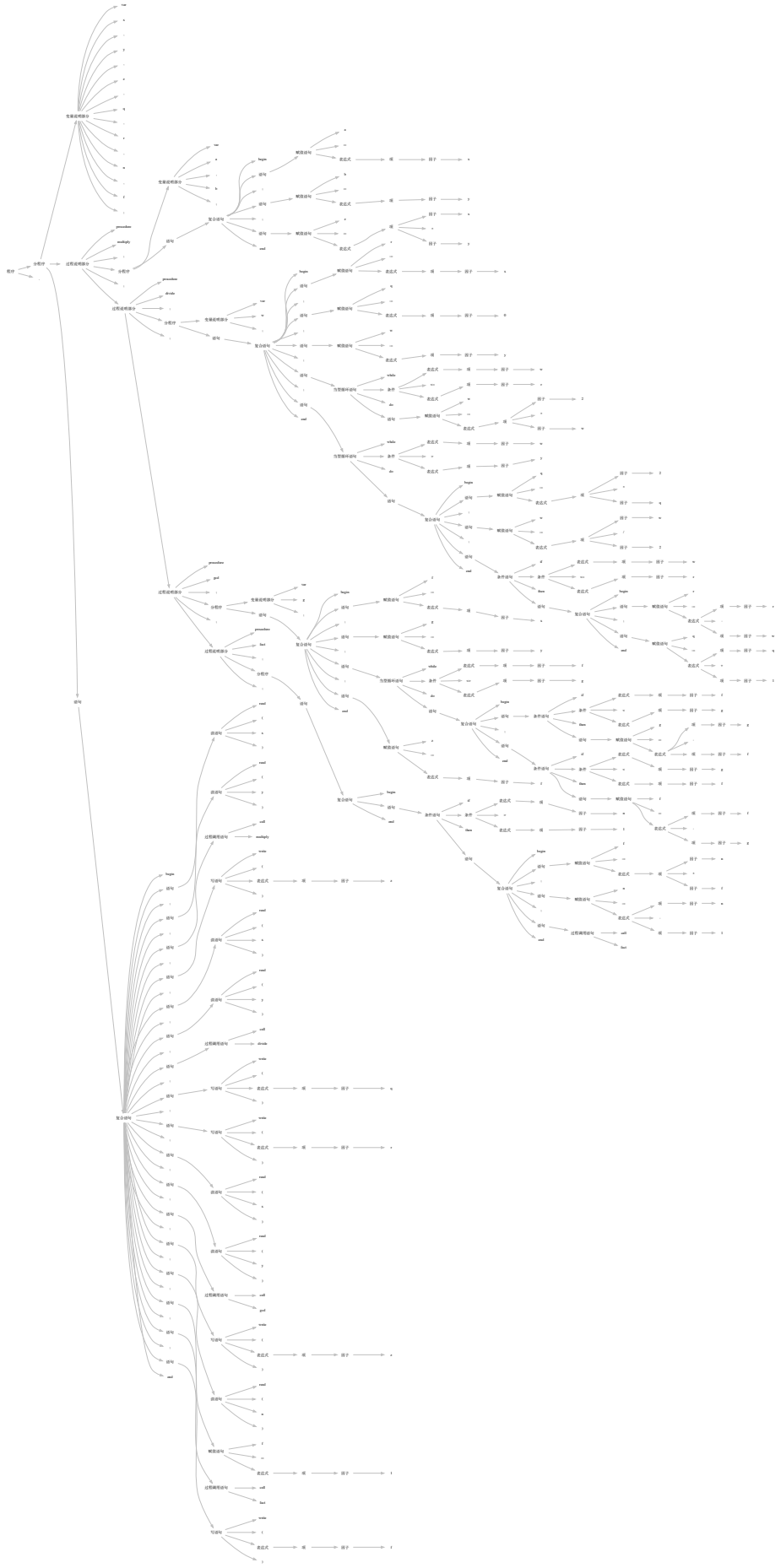

图 7: PL/0 测试程序语法树

3.3 目标指令

3.3.1 任务说明

对语句逐句分析，语法正确则生成目标代码，当遇到标识符的引用则去查 TABLE 表。PL/0 语言的目标指令是一种假想的栈式计算机的汇编语言，其格式如下所示：

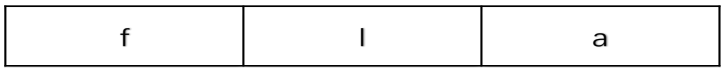

图 8: PL/0 语言指令格式

其中 f 代表功能码，l 代表层次差，a 代表位移量。目标指令一共有 8 类：

表 3: PL/0 语言指令类型

指令类型	具体功能
LIT 0 *a*	将常量 *a* 放到运行栈栈顶
OPR 0 *a*	栈顶与次栈顶执行运算，结果存放在次栈顶，*a* 表示执行的运算类型（*a* 共有 17 种取值），*a* 为 0 时即退出数据区
LOD l a	将变量放到运行栈栈顶（*a* 为变量在声明层中的相对地址，*l* 为调用层与声明层的层差值）
STO l a	将运行栈栈顶内容存入变量（*a* 为相对地址，*l* 为层差）
CAL l a	调用过程（*a* 为被调用过程的目标程序的入口地址，*l* 为层差）
INT 0 *a*	为被调用的过程（或主程序）在运行栈种开辟数据区，即运行栈栈顶指针增加 *a*
JMP 0 *a*	无条件转移到指令地址 *a*
JPC 0 *a*	条件转移到指令地址 *a*，即当栈顶的布尔值为假时，转向地址 *a*，否则顺序执行

接下来给出一个 PL/0 源程序以及其对应的目标指令作为范例。

PL/0 范例程序

```
const a=10;
var b,c;
procedure p;
begin
  c:=b+a
end;
begin
  read(b);
  while b<>0 do
    begin
      call p;
      write(2*c);
      read(b);
    end
end.
```

范例程序对应的目标指令

```
1  1 jmp 0 8
2  2 int 0 3
3  3 lod 1 3
4  4 lit 0 10
5  5 opr 0 2
6  6 sto 1 4
7  7 opr 0 0
8  8 int 0 5
9  9 opr 0 16
10 10 sto 0 3
11 11 lod 0 3
12 12 lit 0 0
13 13 opr 0 9
14 14 jpc 0 24
15 15 cal 0 2
16 16 lit 0 2
17 17 lod 0 4
18 18 opr 0 4
19 19 opr 0 14
20 20 opr 0 15
21 21 opr 0 16
22 22 sto 0 3
23 23 jmp 0 11
24 24 opr 0 0
```

3.3.2 翻译模式

在生成具体目标指令前，我们需要进一步确定上述 8 类指令在栈尺度上的操作，以及各类语句所对应的目标指令。

指令微操作

- *LIT* 0 *a*

```
sp <- sp + 1;
stack[sp] <- a;
```

- *LOD l a*

```
sp <- sp + 1;
stack[sp] <- stack[base(l) + a];
```

- *STO l a*

```
stack[base(l)+a] <- stack[sp];
sp <- sp - 1;
```

- *CAL l a*

```
stack[sp+1] <- base(l);
stack[sp+2] <- ba;
stack[sp+3] <- pc;
ba <- sp + 1;
pc <- a;
```

- *INT* 0 *a*

```
sp <- sp + a;
```

- *JMP* 0 *a*

```
pc <- a;
```

- *JPC* 0 *a*

```
if stack[sp] == 0:
    pc <- a;
    sp <- sp - 1;
```

- *OPR* 0 0 (*RETURN*)

```
stack[sp+1] <- base(l);
sp <- bp-1;
bp <- stack[sp+2];
pc <- stack[sp+3];
```

- *OPR* 0 1 (*NEG*)

```
stack[sp] <- -stack[sp];
```

- *OPR* 0 2 (*ADD*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] + stack[sp+1];
```

- *OPR* 0 3 (*SUB*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] - stack[sp+1];
```

- *OPR* 0 4 (*MUL*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] * stack[sp+1];
```

- *OPR* 0 5 (*DIV*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] / stack[sp+1];
```

