PL/0语言编译器

学号：2019********
姓名：王泽宇
Email:wangzeyujiuyi@qq.com

1. PL/0语言介绍

1.1 语法介绍

PL/0实验是PASCAL语言的子集，语法语义清晰简洁。它提供了如下几种语义功能：

1. 过程。PL/0支持过程调用、跨层次过程调用和过程自调用。
2. 常量。PL/0支持不可改变的无符号整型常量。
3. 变量。PL/0支持可改变的无符号整型变量。
4. if语句。PL/0支持使用if语句进行条件跳转。
5. while语句。PL/0支持使用wile语句编写当型循环程序。
6. read()、write()语句。PL/0支持使用read、write语句进行输入、输出。
7. 四则运算。PL/10支持加减乘除四则混合运算。
   更详细的语法规则我们使用上下文无关文法表示。首先列出其BNF表示，而后列出一般的上下文无关文法表示。

1.2 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>}
<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

1.3 CFG表示

在CFG表示中，我们是用^代替空串符号，并添加了若干语句用于翻译模式设计。

```
E -> EM BLOCK_FIRST
EM -> ^
BLOCK_FIRST -> BLOCKM_FIRST STATEMENT
BLOCKM_FIRST -> CONST_PART' VARIABLE_PART' PROCEDURE_PART'
CONST_PART' -> CONST_PART
CONST_PART' -> ^
VARIABLE_PART' -> VARIABLE_PART
VARIABLE_PART' -> ^
PROCEDURE_PART' -> PROCEDURE_PART
PROCEDURE_PART' -> ^
CONST_PART -> const CONST_DEFINE CONST_MORE ;
CONST_MORE ->  , CONST_DEFINE CONST_MORE
CONST_MORE ->  ^
CONST_DEFINE -> identifier = number
VARIABLE_PART -> var identifier VARIABLE_MORE ;
VARIABLE_MORE -> , identifier VARIABLE_MORE
VARIABLE_MORE -> ^
PROCEDURE_PART -> PROCEDURE_MORE PROCEDURE_HEADER BLOCK ;
PROCEDURE_MORE -> PROCEDURE_PART
PROCEDURE_MORE -> ^
PROCEDURE_HEADER -> procedure identifier ;
BLOCK -> BLOCKM STATEMENT
BLOCKM -> CONST_PART' VARIABLE_PART' PROCEDURE_PART'
STATEMENT -> ASSIGN_STATEMENT
STATEMENT -> CONDITION_STATEMENT
STATEMENT -> WHILE_STATEMENT
STATEMENT -> CALL_STATEMENT
STATEMENT -> READ_STATEMENT
STATEMENT -> WRITE_STATEMENT
STATEMENT -> COMPOUND_STATEMENT
STATEMENT -> ^
ASSIGN_STATEMENT -> identifier := EXPRESSION
CONDITION -> EXPRESSION RELATION_OPT EXPRESSION
CONDITION -> odd EXPRESSION
EXPRESSION -> EXPRESSION_FIRST_CALCULATE EXPRESSION_MORE_OPT
EXPRESSION_FIRST_CALCULATE -> EXPRESSION_HEADER ITEM
EXPRESSION_HEADER -> +
EXPRESSION_HEADER -> -
EXPRESSION_HEADER -> ^
EXPRESSION_MORE_OPT -> EXPRESSION_MORE_OPT EXPRESSION_MORE_OPT_CALCULATE
EXPRESSION_MORE_OPT_CALCULATE -> ADD_SUB_OPT ITEM
EXPRESSION_MORE_OPT -> ^
ITEM -> FACTOR ITEM_MORE_OPT
ITEM_MORE_OPT -> ITEM_MORE_OPT CALCULATE_ITEM
CALCULATE_ITEM -> MULTIPLY_DIVIDE_OPT FACTOR
ITEM_MORE_OPT -> ^
```

```
FACTOR -> identifier
FACTOR -> number
FACTOR -> ( EXPRESSION )
ADD_SUB_OPT -> +
ADD_SUB_OPT -> -
MULTIPLY_DIVIDE_OPT -> *
MULTIPLY_DIVIDE_OPT -> /
RELATION_OPT -> =
RELATION_OPT -> #
RELATION_OPT -> <
RELATION_OPT -> >=
RELATION_OPT -> >
RELATION_OPT -> <=
COMPOUND_STATEMENT -> begin STATEMENT STATEMENT_MORE end
STATEMENT_MORE -> ^
STATEMENT_MORE -> ; STATEMENT STATEMENT_MORE
CONDITION_STATEMENT -> if CONDITION then TRUE_OUT STATEMENT
TRUE_OUT -> ^
CALL_STATEMENT -> call identifier
WHILE_STATEMENT -> WHILE_BEGIN while CONDITION do TRUE_OUT STATEMENT
WHILE_BEGIN -> ^
READ_STATEMENT -> READ_FIRST READ_MORE )
READ_FIRST -> read ( identifier
READ_MORE -> , identifier READ_MORE
READ_MORE -> ^
WRITE_STATEMENT -> WRITE_FIRST WRITE_MORE )
WRITE_FIRST -> write ( EXPRESSION
WRITE_MORE -> , EXPRESSION WRITE_MORE_M WRITE_MORE
WRITE_MORE_M -> ^
WRITE_MORE -> ^
```