- *OPR* 0 6 (*ODD*)

```
stack[sp] <- stack[sp] % 2;
```

- *OPR* 0 7 (*MOD*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] % stack[sp+1];
```

- *OPR* 0 8 (*EQL*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] == stack[sp+1];
```

- *OPR* 0 9 (*NEQ*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] != stack[sp+1];
```

- *OPR* 0 10 (*LSS*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] < stack[sp+1];
```

- *OPR* 0 11 (*LEQ*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] <= stack[sp+1];
```

- *OPR* 0 12 (*GTR*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] > stack[sp+1];
```

- *OPR* 0 13 (*GEQ*)

```
sp <- sp - 1;
stack[sp] <- stack[sp] >= stack[sp+1];
```

- *OPR* 0 14 (*OUT*)

```
print (stack[sp]);
sp <- sp - 1;
```

- *OPR* 0 15 (*OUL*)

```
print ('\n');
```

- *OPR* 0 16 (*IN*)

```
sp <- sp + 1;
scan (stack[sp]);
```

注意：上述微操作中的 sp、pc、ba、base(l) 分别表示栈顶指针、指令计数器、当前数据区基地址、l 层数据区基地址。生成指令时需要先理解各指令的微操作，后续解释目标指令时也将根据上述微操作进行实现。

语句翻译模式

由于 PL/0 语言中只有 *if ... then* 与 *while ... do* 两类语句涉及到了跳转地址回填，因此我们将针对这两种语句给出相应的翻译模式。

- *if ... then*

```
        <condition>
        JPC addr
        <statement>
addr:   ...
```

- *while ... do*

```
addr2:  <condition>
        JPC addr1
        <statement>
        JMP addr2
addr1:  ...
```

3.3.3 指令生成

确定完翻译模式后，我们就可以依照着任务说明中给出的程序范例进行大致模拟，体会到 PL/0 语言目标指令与汇编语言之间的相似性，然后再对于每一个子程序思考其所对应的目标指令。

目标指令存储

本编译系统对于指令定义了 Instruction 类，并将所有指令用 vector 进行存储。

```
struct Instruction {
    std::string f;
    int l, a;
    Instruction() { f = "", l = a = 0; }
```

```
    Instruction(std::string f1, int l1, int a1) {
        f = f1; l = l1; a = a1;
    }
};

extern std::vector<Instruction> CODE;
```

目标指令生成

目标指令生成主要是根据每一个递归下降子程序，考虑在该子程序中可能会生成的目标指令，然后进行代码编写，本质上来说是一个模拟的过程。

在模拟过程中，我们会频繁查询一个标识符的类型、层差、地址等信息，因此首先需要定义一个 "SearchPosition" 函数，用于给出引用的标识符的信息。具体的说，如果标识符是常量，则返回 $(-1,val)$；如果是变量，则返回 (l,a)；如果是过程则返回 $(l,addr)$。此处的 val 表示常量大小，l 表示层差，a 表示变量所在数据区的地址，$addr$ 表示过程的指令入口地址。

```
std::pair<int,int> SearchPosition(std::string NAME, int op = 0) {
    /*
        constant: (-1, val), variable: (l, a), procedure: (l, addr)
        variable name should differ from const name
        procedure name could be same with variable or const name
    */
    int now = TX, cnt = 0;
    while(now != -1) {
        if(!op && TABLE[now].mp1.find(NAME) != TABLE[now].mp1.end()){
            if(TABLE[now].mp1[NAME].second == 0)
                return std::make_pair(-1, TABLE[now].mp1[NAME].first);
            else
                return std::make_pair(cnt, TABLE[now].mp1[NAME].first);
        }
        // search procedure
        if(op && TABLE[now].mp2.find(NAME) != TABLE[now].mp2.end())
            return std::make_pair(cnt, TABLE[now].mp2[NAME]);
        now = TABLE[now].PreTable, cnt++;
    }
    return std::make_pair(-3, -1);
}
```

上述的查询函数主要通过之前在符号表中所定义的 mp1、mp2 实现，可以在 O(logn) 内查找对应的常量、变量、过程。此处回顾一下先前 Table 表的定义，其中 mp1 变量记录 "常量名 → (数值,0)" 与 "变量名 → (相对地址,1) " ，而 mp2 则记录 "过程名 →

入口地址" 。

```
struct Table {
    /*...*/
    std::map<std::string, std::pair<int, int>> mp1;
    std::map<std::string, int> mp2;
    Table() {
        /*...*/
        mp1.clear(); // 用于记录常量-(数值,0), 变量-(相对地址,1)
        mp2.clear(); // 用于记录过程-入口地址
    }
    Table(int t1, int t2) : PreTable(t1), PreEntry(t2) {}
};
```

接下来我们需要针对 PL/0 中每一个子程序给出其对应的用于生成目标指令的代码，但由于 PL/0 中子程序数量较多，因此此处仅以三个子程序为例来说明这个模拟过程，其余子程序的代码可在最后的完整代码中查看。

1. 条件语句

```
void ConditionStatement(int fa) {
    int id = Graph_init("条件语句", fa);
    Judge(20, "'if' in condition statement");
    Graph_init(keywords[20], id);
    Condition(id);
    CODE.push_back(Instruction("jpc", 0, 0));
    int index = CODE.size();

    Judge(21, "'then' in condition statement");
    Graph_init(keywords[21], id);
    Statement(id);
    CODE[index-1].a = CODE.size()+1;
}
```

可以看到，此处的条件语句，采取了先忘全局变量 CODE 中插入一条 jpc 指令，待语句部分的指令填充完毕回溯回来后，再回填 jpc 的跳转地址。

2. 循环语句

```
void LoopStatement(int fa) {
    int id = Graph_init("当型循环语句", fa);
    Judge(23, "'while' in loop statement");
    Graph_init(keywords[23], id);
    int index1 = CODE.size();
    Condition(id);
```

```
    CODE.push_back(Instruction("jpc", 0, 0));
    int index2 = CODE.size();

    Judge(24, "'do' in loop statement");
    Graph_init(keywords[24], id);
    Statement(id);
    CODE.push_back(Instruction("jmp", 0, index1+1));
    CODE[index2-1].a = CODE.size()+1;
}
```

根据之前循环语句的翻译模式，不难发现其目标指令中有两条跳转语句，因此我们在代码实现层面也需要加入两条跳转指令，并且对于第一条有条件的跳转指令需要在循环语句结束后进行跳转地址回填。

3. 过程调用语句

```
void CallStatement(int fa) {
    int id = Graph_init("过程调用语句", fa);
    Judge(22, "'call' in call procedure statement");
    Graph_init(keywords[22], id);
    if(SYM[p1] == tagtype) {
        // acquire the position of precedure
        std::pair<int, int> pos = SearchPosition(ID[p2], 1);
        // pos must be precedure
        if(pos.first >= 0) CODE.push_back(Instruction("cal", pos.first, pos.second));
        else Error("declared 'procedure name' in call procedure statement");

        Graph_init(ID[p2++], id), p1++;
    }
    else return Error("declared 'procedure name' in call procedure statement");
}
```

过程调用语句的目标指令较为简单，只需要用之前介绍的 SearchPosition 函数进行过程名寻找，再将其得到的层差与地址填入 cal 指令中即可。

基本上各子程序中的目标指令都只需要根据其子程序功能，以及一些汇编知识，即可完成代码实现。因此此处不再过多进行讲解，完整代码将在后续内容中给出。

3.3.4 运行测试

该部分我们仍然使用之前在 3.1.5 中给出的 PL/0 测试程序，并给出对应的目标指令结果。

PL/0 测试程序目标指令

```
1 jmp 0 98
2 int 0 5
3 lod 1 3
4 sto 0 3
5 lod 1 4
6 sto 0 4
7 lod 1 3
8 lod 1 4
9 opr 0 4
10 sto 1 5
11 opr 0 0
12 int 0 4
13 lod 1 3
14 sto 1 7
15 lit 0 0
16 sto 1 6
17 lod 1 4
18 sto 0 3
19 lod 0 3
20 lod 1 7
21 opr 0 11
22 jpc 0 28
23 lit 0 2
24 lod 0 3
25 opr 0 4
26 sto 0 3
27 jmp 0 19
28 lod 0 3
29 lod 1 4
30 opr 0 12
31 jpc 0 53
32 lit 0 2
33 lod 1 6
34 opr 0 4
35 sto 1 6
36 lod 0 3
37 lit 0 2
38 opr 0 5
39 sto 0 3
40 lod 0 3
41 lod 1 7
42 opr 0 11
43 jpc 0 52
44 lod 1 7
45 lod 0 3
46 opr 0 3
47 sto 1 7
48 lod 1 6
49 lit 0 1
50 opr 0 2
51 sto 1 6
52 jmp 0 28
53 opr 0 0
54 int 0 4
55 lod 1 3
56 sto 1 9
57 lod 1 4
58 sto 0 3
59 lod 1 9
60 lod 0 3
61 opr 0 9
62 jpc 0 80
63 lod 1 9
64 lod 0 3
65 opr 0 10
66 jpc 0 71
67 lod 0 3
68 lod 1 9
69 opr 0 3
70 sto 0 3
71 lod 0 3
72 lod 1 9
73 opr 0 10
74 jpc 0 79
75 lod 1 9
76 lod 0 3
77 opr 0 3
78 sto 1 9
79 jmp 0 59
80 lod 1 9
81 sto 1 5
82 opr 0 0
83 int 0 3
84 lod 1 8
85 lit 0 1
86 opr 0 12
87 jpc 0 97
88 lod 1 8
89 lod 1 9
90 opr 0 4
91 sto 1 9
92 lod 1 8
93 lit 0 1
94 opr 0 3
95 sto 1 8
96 cal 1 83
97 opr 0 0
98 int 0 10
99 opr 0 16
100 sto 0 3
101 opr 0 16
102 sto 0 4
103 cal 0 2
104 lod 0 5
105 opr 0 14
106 opr 0 15
107 opr 0 16
108 sto 0 3
109 opr 0 16
110 sto 0 4
111 cal 0 12
112 lod 0 6
113 opr 0 14
114 opr 0 15
115 lod 0 7
116 opr 0 14
117 opr 0 15
118 opr 0 16
119 sto 0 3
120 opr 0 16
121 sto 0 4
122 cal 0 54
123 lod 0 5
124 opr 0 14
125 opr 0 15
126 opr 0 16
127 sto 0 8
128 lit 0 1
129 sto 0 9
130 cal 0 83
131 lod 0 9
132 opr 0 14
133 opr 0 15
134 opr 0 0
```

3.4 完整代码

随着语法分析的完成，我们需要继续补充 "main.cpp" 中的代码，具体内容如下所示：

```
#include <iostream>
#include <vector>
#include <string>
```

```
#include <cstring>
#include "lexical_analyzer.h"
#include "syntax_analyzer.h"

char indir[] = "data/input/PL0_code";
char outdir1[] = "data/output/PL0_lexicon.txt"; // lexical analysis result
char outdir2[] = "data/output/PL0_table.txt";   // symbol table
char outdir3[] = "data/output/PL0_graph.txt";   // syntax tree data
char outdir4[] = "data/output/PL0_code.txt";    // PL0 code

int main(int argc, char * argv[]) {
    int len = strlen(indir);
    if(argc > 1) for(int i = 0; argv[1][i]; i++) indir[len++] = argv[1][i];
    indir[len++] = '.'; indir[len++] = 'i'; indir[len++] = 'n';

    // Output keywords
    // for(int i = 0; i < 29; i++)
    //     std::cout << "| " << i << " | " << keywords[i] << " |" << std::endl;

    // Lexical Analyzer - Part One
    if(!GETSYM(indir)) {std::cout << "Lexical Analyzer Error!" << std::endl; return 0;}
    else std::cout << "Lexical Analyzer Succeed!" << std::endl;
    SYM_OUTPUT(outdir1);
    std::cout << "The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: "
              << SYM.size() << '\n' << std::endl;

    // Syntax Analyzer - Part Two
    BLOCK();
    std::cout << "Syntax Analyzer Succeed!" << std::endl;
    TABLE_OUTPUT(outdir2);
    GRAPH_OUTPUT(outdir3);
    CODE_OUTPUT(outdir4);

    return 0;
}
```

3.4.1 syntax_analyzer.h

```
#ifndef SYNTAX_ANALYZER_H
#define SYNTAX_ANALYZER_H

#include <string>
```

```
#include <vector>
#include <map>

#define Judge(i, NAME) if(SYM[p1] == i) {p1++;} \
                       else {return Error(NAME);}

struct TableEntry {
    std::string NAME, KIND;
    int VAL, LEVEL, ADR, NXT;
    TableEntry() { ADR = NXT = -1; }
    TableEntry(std::string t1, int t2, int t3, int t4, int t5, int t6) {
        NAME = t1; KIND = t2; VAL = t3; LEVEL = t4; ADR = t5; NXT = t6;
    }
};

struct Instruction {
    std::string f;
    int l, a;
    Instruction() { f = "", l = a = 0; }
    Instruction(std::string f1, int l1, int a1) {
        f = f1; l = l1; a = a1;
    }
};

struct Table {
    int PreTable, PreEntry, VNUM;
    std::vector<TableEntry> table;
    std::map<std::string, std::pair<int, int> > mp1;
    std::map<std::string, int> mp2;
    Table() {
        PreTable = PreEntry = -1;
        VNUM = 0; // 记录变量个数
        table.clear();
        mp1.clear(); // 用于记录常量-(数值,0), 变量-(相对地址,1)
        mp2.clear(); // 用于记录过程-入口地址
    }
    Table(int t1, int t2) : PreTable(t1), PreEntry(t2) {}
};

extern std::vector<Table> TABLE;
extern std::vector<std::string> label;
extern std::vector<std::vector<int> > G;
```

```
47 extern std::vector<Instruction> CODE;
48
49 void BLOCK();
50 void TABLE_OUTPUT(char *outdir);
51 void GRAPH_OUTPUT(char *outdir);
52 void CODE_OUTPUT(char *outdir);
53
54 /*———————— Procedure Function ————————*/
55 void Procedure();
56 void SubProcedure(int, int);
57
58 /*———————— Specification Function ————————*/
59 void ConstDefinition(int);
60 void ConstSpecification(int);
61 void VariableSpecification(int);
62 void ProcedureSpecification(int);
63
64 /*———————— Statement Function ————————*/
65 void Statement(int);
66 void AssignStatement(int);
67 void ConditionStatement(int);
68 void LoopStatement(int);
69 void CallStatement(int);
70 void ReadStatement(int);
71 void WriteStatement(int);
72 void CompoundStatement(int);
73
74 /*———————— Other Functions ————————*/
75 void Item(int);
76 void Factor(int);
77 void Condition(int);
78 void Expression(int);
79
80 #endif
```