2. PL/0处理机介绍

PL/0处理机是一种假想的栈式处理机，使用一种类p-code的目标代码。PL/0处理机没有传统意义上的可随机访问的内存、通用寄存器等内容。他模拟的硬件结构和指令非常简单。

2.1 存储器

PL/0处理机有两种不同的存储结构：程序存储器（可随机访问任意存储单元）和数据存储（栈）。其中，每个过程在栈中对应一块运行栈，自底向上为返回地址RA、动态链DL、静态链SL、变量存储单元、数据区（存放指令执行过程中的临时数据）。其中，动态链记录调用者运行栈的基地址，静态链记录调用过程的直接外层运行栈的基地址。

PL/0的常量直接写入指令中。变量存储在运行栈的变量存储单元。内层过程通过指令指定的层次差，通过静态链逐层跳跃得到某个外层变量所在过程运行栈的基地址，而后通过指令中指定的偏移量获得变量所在的地址。

2.2 寄存器

寄存器名称	符号表示	存储内容
指令寄存器	I	存放当前正在解释的一条目标指令
程序地址寄存器	P	指向吓一跳要执行的目标指令
栈顶寄存器	T	指向当前过程运行栈的栈顶
基地址寄存器	B	指向当前过程运行栈的栈底

2.3 指令

PL/0的指令格式为：

f	l	a
功能码	层次差	操作数

其中，功能码表示指令类型，层次差表示该条指令所指示数据所在过程与当前过程在代码中层次的差异。

具体指令如下。

功能码	层次差	操作数	功能
LIT	0	a	将常数a取到栈顶
LOD	l	a	将层次差l、偏移量a所确定的变量取到栈顶
STO	l	a	将栈顶内容送到层次差l、偏移量a确定的变量，弹出栈顶
CAL	l	a	调用层次差为l， 代码区域过程地址为a的过程
INT	0	a	在运行栈中开辟大小为a的数据区
JMP	0	a	无条件跳转到地址a
JPC	0	a	当栈顶布尔值非真时，跳转到地址a
OPR	0	0	过程结束，执行返回相关操作
OPR	0	1	将栈顶元素取反
OPR	0	2	将次栈顶与栈顶相加，弹出两个元素，结果入栈
OPR	0	3	次栈顶减去栈顶， 弹出两个元素，结果入栈
OPR	0	4	次栈顶乘以栈顶， 弹出两个元素，结果入栈
OPR	0	5	次栈顶除以栈顶， 弹出两个元素，结果入栈
OPR	0	6	判断栈顶元素奇偶，弹出栈顶元素，结果入栈
OPR	0	7	空缺
OPR	0	k	判断次栈顶与栈顶关系，弹出两个元素，结果入栈，k=8-相等，9-不等，10-小于，11-大于等于，12-大于，13-小于等于
OPR	0	14	将栈顶值输出到屏幕， 弹出栈顶
OPR	0	15	屏幕输出换行
OPR	0	16	从命令行读入一个输入置于栈顶

3. PL/0语言编译器整体架构

本编译器共分为六大部分：词法分析程序、语法分析程序、语义分析程序、错误处理、表格管理、解释程序。其中，语法分析与语义分析在一个过程中完成、

我们使用C++作为主要语言进行设计。使用C++的原因之一是其丰富的特性支持可以为我们提供足够的特性支持，尤其是C++11以后，更多更高级的特性帮助我们更加高效、安全地开发易维护可扩展的大型程序，可以在未来对改编译器进行扩展；二是C++靠近底层，效率高。

本章节我们将从总体上介绍本编译器的结构和功能，下一章节我们将分模块介绍核心模块的具体实现。

3.1 词法分析程序

词法分析器将源代码逐个字符进行扫描，产生一个个单词序列交由编译器其它部分处理。

首先是单词的定义。我们在defs.h和defs.cpp中记录了单词的名称、类型等属性。可以非常方便的进行扩展和修改。词法分析程序将从这里读取单词属性，并据此识别单词。
我们定义了Token类来表示读到的单词，它包含了单词的基本属性和相应的获取、判定方法。词法分析的核心算法在类Lexerr中。管理类（Manage）可以向其中传递单词列表、错误信息等对象开启词法分析。
词法分析结束后，相关单词会存储在Token类型的列表中，传递给语法语义分析程序。

3.2 语法语义分析程序

语法分析的目的是分析单词序列的组成生成对应语法规则的短语，语义分析的目的是根据语法分析程序语义，按照一定的翻译模式，把目标语言翻译成目标代码。由于语法与语义的一致性，我们通常在语法分析的过程中运行翻译过程进行翻译。我们不再单独为语法分析单独抽象，而将它与语义分析耦合在一起。这样的耦合通常不影响程序的调试难度和可扩展性。

我们首先为生成式定义一定的对象。ProductionChar类定义了表达式中的变量，内含符号的类型（终结符、非终结符、空串、未定义）、名称、在表达式中的名称、属性。Production类抽象了产生式。它包括产生式左部（ProductionChar类型）、右部（ProductionChar类型的列表）、指向翻译模式运行语句的函数指针。

Parser类定义了语法分析的核心过程。包括生成式的导入、翻译模式的编写以及与表格管理、词法分析、错误处理等模块的交互。在执行语法分析的过程中调用相应归约语句的翻译函数生成目标代码。

3.3 错误处理

当词法、语法分析发生错误时，将会调用错误处理程序。在词法分析过程中，我们记录下了每个单词所在的行数。当出现错误时，将会调用错误处理程序中相应的处理函数，将错误信息加入到错误信息的字符串中。

3.4 表格管理

我们首先定义了表格管理项SymbolItem，包含项的类型、名称及相关属性。其中包含了回填功能，回填的核心逻辑将在具体实现中介绍。
SymbolTable类定义了表格类型，它记录了表格项列表，并允许插入常量、变量、过程记录项。同时，封装了回填功能。
SymbolTableManage是管理所有表格的类，包括新建表格、从表格栈中弹出表格、插入元素、查询表格项、回填等操作。
PL/0的表格是以栈的形式存储的。由于PL0/0规定，内层过程可以访问外层过程的变量但外层过程不能访问内层过程的局部常量和变量，但层过程的变量必须被建立（即在嵌套调用过程中外层过程应该被调用过），这也就意味着我们在做语法分析时，对于第l层过程的处理，同一层并列的多个过程仅需要保留当前存储的一个，而子过程还未分析过无需保存。因而l-1及之前层的过程在翻译时也只需保留一个，这些表格中存储子过程的信息可以被随时更新，因此，我们使用栈进行表格管理即可。

3.5 解释程序（虚拟机）

我们首先定义了Instruction类型来存储指令。VirtualMachine类定义了一个虚拟机。里面完整模拟了一个栈式存储结构虚拟机的指令流。除了给定的机器模式外，没有引入任何不属于该机器的运行模式和存储结构。

4. 具体实现解析及核心代码展示

4.1 词法分析程序

4.1.1 词法定义

```
//defs.h
#define SYM_comma 0
#define SYM_semicolon 1
#define SYM_dot 2
#define SYM_left_parenthesis 3
#define SYM_right_parenthesis 4

#define SYM_min_delimiter 0
#define SYM_max_delimiter 4

//...

enum class TokenType
{
    //Delimiter
    COMMASYM =  SYM_comma,
    SEMICOLONSYM = SYM_semicolon,
    DOTSYM = SYM_dot,
    LEFTPARENTHESISSYM = SYM_left_parenthesis,
    RIGHTPARENTHESISSYM = SYM_right_parenthesis,
    //...
 }

extern QString keyWord[];
```

```
//defs.cpp
String keyWord[] = {
        ",",            //0
        ";",            //1
        ".",            //2
        "(",            //3
        ")",            //4
        //...
}
```

我们使用一系列宏定义了单词类型的机内码，这些机内码分别对应TokenType枚举的类型值、对应类型名称的下标keyWord。其中，数字和标识符类型分别用number和identifier对应，其余类型用符号本身的名称对应。

4.1.2 Token类

```
class Token
{
public:

    Token();
    Token(uint64 _line, const QString& _string, TokenType _type);
    Token(uint64 _line, const QString& _string, TokenType _type, const QString&
_id);
    Token(uint64 _line, const QString& _string, TokenType _type, int64 _num);

    void set(uint64 _line, const QString& _string, TokenType _type);
```

```
    void set(uint64 _line, const QString& _string, TokenType _type, const
QString& _id);
    void set(uint64 _line, const QString& _string, TokenType _type, int64 _num);

    bool isDelimiter() const;
    bool isOperator() const;
    bool isKeyWord() const;
    bool isIdent() const;
    bool isNumber() const;
    bool isUndefine() const;

    TokenType getType() const;
    int64 getNum() const;
    uint64 getLine() const;
    QString getId() const;
    QString getString() const;

    void debugPrint() const;
  private:
    QString string;
    TokenType type;
    QString id;
    int64 num;
    uint64 line;
};
```

Token类包含了五个私有属性。string存储单词名，type存储单词类型（操作符、阶符和关键字各单独定义一种类型，它们的string域与类型对应的wordType相同，而number与identiier的string域为数字或标识符本身的名称，与wordType的字符串不同)。当类型为identifier时，id记录了标识符的名称，当类型为number时，num记录了数字的数值。line记录了单词所在代码的行数，用于错误处理。get开头的函数用于获取单词值，set用于设置，分别对应数字、标识符和其它三中类型。