3.4.2 syntax_analyzer.cpp

输出函数 & 基础函数

```
1 #include <iostream>
2 #include <fstream>
3 #include <iomanip>
```

```
#include <algorithm>
#include "lexical_analyzer.h"
#include "syntax_analyzer.h"

const std::string CONSTANT = "CONSTANT", VARIABLE = "VARIABLE", PROCEDURE = "PROCEDURE";
const int DX = 3;
int TX = 0, LEV = 0, ADR = DX, p1, p2, p3, sz;

std::vector<Table> TABLE;
std::vector<int> empty_vec;
std::vector<std::string> label;
std::vector<std::vector<int> > G;
std::vector<Instruction> CODE;

void DFS_OUTPUT(int TableNum, int EntryNum, int flag, std::ofstream &output) {
    // Table Header Output
    if(flag == 1){
        output << std::left << std::setfill('-') << std::setw(100) << "-" << std::endl;
        output << "TX: " << std::left << std::setfill(' ') << std::setw(10) << TableNum;
        if(EntryNum >= TABLE[TableNum].table.size()){
            output << std::endl;
            return;
        }
        flag = 0;
    }
    else output << "    " << std::left << std::setfill(' ') << std::setw(10) << " ";

    // Table Information Output
    output << "NAME: " << std::left << std::setw(20)
           << TABLE[TableNum].table[EntryNum].NAME;

    output << "KIND: " << std::left << std::setw(25)
           << TABLE[TableNum].table[EntryNum].KIND;

    if(TABLE[TableNum].table[EntryNum].KIND == CONSTANT)
        output << "VAL: " << std::left << std::setw(15)
               << TABLE[TableNum].table[EntryNum].VAL << std::endl;
    else{
        if(TABLE[TableNum].table[EntryNum].KIND == PROCEDURE) {
            output << "LEVEL: " << std::left << std::setw(15)
                   << TABLE[TableNum].table[EntryNum].LEVEL
                   << "ADR: " << " " << std::endl;
```

```
46          }
47          else{
48              output << "LEVEL: " << std::left << std::setw(15)
49                     << TABLE[TableNum].table[EntryNum].LEVEL
50                     << "ADR: " << TABLE[TableNum].table[EntryNum].ADR << std::endl;
51          }
52      }
53
54      // Recursion & Backtrack Output
55      if(TABLE[TableNum].table[EntryNum].KIND == PROCEDURE)
56          DFS_OUTPUT(TABLE[TableNum].table[EntryNum].NXT, 0, 1, output), flag = 1;
57      if(EntryNum+1 < TABLE[TableNum].table.size())
58          DFS_OUTPUT(TableNum, EntryNum+1, flag, output);
59  }
60
61  void TABLE_OUTPUT(char *outdir) {
62      std::ofstream output(outdir);
63      DFS_OUTPUT(0, 0, 1, output);
64      output << std::left << std::setfill('-') << std::setw(100) << "-" << std::endl;
65      output.close();
66  }
67
68  void GRAPH_OUTPUT(char *outdir) {
69      std::ofstream output(outdir);
70      int n = label.size();
71      output << n << std::endl;
72      for(int i = 0; i < n; i++)
73          output << label[i] << " \n"[i==n-1];
74      for(int i = 0; i < n; i++){
75          int sz = G[i].size();
76          if(sz == 0) output << std::endl;
77          for(int j = 0; j < sz; j++)
78              output << G[i][j] << " \n"[j==sz-1];
79      }
80      output.close();
81  }
82
83  void CODE_OUTPUT(char *outdir) {
84      std::ofstream output(outdir);
85      int sz = CODE.size();
86      for(int i = 0; i < sz; i++)
87          output << i+1 << " " << CODE[i].f << " "
```

```
88                  << CODE[i].l << " " << CODE[i].a << std::endl;
89      output.close();
90  }
91
92  std::string ll2string(long long x){
93      std::string result = "";
94      if(x == 0) result += '0';
95      else{
96          while(x){
97              result += '0'+(x%10);
98              x /= 10;
99          }
100     }
101     reverse(result.begin(), result.end());
102     return result;
103 }
104
105 void Error(std::string error) {
106     std::cout << "error: expected " << error << " at the position "
107                 << ll2string(p1) << "." << std::endl;
108     std::cout << "Syntax Analyzer Error!" << std::endl;
109     exit(0);
110 }
111
112 int Graph_init(std::string name, int fa){
113     // Graph Construct
114     label.push_back(name);
115     G.push_back(empty_vec);
116     int id = label.size()-1;
117     G[fa].push_back(id);
118     return id;
119 }
120
121 void BLOCK() {
122     TABLE.push_back(Table());
123     p1 = p2 = p3 = 0; sz = SYM.size()-1;
124     label.clear(); G.clear();
125     Procedure();
126 }
```

程序 & 分程序

```
/*———————————— Procedure Function ————————————*/
void Procedure() {
    // Graph Construct
    label.push_back("程序");
    G.push_back(empty_vec);
    int id = label.size()-1;

    // the first code: jump to main procedure
    CODE.push_back(Instruction("jmp",0,0));
    SubProcedure(id, 1);

    if(SYM[p1] == 0 && p1 == sz){
        Graph_init(keywords[0], id);
        CODE.push_back(Instruction("opr", 0, 0));
    }
    else Error("'.' in the end of main procedure");
}

void SubProcedure(int fa, int op = 0) {
    int id = Graph_init("分程序", fa);

    if(SYM[p1] == 1) ConstSpecification(id);
    if(SYM[p1] == 5) VariableSpecification(id);
    if(SYM[p1] == 6) ProcedureSpecification(id);
    if(op){
        // main procedure add INT code and backfill JMP addr
        CODE.push_back(Instruction("int", 0, TABLE[0].VNUM+3));
        CODE[0].a = CODE.size();
    }
    Statement(id);
}
```

常量 & 变量 & 过程说明部分

```
/*———————————— Specification Function ————————————*/
void ConstDefinition(int fa) {
    int id = Graph_init("常量定义", fa);
    TableEntry entry;
    if(SYM[p1] == tagtype) {
        // std::cout << p1 << "," << ID[p2] << "," << SYM[p1] << std::endl;
        entry.NAME = ID[p2++];
        entry.KIND = CONSTANT;
        p1++;
```

```
10          Graph_init(entry.NAME, id);
11      }
12      else Error("'tagtype' in const definition");
13
14      Judge(4, "'=' in const definition");
15      Graph_init(keywords[4], id);
16
17      if(SYM[p1] == numtype) {
18          entry.VAL = NUM[p3++];
19          entry.LEVEL = LEV;
20          // redifinition const error
21          if(TABLE[TX].mp1.find(entry.NAME) != TABLE[TX].mp1.end())
22              Error("undeclared 'variable name' in const definition");
23          // assign value to the map of TABLE[TX], const's second dimensionality is 0
24          TABLE[TX].mp1[entry.NAME] = std::make_pair(entry.VAL, 0);
25          TABLE[TX].table.push_back(entry);
26          p1++;
27          Graph_init(ll2string(entry.VAL), id);
28      }
29      else Error("'numtype' in const definition");
30  }
31
32  void ConstSpecification(int fa) {
33      int id = Graph_init("常量说明部分", fa);
34      Judge(1, "'const' in const declaration");
35      Graph_init(keywords[1], id);
36
37      ConstDefinition(id);
38
39      while(SYM[p1] == 2){
40          p1++;
41          Graph_init(keywords[2], id);
42          ConstDefinition(id);
43      }
44      Judge(3, "';' in const declaration");
45      Graph_init(keywords[3], id);
46  }
47
48  void RecognizeFlag(std::string flag, int fa, int op = 0) {
49      TableEntry entry;
50      entry.NAME = ID[p2++];
51      entry.KIND = flag;
```

```
52      entry.LEVEL = LEV;
53      entry.ADR = ADR++;
54      if(op){
55          // redifinition procedure error
56          if(TABLE[TX].mp2.find(entry.NAME) != TABLE[TX].mp2.end())
57              Error("different 'procedure name' in procedure declaration");
58          // assign value to the map of TABLE[TX]
59          TABLE[TX].mp2[entry.NAME] = CODE.size()+1;
60      }
61      else{
62          // redifinition variable error
63          if(TABLE[TX].mp1.find(entry.NAME) != TABLE[TX].mp1.end())
64              return Error("different 'variable name' in variable / const declaration");
65          // assign value to the map of TABLE[TX], variable's second dimensionality is 1
66          TABLE[TX].mp1[entry.NAME] = std::make_pair(entry.ADR, 1);
67          // the number of variable in TABLE[TX] ++
68          TABLE[TX].VNUM++;
69      }
70      TABLE[TX].table.push_back(entry);
71      p1++;
72      Graph_init(entry.NAME, fa);
73  }
74
75  void VariableSpecification(int fa) {
76      int id = Graph_init("变量说明部分", fa);
77      Judge(5, "'var' in variable declaration");
78      Graph_init(keywords[5], id);
79      if(SYM[p1] != tagtype) Error("'tagtype' in variable declaration");
80      RecognizeFlag(VARIABLE, id);
81
82      while(SYM[p1] == 2) {
83          p1++;
84          Graph_init(keywords[2], id);
85          if(SYM[p1] != tagtype) Error("'tagtype' in variable declaration");
86          RecognizeFlag(VARIABLE, id);
87      }
88      Judge(3, "';' in variable declaration");
89      Graph_init(keywords[3], id);
90  }
91
92  void ProcedureSpecification(int fa) {
93      int id = Graph_init("过程说明部分", fa);
```

```
94      Judge(6, "'procedure' in procedure declaration");
95      Graph_init(keywords[6], id);
96      if(SYM[p1] == tagtype){
97          RecognizeFlag(PROCEDURE, id, 1);
98          TABLE.push_back(Table());
99          TABLE[TABLE.size()-1].PreTable = TX;
100         TABLE[TABLE.size()-1].PreEntry = TABLE[TX].table.size()-1;
101         TABLE[TX].table[TABLE[TX].table.size()-1].NXT = TABLE.size()-1;
102         TX = TABLE.size()-1, LEV++, ADR = DX;
103     }
104     else return Error("'tagtype' in procedure declaration");
105
106     Judge(3, "';' in procedure declaration");
107     Graph_init(keywords[3], id);
108
109     // add the first INT code of procedure
110     int index = CODE.size();
111     CODE.push_back(Instruction("int",0,0));
112     SubProcedure(id);
113     // backfill the size of INT code
114     CODE[index].a = TABLE[TX].VNUM + 3;
115
116     Judge(3, "';' in procedure declaration");
117     Graph_init(keywords[3], id);
118     LEV--;
119     int PreEntry = TABLE[TX].PreEntry;
120     TX = TABLE[TX].PreTable;
121     ADR = ADR - 1 + TABLE[TX].table[PreEntry].ADR;
122     TABLE[TX].table[PreEntry].ADR = ADR++;
123     CODE.push_back(Instruction("opr", 0, 0));
124
125     while(SYM[p1] == 6) ProcedureSpecification(id);
126 }
```