由于实现显而易见，不再给出具体代码。

4.1.3 Lexer类

```
//lexicalanalyzer.h
class Lexer
{
public:
    Lexer();
    bool analyse();                          //进行词法分析
    void send(TokenList *_wordlist, ErrorInformation* _errorInformation, const
QString& _code);

    bool hasFinish();                        //是否分析完成
    void clear();                            //清空

    /*测试用*/
    void debugPrintTokenList();               //输出WordList

private:
    TokenList *tokenList;

    QString code;                            //代码

    ErrorInformation* errorInformation;      //存储错误信息
```

```
    std::tuple<int32, Token> analysisToken(int beginRank, uint64 line) const;
 //从beginRank开始(非空格、换行)，识别一个单词，返回终止下标和识别到的单词

    void sendTokenList(TokenList *_tokenList);
    void sendErrorInformation(ErrorInformation* _errorInformation);
    void sendCode(const QString& _code);      //输入代码

    bool hasAnalyze = false;                  //是否进行过词法分析

};
```

Lexer类包含三个私有属性：token列表，代码字符串，错误信息。其中，token列表、错误讯息均为指针，与编译器的其它部分共用。TokenList是使用typdef定义的vector<Token>类型。

Lexer类包含许多私有方法，sendTokenList、sendErrorInformation、sendCode用于向类中传递参数，公有方法send调用它们完成必要信息的传递。

analysisToken方法表示从beginRank开始寻找下一个单词，返回一个tuple<int, Token>的返回值，表示该token在code中的结束为止和token类型的一个实力。其中，tuple是C++11添加的新特性，用于存储多个不同类型的变量。analyse方法是词法分析的核心逻辑，不断调用analysisToken将code中的代码分析文单词序列存入tokenList中。下面展示analysisToken方法和analyse方法的核心代码。

```
//lexicalanalyzer.cpp
//....
std::tuple<int32, Token> Lexer::analysisToken(int beginRank, uint64 line) const
{
  Token token;
  QString string;
  int endRank = beginRank;

  //是否为数字
  if(this->code[beginRank].isNumber())
  {
    //如果第一个字符为数字，将它和它后面的数字解析为数字
    uint64 number = 0;
    for(;endRank < code.size() && beginRank + 9 > endRank;++ endRank)
    {
      if(this->code[endRank] < '0' || this->code[endRank] > '9')
      {
        -- endRank;
        break;
      }
    }

    //如果长度超过10，返回未定义
    if(beginRank + 9 < endRank)
    {
      this->errorInformation->addLexicalAnalyzerErrorNumberTooLong(line);
      token.set(line, string, (TokenType)SYM_undefine);
    }
    else
    {
      //截取数字转换为int存入wordlist
      string = code.mid(beginRank, endRank - beginRank + 1);
      token.set(line, string, (TokenType)SYM_number, string.toInt());
```

```
    }
    return std::make_tuple(endRank, token);
  }

  //是否为阶符
  for(int i = SYM_min_delimiter;i <= SYM_max_delimiter;++ i)
  {
    //越界
    if(beginRank + keyWord[i].size() > code.size())
    {
      continue;
    }
    if(code.mid(beginRank, keyWord[i].size()) == keyWord[i])
    {
      return std::make_tuple(beginRank + keyWord[i].size() - 1, Token(line,
keyWord[i], (TokenType)i));
    }
  }

  //是否为运算符
  for(int i = SYM_min_operator;i <= SYM_max_operator;++ i)
  {
    //越界
    if(beginRank + keyWord[i].size() > code.size())
    {
      continue;
    }
    if(code.mid(beginRank, keyWord[i].size()) == keyWord[i])
    {
      if(beginRank + keyWord[i].size() < code.size() && (keyWord[i] == ":" ||
keyWord[i] == "<" || keyWord[i] == ">"))
      {
        if(code[beginRank + keyWord->size()] == '=')
        {
          continue;
        }
      }
      return std::make_tuple(beginRank + keyWord[i].size() - 1, Token(line,
keyWord[i], (TokenType)i));
    }
  }

  //是否为关键字
  for(int i = SYM_min_keyword;i <= SYM_max_keyword;++ i)
  {
    //越界
    if(beginRank + keyWord[i].size() > code.size())
    {
      continue;
    }
    if(code.mid(beginRank, keyWord[i].size()) == keyWord[i])
    {
      //如果下一个字符既不是字母，也不是数字，或者没有下一个字符，则识别为关键字
      if(beginRank + keyWord[i].size() + 1 > code.size() ||
          (!code[beginRank + keyWord[i].size()].isLetterOrNumber()))
      {
        return std::make_tuple(beginRank + keyWord[i].size() - 1, Token(line,
keyWord[i], (TokenType)i));
```

```
      }
    }
  }

  //否则尝试识别为标识符
  if(!code[beginRank].isLetterOrNumber())
  {
    //如果开始字符不是数字或字母，不是合法标识符，报错
    this->errorInformation->addLexicalAnalyzerErrorIdentifierBegin(line);
    return std::make_tuple(beginRank + 1, Token(line, code[beginRank],
(TokenType)SYM_undefine));
  }

  for(;endRank < code.size();++ endRank)
  {
    if(!code[endRank].isLetterOrNumber())
    {
      -- endRank;
      break;
    }
  }
  return std::make_tuple(endRank, Token(line, code.mid(beginRank, endRank -
beginRank + 1), (TokenType)SYM_ident, code.mid(beginRank, endRank - beginRank +
1)));

}
//...
```

按照语法规定，标识符只能以字母开头且只能包含数字和字母，而关键字、运算符和阶符均不以数字开头。因此，以数字开头的单词必然是数字类型。我们首先尝试识别数字。如果beginRank位置字符是数字，那么我们就将它和它后面的数字解析为十进制数字存储。如果数字长度超过10，违反PL/0语法的规定，输出错误并返回。
之后，我们判断是否为阶符。由于阶符仅一个字符构成，因此只需遍历阶符比对即可。
而后我们执行运算符相关判断。由于部分运算符包含两个字符，且第一个字符本身也是一种运算符，因此当我们遇到这一类字符时，尝试超前搜索一个字符，判断是否为连续的一个运算符，如果是则将两个字符识别为一个运算符返回，否则只将当前字符识别为一个运算符返回。
然后我们执行标识符判断，标识符应当以字母开头，如若不然应是非法标识符，报错。
在上述识别后，要么报错返回，要么识别返回，如果均未返回，则出现了未定义单词和未定义错误，返回一个类型为未定义的token并向错误信息中加入一个未定义错误信息。

```
//lexicalanalyzer.cpp
//...
bool Lexer::analyse()
{
  if(this->hasAnalyze) return false;

  int line = 1;

  //去除前置空格回车
  for(auto c = code.begin();c != code.end();++ c)
  {
    if(*c == '\n') ++ line;
    if(*c == ' ' || *c == '\n' || *c == '\r' || *c == '\t')
    {
      c = code.erase(c, c+1);
    }
```

```
    else break;
  }

  int nowRank = 0;
  QChar& nowChar = code[nowRank];
  for(;nowRank < code.size();++ nowRank)
  {
    nowChar = code[nowRank];
    if(nowChar == '\n') ++ line;
    else if(nowChar != ' ' && nowChar != '\r' && nowChar != '\t')
    {
      auto [newRank, tmpWord] = this->analysisToken(nowRank, line);
      this->tokenList->push_back(tmpWord);
      nowRank = newRank;
    }
  }

  this->hasAnalyze = true;
  return true;
}
//...
```

analyse方法首先去除前置的空格回车等无效字符，找到第一个有效字符，开始不断调用analysisToken方法查找单词。analysisToken方法接受开始位置，返回结束位置和一个单词，analyse方法将单词记入单词表，并从返回的结束位置开始继续执行analysisToken方法，如此反复循环直到结束位置达到code的末尾未知。期间，每次遇到换行符，行数都要加一，为每一个单词记录行数，以便错误处理时输出错误位置。

4.2 表格管理程序

4.2.1 SymbolTableItem类

```
// symboltableitem.h
// ...
enum class SymbolType
{
  CONSTANT = 0,
  VARIABLE = 1,
  PROCEDURE = 2,
  NONE = 3,
};

class SymbolTableItem
{
public:
  SymbolTableItem();
  SymbolTableItem(const QString &name, SymbolType type, int value, int level,
int address, int size, bool needFillAddress, int fillAddressNext, bool
needFillSize, int fillSizeNext);

  const QString &getName() const;
  void setName(const QString &newName);

  SymbolType getType() const;
  void setType(SymbolType newType);
```

```
    int getValue() const;
    void setValue(int newValue);

    //...
    //此处省略一系列私有成员的get与set函数
    //...

private:
    QString name;
    SymbolType type;
    int value;
    int level;
    int address;
    int size;

    bool needFillAddress;
    int fillAddressNext;

    bool needFillSize;
    int fillSizeNext;

};

// ...
```

SymbolTableItem定义了每个表项的信息。我们定义了SymbolType枚举类型，用于定义每个表项的类型，包括常数、变量、过程和未定义四种类型，未定义表示该表项为空，是一个非法无效表项。该类包含10个私有成员。name记录了表项的名称，可能是方法名、变量名、数字的字符串表示。type记录了表项的类型。level记录了层次差（仅变量类型和过程类型有效），address记录了变量在过程运行栈变量存储区的偏移地址（仅变量类型有效），size记录了过程变量的个数+3（仅过程类型有效，用于开辟运行栈固有内存空间，+3是为了预留三个位置，分别存储返回地址、静态链和动态链）。
这里着重介绍needFillAdress、needFileSize、fillAddressNext和fillSizeNext四个私有属性。这四个私有属性均与回填有关。needFillAdress表示地址属性需要被回填，needFileSize表示size属性需要被回填，需要被填的表项的对应adress或size属性均为-1，fileeAddressNext和fillSizeNext指向了需要回填的代码地址，如果没有则为-1。回填时，needFillAdress或needFileSize被置为false，对应address或size被置为回填值。从filllAddressNext或fillSizeNext获得第一个需要回填的代码地址，该代码的数据部分存放着下一个需要回填的代码地址，如此便利下去，最后一个需要回填的代码数值部分为-1，停止回填。