各类语句

```
1  /*———————————— Statement Function ————————————*/
2  std::pair<int,int> SearchPosition(std::string NAME, int op = 0) {
3      /*
4          constant: (-1, val), variable: (l, a), procedure: (l, addr)
5          variable name should differ from const name
6          procedure name could be same with variable or const name
```

```
7       */
8       int now = TX, cnt = 0;
9       while(now != -1) {
10          if(!op && TABLE[now].mp1.find(NAME) != TABLE[now].mp1.end()){
11              if(TABLE[now].mp1[NAME].second == 0)
12                  return std::make_pair(-1, TABLE[now].mp1[NAME].first);
13              else
14                  return std::make_pair(cnt, TABLE[now].mp1[NAME].first);
15          }
16          // search procedure
17          if(op && TABLE[now].mp2.find(NAME) != TABLE[now].mp2.end())
18              return std::make_pair(cnt, TABLE[now].mp2[NAME]);
19          now = TABLE[now].PreTable, cnt++;
20      }
21      return std::make_pair(-3, -1);
22  }
23
24  void Statement(int fa) {
25      if(SYM[p1] == tagtype) AssignStatement(Graph_init("语句", fa));
26      else if(SYM[p1] == 20) ConditionStatement(Graph_init("语句", fa));
27      else if(SYM[p1] == 23) LoopStatement(Graph_init("语句", fa));
28      else if(SYM[p1] == 22) CallStatement(Graph_init("语句", fa));
29      else if(SYM[p1] == 25) ReadStatement(Graph_init("语句", fa));
30      else if(SYM[p1] == 26) WriteStatement(Graph_init("语句", fa));
31      else if(SYM[p1] == 8) CompoundStatement(Graph_init("语句", fa));
32  }
33
34  void AssignStatement(int fa) {
35      int id = Graph_init("赋值语句", fa);
36      if(SYM[p1] == tagtype) Graph_init(ID[p2++], id), p1++;
37      else return Error("'tagtype' in assign statement");
38
39      // acquire the position of variable
40      std::pair<int, int> pos = SearchPosition(ID[p2-1]);
41      // pos must be variable
42      if(pos.first < 0) return Error("declared 'variable name' in assign statement");
43
44      Judge(7, "':=' in assign statement");
45      Graph_init(keywords[7], id);
46      Expression(id);
47      // assign code
48      CODE.push_back(Instruction("sto", pos.first, pos.second));
```

```
49 }
50
51 void ConditionStatement(int fa) {
52     int id = Graph_init("条件语句", fa);
53     Judge(20, "'if' in condition statement");
54     Graph_init(keywords[20], id);
55     Condition(id);
56     CODE.push_back(Instruction("jpc", 0, 0));
57     int index = CODE.size();
58
59     Judge(21, "'then' in condition statement");
60     Graph_init(keywords[21], id);
61     Statement(id);
62     CODE[index-1].a = CODE.size()+1;
63 }
64
65 void LoopStatement(int fa) {
66     int id = Graph_init("当型循环语句", fa);
67     Judge(23, "'while' in loop statement");
68     Graph_init(keywords[23], id);
69     int index1 = CODE.size();
70     Condition(id);
71     CODE.push_back(Instruction("jpc", 0, 0));
72     int index2 = CODE.size();
73
74     Judge(24, "'do' in loop statement");
75     Graph_init(keywords[24], id);
76     Statement(id);
77     CODE.push_back(Instruction("jmp", 0, index1+1));
78     CODE[index2-1].a = CODE.size()+1;
79 }
80
81 void CallStatement(int fa) {
82     int id = Graph_init("过程调用语句", fa);
83     Judge(22, "'call' in call procedure statement");
84     Graph_init(keywords[22], id);
85     if(SYM[p1] == tagtype) {
86         // acquire the position of precedure
87         std::pair<int, int> pos = SearchPosition(ID[p2], 1);
88         // pos must be precedure
89         if(pos.first >= 0) CODE.push_back(Instruction("cal", pos.first , pos.second));
90         else Error("declared 'procedure name' in call procedure statement");
```

```
91
92          Graph_init(ID[p2++], id), p1++;
93      }
94      else return Error("declared 'procedure name' in call procedure statement");
95  }
96
97  void ReadStatement(int fa) {
98      int id = Graph_init("读语句", fa);
99      Judge(25, "'read' in read statement");
100     Graph_init(keywords[25], id);
101     Judge(27, "'(' in read statement");
102     Graph_init(keywords[27], id);
103     if(SYM[p1] == tagtype) {
104         CODE.push_back(Instruction("opr", 0, 16));
105         // acquire the position of variable
106         std::pair<int, int> pos = SearchPosition(ID[p2]);
107         // pos must be variable
108         if(pos.first >= 0) CODE.push_back(Instruction("sto", pos.first, pos.second));
109         else Error("declared 'variable name' in read statement");
110
111         Graph_init(ID[p2++], id), p1++;
112     }
113     else Error("declared 'variable name' in read statement");
114
115     while(SYM[p1] == 2){
116         p1++;
117         Graph_init(keywords[2], id);
118         if(SYM[p1] == tagtype) {
119             CODE.push_back(Instruction("opr", 0, 16));
120             // acquire the position of variable
121             std::pair<int, int> pos = SearchPosition(ID[p2]);
122             // pos must be variable
123             if(pos.first >= 0) CODE.push_back(Instruction("sto", pos.first, pos.second));
124             else Error("declared 'variable name' in read statement");
125
126             Graph_init(ID[p2++], id), p1++;
127         }
128         else Error("declared 'variable name' in read statement");
129     }
130
131     Judge(28, "')' in read statement");
132     Graph_init(keywords[28], id);
```

```
}

void WriteStatement(int fa) {
    int id = Graph_init("写语句", fa);
    Judge(26, "'write' in write statement");
    Graph_init(keywords[26], id);
    Judge(27, "'(' in write statement");
    Graph_init(keywords[27], id);
    Expression(id);
    CODE.push_back(Instruction("opr", 0, 14));
    CODE.push_back(Instruction("opr", 0, 15));

    while(SYM[p1] == 2){
        p1++;
        Graph_init(keywords[2], id);
        Expression(id);
        CODE.push_back(Instruction("opr", 0, 14));
        CODE.push_back(Instruction("opr", 0, 15));
    }

    Judge(28, "')' in write statement");
    Graph_init(keywords[28], id);
}

void CompoundStatement(int fa) {
    int id = Graph_init("复合语句", fa);
    Judge(8, "'begin' in compound statement");
    Graph_init(keywords[8], id);
    Statement(id);
    while(SYM[p1] == 3){
        p1++;
        Graph_init(keywords[3], id);
        Statement(id);
    }
    Judge(9, "'end' in compound statement");
    Graph_init(keywords[9], id);
}
```