以上介绍了回填过程，后续将结合其它部分介绍如何实现回填链表的填写。

SymbolTableItem的大部分公有方法用于获取和设置各个私有成员的值，在此不再赘述。

4.2.2 SymbolTable类

```
//symboltable.h
//...
class SymbolTable
{
public:
    SymbolTable(const QString &newProcedureName);

    void insertConstant(const QString& name, int value);
    void insertVariable(const QString& name, int level);
    void insertProcedure(const QString& name, int level, int address, int size);
```

```
  SymbolTableItem* search(const QString& name);
  int searchAddress(const QString& name, int instructionNumber);
  int searchSize(const QString& name, int instructionNumber);

  int refillAddress(const QString& name, int address);
  int refillSize(const QString& name, int size);


  const QString &getProcedureName() const;

  void setProcedureName(const QString &newProcedureName);


private:
  std::vector<SymbolTableItem> itemList;
  int offset;
  QString procedureName;        //表所代表过程的名字
};
//...
```

SymbolTable类记录了一个过程所对应的符号表。私有属性itemList是一个SybolTableTtem类型的列表。offset存储了当前最大变量地址偏移量+1（初始值为3，0、1、2分别存储返回地址、静态链和动态链）。procedureName存储了当前符号表所对应的过程的过程名。三个以insert开头的公有方法分别用于插入常量、变量和过程类型的表格信息。search函数用于返回无需回填的表格属性。

接下来我们介绍与回填有关的四个方法：searchAddress、searchSize、refillAddress和refillSize。

```
//symboltable.cpp
//...
int SymbolTable::searchAddress(const QString &name, int instructionNumber)
{
  for(auto& i : this->itemList)
  {
    if(i.getName() == name)
    {
      if(i.getNeedFillAddress())
      {
        int re = i.getFillAddressNext();
        i.setFillAddressNext(instructionNumber);
        return re;
      }
      else
      {
        return i.getAddress();
      }
    }
  }
  return -1;
}

int SymbolTable::searchSize(const QString &name, int instructionNumber)
{
  for(auto& i : this->itemList)
  {
    if(i.getName() == name)
    {
      if(i.getNeedFillSize())
```

```
      {
        int re = i.getFillSizeNext();
        i.setFillSizeNext(instructionNumber);
        return re;
      }
      else
      {
        return i.getSize();
      }
    }
  }
  return -1;
}

int SymbolTable::refillAddress(const QString &name, int address)
{
  for(auto& i : this->itemList)
  {
    if(i.getName() == name)
    {
      if(i.getNeedFillAddress())
      {
        i.setNeedFillAddress(false);
        int re = i.getFillAddressNext();
        i.setFillAddressNext(-1);
        i.setAddress(address);
        return re;
      }
    }
  }
  return -1;
}

int SymbolTable::refillSize(const QString &name, int size)
{
  for(auto& i : this->itemList)
  {
    if(i.getName() == name)
    {
      if(i.getNeedFillSize())
      {
        i.setNeedFillSize(false);
        int re = i.getFillSizeNext();
        i.setFillSizeNext(-1);
        i.setSize(size);
        return re;
      }
    }
  }
  return -1;
}
//...
```

refillAddress函数和reflleesize函数完成回填，回填过程在4.2.1节有详细阐释。在4.2.1节我们留下了关于回填的最后一个问题：如何构建回填序列。在这里，searchAddress和searchSize完成了这个功能，我们以searchAddress为例进行讲解。

searchAddress接受两个参数：name和instructionNumber，前者代表所查标识符名，后者表示返回值将被填到的指令的地址。我们首先在该表中查找对应name的表项，找不到则返回-1（一般出现语义错误才会有这样的返回值）。找到名称后，查看是否需要回填，如果不需要回填，说明Address已经是正确值，直接返回正确值即可，这个正确值将被直接填写到目标代码中。如果需要回填，说明此事fillAddressNext中存储的是回填链表的表头（-1表示没有建立链表），我们把指令地址插入表头，并返回原表头。这样，当前表项指向了地址为instructionNumber的目标代码，而该目标代码的数据区又指向了原表头代表的需要回填的代码，构成回填链表。

4.2.3 SymbolTableManage类

```
//symboltablemanage.h
//...
class SymbolTableManage
{
public:
  SymbolTableManage();
  SymbolTableManage(ErrorInformation *_errorInformation);

  void pushNewTable(const QString& name);
  void popDeleteTable();

  void enterConstant(const QString& name, int value);
  void enterVariable(const QString& name);
  void enterProcedure(const QString& name, int address, int size);

  void sendErrorInformation(ErrorInformation *_errorInformation);

  SymbolTableItem* search(const QString& name);
  int searchAddress(const QString& name, int instructionNumber);
  int searchSize(const QString& name, int instructionNumber);

  int refillAddres(const QString& name, int address);
  int refillSize(const QString& name, int size);

  QString getTopTableName();

  int getNowLevel();

private:
  std::vector<SymbolTable> symbolTableList;
  ErrorInformation *errorInformation;
};
//...
```