项 & 因子 & 条件 & 表达式

```
/*———————————— Other Functions ————————————*/
void Item(int fa) {
    int id = Graph_init("项", fa), op = 0;
```

```
4       Factor(id);
5       while(SYM[p1] == 13 || SYM[p1] == 14){
6           if(SYM[p1] == 13) op = 1;
7           else op = -1;
8           Graph_init(keywords[SYM[p1++]], id);
9           Factor(id);
10          // solve *、/
11          if(op == 1) CODE.push_back(Instruction("opr", 0, 4));
12          else CODE.push_back(Instruction("opr", 0, 5));
13      }
14  }
15
16  void Factor(int fa) {
17      int id = Graph_init("因子", fa);
18      if(SYM[p1] == tagtype) {
19          // acquire the position of variable / constant
20          std::pair<int, int> pos = SearchPosition(ID[p2]);
21          // pos must be variable
22          if(pos.first >= 0) CODE.push_back(Instruction("lod", pos.first , pos.second));
23          else if(pos.first == -1) CODE.push_back(Instruction("lit", 0, pos.second));
24          else Error("declared 'variable name' as a factor");
25
26          Graph_init(ID[p2++], id), p1++;
27          return;
28      }
29      if(SYM[p1] == numtype) {
30          CODE.push_back(Instruction("lit", 0, NUM[p3]));
31          Graph_init(ll2string(NUM[p3++]), id), p1++;
32          return;
33      }
34      if(SYM[p1] == 27) {
35          p1++;
36          Graph_init(keywords[27], id);
37          Expression(id);
38          Judge(28, "')' in the end of expression statement as a factor");
39          Graph_init(keywords[28], id);
40          return;
41      }
42      return Error("'tagtype' / 'numtype' / 'expression statement' as a factor");
43  }
44
45  void Condition(int fa) {
```

```
46      int id = Graph_init("条件", fa);
47      if(SYM[p1] == 10){
48          p1++;
49          Graph_init(keywords[10], id);
50          Expression(id);
51          CODE.push_back(Instruction("opr", 0, 6));
52      }
53      else{
54          Expression(id);
55          if(SYM[p1] == 4 || (SYM[p1] >= 15 && SYM[p1] <= 19)) {
56              Graph_init(keywords[SYM[p1++]], id);
57              int op = SYM[p1-1];
58              Expression(id);
59              if(op == 4) CODE.push_back(Instruction("opr", 0, 8));
60              else CODE.push_back(Instruction("opr", 0, op-6));
61          }
62          else Error("'operator specifier' between expression statements");
63      }
64  }
65
66  void Expression(int fa) {
67      int id = Graph_init("表达式", fa), op = 0;
68      if(SYM[p1] == 11 || SYM[p1] == 12) {
69          if(SYM[p1] == 12) op = -1;
70          Graph_init(keywords[SYM[p1++]], id);
71      }
72      Item(id);
73      // solve uminus
74      if(op == -1) {
75          CODE.push_back(Instruction("lit", 0, -1));
76          CODE.push_back(Instruction("opr", 0, 4));
77      }
78      while(SYM[p1] == 11 || SYM[p1] == 12) {
79          if(SYM[p1] == 11) op = 1;
80          else op = -1;
81          Graph_init(keywords[SYM[p1++]], id);
82          Item(id);
83          // solve +、-
84          if(op == 1) CODE.push_back(Instruction("opr", 0, 2));
85          else CODE.push_back(Instruction("opr", 0, 3));
86      }
87  }
```

3.4.3 syntax_tree.py

```
from graphviz import Digraph

def dfs(x, fa, lable, G, graph):
    graph.node(str(x), lable[x])
    if fa != -1:
        graph.edge(str(fa), str(x))
    for y in G[x]:
        if y == "":
            break
        dfs(int(y), x, lable, G, graph)


def data_acquire(path):
    # read file
    fo = open(path, "r+")
    data = fo.read().split("\n")

    # acquire data
    n = int(data[0])
    lable = data[1].split(" ")
    for i in range(0, len(lable)):
        if lable[i] == "<>":
            lable[i] = "\<\>"

    G = []
    for row in range(2, len(data)):
        G.append(data[row].split(" "))
    return n, lable, G


if __name__ == "__main__":
    # path list
    path = "data/output/PL0_graph.txt"
    out_path1 = "data/output/PL0_graph_structure.txt"
    out_path2 = "data/output/PL0_graph"

    # init data acquire
    n, lable, G = data_acquire(path)
    # graph = gz.Graph()
    graph = Digraph('G', filename=out_path2)
    graph.attr(rankdir="LR", style="filled", bgcolor="white")
```

```
    graph.node_attr.update(style='filled', color='white')
    graph.edge_attr.update(style='filled', color='grey')

    # dfs construct graph
    dfs(int(0), int(-1), lable, G, graph)

    # output graph structure
    fo = open(out_path1, "w")
    fo.write(graph.source)
    graph.render(out_path2, view=True)
```

4 目标指令解释器

4.1 虚拟机

4.1.1 任务说明

执行完词法分析、语法分析后，内存中只剩下用于存放目标程序的 CODE 数组和运行时的数据区 S。其中 S 是由解释程序定义的一维整型数组，代表栈式计算机的存储空间，遵循后进先出的规则。另外，只有过程被调用时，才会在 S 中分配数据空间，并且退出过程时，所分配的数据空间被释放。

除了 CODE 数组与数据区 S 外，我们还需要定义三个寄存器，分别是：

1. PC: 指令地址寄存器，作为 CODE 数组的下标指向即将要执行的目标指令。
2. SP: 栈顶地址寄存器，作为 S 数据区的下标指向栈顶元素。
3. BA: 基地址寄存器，存储当前被调用的过程在数据区中的数据段起始地址。

另外，对于被调用的过程来说，其在数据区 S 中的数据段都分为两个部分：

1. 静态部分：

   用于变量存放与三个联系单元的存储，三个联系单元分别为：

   - 静态链：SL，指向定义该过程的直接外过程运行时数据段的基地址
   - 动态链：DL，指向调用该过程前正在运行过程的数据段的基地址
   - 返回地址：RA，记录调用该过程时目标程序的断点，即当时的指令地址寄存器 PC 的值

2. 动态部分：

   作为临时工作单元使用，需要时临时分配，用完立即释放。

4.1.2 指令解释

变量定义

根据上述的任务描述，我们只需额外定义数据区 S 以及三个全局变量 PC、SP、BA 即可。而由于数据区 S 是动态增长的，因此本编译系统仍旧采用 vector 进行实现。并且为了避免每次加入新数据都要扩大 vector 容量，造成时间上的浪费，本编译系统在每次申请 vector 容量时，都会以 100 为单位进行容量申请，降低时间开销。

```
int CAPACITY = 100, BASE = 100, PC, SP, BA;
std::vector<int> S;

void Push(int v) {
    if(SP+2 > S.size()) CAPACITY += BASE, S.resize(CAPACITY);
    S[++SP] = v;
}
```

执行目标指令

接下来我们主要是根据在 3.3.2 中所给出的指令微操作来编写代码，执行目标指令。此处以 *LOD l a*、*STO l a*、*CAL l a* 为例进行说明。

1. *LOD l a*

```
if(op == "lod") {
    int tb = BA;
    while(l--) tb = S[tb];
    Push(S[tb+a]);
}
```

BA 为基地址，由于基地址处存储的是当前数据段的静态链，因此 *while* 部分的代码可以直接定位到层差为 *l* 的数据段，再加上 *a* 的偏移量即可取出对应数据。

2. *STO l a*

```
if(op == "sto") {
    int tb = BA;
    while(l--) tb = S[tb];
    if(tb+a >= S.size()) RuntimeError("access the undefined stack space");
    S[tb+a] = S[SP];
    SP--;
}
```

在该指令的执行过程中，我们加入了 Runtime Error 的判断。当然正常情况下，跳静态链 SP 的值都是越跳越小的，但仍然不排除存在 tb+a 越界的情况，因此我们需要进行 RE 判断。

3. *CAL l a*

```
if(op == "cal") {
    int tb = BA;
    while(l--) tb = S[tb];
    Push(tb);
    Push(BA);
    Push(PC);
    SP -= 3;
    PC = a-1;
    BA = SP+1;
}
```

对于该指令的执行，我们需要先定位具体的过程入口，再进行 SL、DL、RA 赋值，本质上还是对于指令微操作的一种模拟复现。

4.1.3 运行测试

运行测试部分，我们一共给出五段测试代码，以及其对应的命令行测试结果。

PL/0 测试程序 1

```
var x, y, z, q, r, n, f;

procedure multiply;
var a, b;
begin
  a := x;
  b := y;
  z := x*y
end;

procedure divide;
var w;
begin
  r := x;
  q := 0;
  w := y;
  while w <= r do w := 2 * w;
  while w > y do
  begin
    q := 2 * q;
    w := w / 2;
    if w <= r then
      begin
        r := r - w;
        q := q + 1
      end
  end
end;

procedure gcd;
var  g;
begin
  f := x;
  g := y;
  while f <> g do
  begin
    if f < g then g := g - f;
    if g < f then f := f - g
  end;
  z := f
end;

procedure fact;
begin
  if n > 1 then
```

```
    begin
      f := n * f;
      n := n - 1;
      call fact
    end
end;

begin
  read(x);read(y); call multiply;write(z);
  read(x);read(y); call divide; write(q); write(r);
  read(x);read(y); call gcd; write(z);
  read(n); f := 1; call fact; write(f)
end.
```

```
gene_liu@Gene-Liu Project % ./compiler 1
-------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 262

-------------------------------- PART TWO ---------------------------------
Syntax Analyzer Succeed!

------------------------------- PART THREE --------------------------------
Choose: '0' for Run, '1' for Debug
0
Please input an integer: 10
Please input an integer: 20
200
Please input an integer: 30
Please input an integer: 40
0
30
Please input an integer: 50
Please input an integer: 60
10
Please input an integer: 10
3628800
Program Finished Successfully!
```

图 9: PL/0 测试程序 1 运行结果

```
gene_liu@Gene-Liu Project % ./compiler 2
-------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 75

-------------------------------- PART TWO ---------------------------------
Syntax Analyzer Succeed!

------------------------------- PART THREE --------------------------------
Choose: '0' for Run, '1' for Debug
0
Please input an integer: 10
Please input an integer: 20
10
20
0
Program Finished Successfully!
```

图 10: PL/0 测试程序 2 运行结果

PL/0 测试程序 2

```
const a=10;
var d,e,f;
procedure p;
```

```
var g;
begin
  d:=a*2;
  e:=a/3;
  if d<=e then f:=d+e
end;
```

```
begin
  read(e,f);
  write(e,f,d);
  call p;
  while odd d do e:=-e+1
end.
```

PL/0 测试程序 3

```
const a=10;
var b,c;
procedure p;
begin
  c:=b+a
end;
begin
  read(b);
  while b<>0 do
    begin
      call p;
      write(2*c);
      read(b);
    end
end.
```

```
gene_liu@Gene-Liu Project % ./compiler 3
-------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 51

-------------------------------- PART TWO ---------------------------------
Syntax Analyzer Succeed!

------------------------------- PART THREE --------------------------------
Choose: '0' for Run, '1' for Debug
0
Please input an integer: 10
40
Please input an integer: 20
60
Please input an integer: 30
80
Please input an integer: 10
40
Please input an integer: 50
120
Please input an integer: 0
Program Finished Successfully!
```

图 11: PL/0 测试程序 3 运行结果

PL/0 测试程序 4

```
var x, squ;

procedure square;
begin
   squ:= x * x
end;

begin
   x := 1;
   while x <= 10 do
   begin
      call square;
      write(squ);
      x := x + 1
   end
end.
```

```
gene_liu@Gene-Liu Project % ./compiler 4
--------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 43

--------------------------------- PART TWO ---------------------------------
Syntax Analyzer Succeed!

-------------------------------- PART THREE --------------------------------
Choose: '0' for Run, '1' for Debug
0
1
4
9
16
25
36
49
64
81
100
Program Finished Successfully!
```

图 12: PL/0 测试程序 4 运行结果

```
gene_liu@Gene-Liu Project % ./compiler 5
--------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 96

--------------------------------- PART TWO ---------------------------------
Syntax Analyzer Succeed!

-------------------------------- PART THREE --------------------------------
Choose: '0' for Run, '1' for Debug
0
2
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
Program Finished Successfully!
```

图 13: PL/0 测试程序 5 运行结果

PL/0 测试程序 5

```
const max = 100;
var arg, ret;

procedure isprime;
var i;
```

```
6  begin
7      ret := 1;
8      i := 2;
9      while i < arg do
10     begin
11         if arg / i * i = arg then
12         begin
13             ret := 0;
14             i := arg
15         end;
16         i := i + 1
17     end
18 end;
19
```

```
20 procedure primes;
21 begin
22     arg := 2;
23     while arg < max do
24     begin
25         call isprime;
26         if ret = 1 then write(arg);
27         arg := arg + 1
28     end
29 end;
30
31 call primes
32 .
```

DEBUG 模式

另外，本编译系统还提供了 DEBUG 模式，即单步执行指令，每执行一次便输出 BA、SP、PC、CODE[PC] 以及 STACK[0 10] 中的数据。我们选取 PL/0 测试程序 4 在 DEBUG 模式下的部分运行情况来进一步说明。

```
gene_liu@Gene-Liu Project % ./compiler 4
--------------------------------- PART ONE ---------------------------------
Lexical Analyzer Succeed!
The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: 43

--------------------------------- PART TWO ---------------------------------
Syntax Analyzer Succeed!

-------------------------------- PART THREE --------------------------------
Choose: '0' for Run, '1' for Debug
1
PC: 1, CODE[PC]: jmp 0 8
One Instruction Executed, BA: 0, SP: -1, STACK[0~10]: [0 0 -1 0 0 0 0 0 0 0 0]
Please input '1' to continue
1

PC: 8, CODE[PC]: int 0 5
One Instruction Executed, BA: 0, SP: 4, STACK[0~10]: [0 0 -1 0 0 0 0 0 0 0 0]
Please input '1' to continue
1

PC: 9, CODE[PC]: lit 0 1
One Instruction Executed, BA: 0, SP: 5, STACK[0~10]: [0 0 -1 0 0 1 0 0 0 0 0]
Please input '1' to continue
```