SymbolTableManage是所有表格的管理类。它包含了表格栈和错误信息。pushNewTable方法用于建立新表，popDeleteTable方法用于删除旧表。其它方法与SymbolTable类似，只是在查询时，从栈顶到栈底倒序遍历符号表进行查询，以做到重名时，调用最近层次的信息；在填写时，直接填入栈顶符号表。这些操作符合通常的嵌套过程的语义定义。

4.3 语法语义分析程序

4.3.1 ProductionChar类

```
//productionchar.h
//...
typedef std::vector<int> Attribute;

#define PRODUCTION_TYPE_NONE 0
#define PRODUCTION_TYPE_TERMINAL 1
#define PRODUCTION_TYPE_UNTERMINAL 2
#define PRODUCTION_TYPE_EMPTY 3


enum class ProductionCharType
{
  NONE = PRODUCTION_TYPE_NONE,
  TERMINAL = PRODUCTION_TYPE_TERMINAL,
  UNTERMINAL = PRODUCTION_TYPE_UNTERMINAL,
  EMPTY = PRODUCTION_TYPE_EMPTY,
};

class ProductionChar
{
public:
  ProductionChar(const ProductionCharType& _type, const QString& _name, const
QString& _tokenName);
  ProductionChar(const int& _type, const QString& _name, const QString&
_tokenName) : ProductionChar((ProductionCharType) _type, _name, _tokenName) {}
  ProductionChar();

  ProductionCharType getType() const;
  QString getName() const;
  Attribute getAttribute(const QString& name) const;
  bool empty() const;

  void setType(const ProductionCharType& _type);
  void setType(const int& _type);
  void setName(const QString& _name);
  void setAttribute(const QString& name, const Attribute& _attribute);
  QString toString() const;


  bool operator==(const ProductionChar& b) const;
  bool operator!=(const ProductionChar& b) const;

  const QString &getTokenName() const;
  void setTokenName(const QString &newTokenName);

private:
  ProductionCharType type;
  QString name;
  QString tokenName;

  std::map<QString, Attribute> attribute;   //综合属性。pair<属性名，属性值>
};

typedef std::vector<ProductionChar> ProductionCharList;
//...
```

我们定义属性类型为std::vector<int>。使用这个类型的原因是int型的列表可以编码几乎所有类型的属性。尽管在PK/0翻译模式设计中我们只使用到了整型属性，但为了程序的可扩展性，我们还是使用了更加灵活的int型列表作为属性的存储结构。
接下来是ProductionCharType类型的枚举类，表示了表达式字符的类型：未定义、终结符、非终结符、空串。
ProductionChar类定义了表达式中的字符。它包含四个私有属性：类型type、该字符的名称name、字符所对应单词的额名称tokenName和属性列表。其中，只有当字符名称为"identifier"（表示标识符）和""number"（表示数字）时，tokenName会存储标识符字符串或数字字符串，与name不同，其它时候与name相同。属性列表使用了map<QString, Attribute>类型，表示从属性名到属性值的一个映射。
ProductionChar类提供了许多公有方法。大多是普通的get与set方法，可以很容易地从函数名中推测出其功能和实现，在这里不过多赘述。

4.3.2 Production类

```
//production.h
//...
class Production
{
public:
  Production(const ProductionChar& _left, const ProductionCharList& _right,
const std::function<void(ProductionChar &left, std::vector<ProductionChar>&)>&
_func);
  Production();

  ProductionChar getLeft() const;
  ProductionCharList getRight() const;
  std::function<void(ProductionChar &left, std::vector<ProductionChar>&)>
getfunc() const;

  void setLeft(const ProductionChar& _left);
  void setRight(const ProductionCharList& _right);
  void setFunc(const std::function<void(ProductionChar &left,
std::vector<ProductionChar>&)>& _func);


  bool empty() const;

private:
  ProductionChar left;
  ProductionCharList right;

  std::function<void(ProductionChar &left, std::vector<ProductionChar>&)> func;
};

typedef std::vector<Production> ProductionList;
//...
```

Production抽象了产生式。它包含三个私有成员：ProductionChar类型的left，表示产生式左部符号；ProductionCharList类型的right，表示产生式右部的符号串，以及function<void(ProductionChar &left, std::vector<ProductionChar>&)>类型的func，表示翻译函数，即使用该产生式归约时执行的翻译函数指针。这里使用了C++14提供的新特性function，用于存储函数指针，这个函数的返回值是void，有两个参数，一个参数是ProductionChar类型的产生式左部的引用，一个是ProductionChar列表类型的产生式右部的引用。传入引用的原因是让翻译模式对左右两边属性值的修改可以保留。
Production的成员方法主要用来获取和修改私有方法，可以很容易从方法名得知他们的功能和操作，这里不再过多介绍。