图 14: PL/0 测试程序 4 - DEBUG 运行结果

其中每一条指令结束都会暂停，需要输入 1 才会执行下一条指令，用于程序出错时的单步调试使用。

4.2 完整代码

随着目标指令解释器代码的完成，我们实现了整个编译系统，因此我们给出最终完整版的 "main.cpp" 的代码，具体内容如下所示：

```
#include <iostream>
#include <vector>
#include <string>
#include <cstring>
#include "lexical_analyzer.h"
#include "syntax_analyzer.h"
#include "code_interpreter.h"

char indir[] = "data/input/PL0_code";
char outdir1[] = "data/output/PL0_lexicon.txt"; // lexical analysis result
char outdir2[] = "data/output/PL0_table.txt";   // symbol table
char outdir3[] = "data/output/PL0_graph.txt";   // syntax tree data
char outdir4[] = "data/output/PL0_code.txt";    // PL0 code

int main(int argc, char * argv[]) {
    int len = strlen(indir);
    if(argc > 1) for(int i = 0; argv[1][i]; i++) indir[len++] = argv[1][i];
    indir[len++] = '.'; indir[len++] = 'i'; indir[len++] = 'n';

    // Output keywords
    // for(int i = 0; i < 29; i++)
    //     std::cout << "| " << i << " | " << keywords[i] << " |" << std::endl;

    // Lexical Analyzer - Part One
    if(!GETSYM(indir)) {std::cout << "Lexical Analyzer Error!" << std::endl; return 0;}
    else std::cout << "Lexical Analyzer Succeed!" << std::endl;
    SYM_OUTPUT(outdir1);
    std::cout << "The Total Number of Keywords + Identifiers + Numbers in PL/0 Code: "
              << SYM.size() << '\n' << std::endl;

    // Syntax Analyzer - Part Two
    BLOCK();
    std::cout << "Syntax Analyzer Succeed!" << std::endl;
    TABLE_OUTPUT(outdir2);
    GRAPH_OUTPUT(outdir3);
    CODE_OUTPUT(outdir4);

    // Code Interpreter - Part Three
```

```
    Run();
    std::cout << "Program Finished Successfully!" << std::endl;
    return 0;
}
```

4.2.1 code_interpreter.h

```
#ifndef CODE_INTERPRETER_H
#define CODE_INTERPRETER_H

#include <string>
#include <vector>
#include <iostream>

extern int CAPACITY, BASE, PC, SP, BA;
extern std::vector<int> S;

void Run();

#endif
```

4.2.2 code_interpreter.cpp

```
#include "syntax_analyzer.h"
#include "code_interpreter.h"

int CAPACITY = 100, BASE = 100, PC, SP, BA;
std::vector<int> S;

void RuntimeError(std::string error) {
    std::cout << "error: " << error << " at the CODE[" << PC << "], "
              << CODE[PC-1].f << " " << CODE[PC-1].l << " "
              << CODE[PC-1].a << "." << std::endl;
    std::cout << "Code Interpreter Error!" << std::endl;
    exit(0);
}

void Push(int v) {
    if(SP+2 > S.size()) CAPACITY += BASE, S.resize(CAPACITY);
    S[++SP] = v;
}
```

```
19
20 bool OneInstruction() {
21     std::string op = CODE[PC-1].f;
22     int l = CODE[PC-1].l, a = CODE[PC-1].a;
23     if(op == "lit") {
24         Push(a);
25     }
26     else if(op == "lod") {
27         int tb = BA;
28         while(l--) tb = S[tb];
29         Push(S[tb+a]);
30     }
31     else if(op == "sto") {
32         int tb = BA;
33         while(l--) tb = S[tb];
34         if(tb+a >= S.size()) RuntimeError("access the undefined stack space");
35         S[tb+a] = S[SP];
36         SP--;
37     }
38     else if(op == "cal") {
39         int tb = BA;
40         while(l--) tb = S[tb];
41         Push(tb);
42         Push(BA);
43         Push(PC);
44         SP -= 3;
45         PC = a-1;
46         BA = SP+1;
47     }
48     else if(op == "int") {
49         SP += a;
50     }
51     else if(op == "jmp") {
52         PC = a-1;
53     }
54     else if(op == "jpc") {
55         if(!S[SP]) PC = a-1;
56         SP--;
57     }
58     else if(op == "opr") {
59         switch (a)
60         {
```

```
61              case 0:
62                  SP = BA-1;
63                  PC = S[BA+2];
64                  BA = S[BA+1];
65                  break;
66              case 1:
67                  S[SP] = -S[SP];
68                  break;
69              case 2:
70                  SP--;
71                  S[SP] = S[SP] + S[SP+1];
72                  break;
73              case 3:
74                  SP--;
75                  S[SP] = S[SP] - S[SP+1];
76                  break;
77              case 4:
78                  SP--;
79                  S[SP] = S[SP] * S[SP+1];
80                  break;
81              case 5:
82                  SP--;
83                  if(S[SP+1] == 0) RuntimeError("divisor is 0");
84                  S[SP] = S[SP] / S[SP+1];
85                  break;
86              case 6:
87                  S[SP] = S[SP] % 2;
88                  break;
89              case 7:
90                  SP--;
91                  S[SP] = S[SP] % S[SP+1];
92                  break;
93              case 8:
94                  SP--;
95                  S[SP] = (S[SP] == S[SP+1]);
96                  break;
97              case 9:
98                  SP--;
99                  S[SP] = (S[SP] != S[SP+1]);
100                 break;
101             case 10:
102                 SP--;
```

```
103                 S[SP] = (S[SP] < S[SP+1]);
104                 break;
105             case 11:
106                 SP--;
107                 S[SP] = (S[SP] <= S[SP+1]);
108                 break;
109             case 12:
110                 SP--;
111                 S[SP] = (S[SP] > S[SP+1]);
112                 break;
113             case 13:
114                 SP--;
115                 S[SP] = (S[SP] >= S[SP+1]);
116                 break;
117             case 14:
118                 std::cout << S[SP--];
119                 break;
120             case 15:
121                 std::cout << "\n";
122                 break;
123             case 16:
124                 Push(0);
125                 std::cout << "Please input an integer: ";
126                 std::cin >> S[SP];
127                 break;
128             default:
129                 break;
130         }
131     }
132     return true;
133 }
134
135 void Run() {
136     S.resize(CAPACITY);
137     PC = 1, SP = -1, BA = 0;
138     Push(0);
139     Push(0);
140     Push(-1);
141     SP -= 3;
142
143     std::cout << "Choose: '0' for Run, '1' for Debug" << std::endl;
144     int op;
```

```
145     while(std::cin >> op) {
146         if(op == 0 || op == 1) break;
147         else std::cout << "Choose: '0' for Run, '1' for Debug" << std::endl;
148     }
149     std::cin.clear();
150     std::cin.sync();
151     if(!op) {
152         while(PC) {
153             OneInstruction();
154             PC += 1;
155         }
156     }
157     else {
158         while(PC) {
159             std::cout << "PC: " << PC << ", CODE[PC]: " << CODE[PC-1].f
160                       << " " << CODE[PC-1].l << " " << CODE[PC-1].a << std::endl;
161             OneInstruction();
162             std::cout << "One Instruction Executed, BA: " << BA << ", SP: "
163                       << SP << ", STACK[0~10]: [";
164             for(int i = 0; i <= 10; i++)
165                 std::cout << S[i] << " ]"[i==10];
166             std::cout << std::endl << "Please input '1' to continue" << std::endl;
167             while(getchar() != '1');
168             std::cout << std::endl;
169             PC += 1;
170         }
171     }
172 }
```

5 致谢与展望

感谢本次 PL/0 编译系统实践中王丽荣老师对我们的教导与指点，这些编译知识在原理理解与代码编写过程中起到了很大的帮助，为最终编译系统的实现打下了坚实的基础；感谢助教袁娜学姐解答我们在系统实践中的疑问并为我们提供 PL/0 测试程序，这对我们后续验证编译系统正确性起到了很大的帮助；感谢我的同学们在实践过程中与我一同讨论编译系统的实现方案，帮助我考虑到了很多未曾想到的漏洞并为我提供了很多更高效实现方案的思路，这对整个编译系统的鲁棒性、高效性起到了重要作用。

在编译系统的搭建过程中，我遇到了大量未在课堂中出现的问题，并最终通过资料搜寻进行解决。当然，目前的编译系统仍然是粗糙的，缺乏对 PL/0 语法的进一步扩展以及目标指令的优化，望后来者能够在当前基础上进一步进行扩展强化，以期获得一个适用范围更广、编译运行效率更高的 PL/0 编译系统。

最后，这次从零开始的 PL/0 编译系统搭建，极大地加深了我对于编译程序以及编译原理的理解，希望在未来的研究过程中，我能学会以一种底层的、编译原理的角度来审视不同的编译语言，发现创新点，改进可优化点，以期在这条漫漫长路上乐此不疲地持续前行！

6 参考文献

[1] 陈火旺, 刘春林, 谭庆平, 等. 程序设计语言编译原理（第 3 版）[M]. 国防工业出版社: 北京,2000:1.