4.3.3 Parser类

```
//parser.h
//...
class Parser
{
public:
  Parser();
  Parser(const TokenList* _tokenList, ErrorInformation* _errorInformation,
InstructionList* _instrunctionList);

  void sendTokenList(const TokenList* _tokenList);                              //
传入Token List
  void sendErrorInformation(ErrorInformation* _errorInformation);               //
传入Error Information
  void sendInstructionList(InstructionList* _instructionList);                  //
传入Instruction List

  bool empty();                                                                 //
是否未传入Token List,Error Information,Instruction List

  void build();                                                                 //
创建PL0相关生成式及翻译模式、LR分析表

  void debugPrintInstruction();

private:
  const TokenList* tokenList = nullptr;
  ErrorInformation* errorInformation = nullptr;

  enum class ParsingTableItemType
  {
    STATE = 0,
    PRODUCTION = 1,
    GOTO = 2,
    SUCCESS = 3
  };


  class ParsingTableItem
  {
  public:
    ParsingTableItemType type;
    int id;
  };

  //LR分析表管理
  std::vector<std::pair<std::pair<LRState, ProductionChar>, ParsingTableItem>>
parsingTable;

  void insertTableItem(const LRState& _state, const ProductionChar& _variable,
const ParsingTableItem& _item);
  void insertTableItem(const LRState& _state, const ProductionChar& _variable,
const ParsingTableItemType& _type, int _id);
  ParsingTableItem searchParsingTable(const LRState& _state, const
ProductionChar& _terminal);
```

```
  void buildParsingTable();


  //语法&语义分析管理
  ProductionList productionList;
  int nxq = 0;

  int insertProduction(const Production& _production);      //创建生成式，返回生成式
id
  int insertProduction(const QString& _leftName, const
std::vector<std::pair<int, QString>>& _right,
                       std::function<void(ProductionChar &left,
std::vector<ProductionChar>&)> const & _func); //创建生成式，返回生成式id

  void buildProduction();
  void runTranslator();


  //表格管理
  SymbolTableManage symbolTableManage;

  //代码生成管理
  InstructionList* instructionList = nullptr;
  void gen(int address, InstructionType type, int levelDiff, int operand);
  void reFill(int nxq, int value);
};
//...
```

语法语义分析使用一个类Parser表示。其中耦合了语法分析相关类并加入了表格管理。这似乎是不符合面向对象程序设计规范的。但实际上，语法语义分析过程相对固定，其它部分处理好后，仅需固定的LR分析法运行相应归约和翻译模式即可。没有其它拓展的需要。即便高度耦合，但在编译器的拓展之中几乎无需直接修改Parser类的核心部分，只需加入、删除、修改翻译模式和表达式即可，因此，这样的耦合并不会带来什么麻烦，反而会让翻译过程的编写更加的直接、方便而无需过多的嵌套层次调用，使得语法语义分析部分过于冗杂。

Parser类包含多个私有属性。tokenList记录了词法分析产生的单词表，errorInformation指针指向错误处理信息实例，symbolTableManage是表格管理类的一个实力，instructionList存储了语义分析产生的指令序列。

Parser类汇总定义了一些内部类和枚举类来进行词法分析。我们选择了LR分析法，构造出SLR分析表。将其存储在文件grammar中读取。ParsingTableItemType枚举类型存储了分析表项的类型，包括状态（Action sx）、归约表达式(Action rx)、Goto、success四中。ParsingTableItem类顶一个一个表项，它包含表项的类型和值。由于表格是稀疏的，我们是用vector存储，每一个vector元素是一个表格项，类型为<pair<LRState, ProductionChar>, ParsintTableItem>。LRState表示LR分析时所g处的状态值，是一个int类型的冲定义，第一个二元组<LRState, ProductionChar>表示栈顶状态和语法符号，第二个元素ParsingTableItem表示其对应的表项。相关的insertTableItem和searchTableItem表示插入和查找表项，buildParsintTable()在文件中读取表项并加入SLR分析表中。

在代码生成部分，gen方法直接生成代码，address表示代码地址，type、levelDiff和oprand分别表示了指令类型、层次差和操作数。reFill方法用于地址回填，第一个参数指示了待回填的代码地址头，第二个参数表示回填值。

```
//parser.cpp
//...
void Parser::gen(int address, InstructionType type, int levelDiff, int operand)
{
```

```
  if(address > this->instructionList->size())
  {
    qDebug() << "Panic: instruction address is larger than it should be!\n";
    return;
  }

  /*替换or新建*/
  if(address < this->instructionList->size())
  {
    (*(this->instructionList))[address] = Instruction(type, levelDiff, operand);
  }
  else //==
  {
    (*(this->instructionList)).push_back(Instruction(type, levelDiff, operand));
  }
}

void Parser::reFill(int nxq, int value)
{
  while(nxq != -1)
  {
    int tmp = (*(this->instructionList))[nxq].getOperand();
    (*(this->instructionList))[nxq].setOperand(value);
    nxq = tmp;
  }
}

//...
```

下面介绍语法分析部分。buildProduction()用于构建生成式，在这里面我们使用insertProduction方法插入了所需全部生成式。它包含三个参数：左部符号，右部符号列表和翻译函数。得益于面向对象强大的特性和C++11支持的Lambda表达式，我们可以很容易的插入一个表达式，下面给出一个实例：

```
//parser.cpp
//...
  //18: PROCEDURE_PART -> PROCEDURE_MORE PROCEDURE_HEADER BLOCK ;
  this->insertProduction("PROCEDURE_PART", std::vector<std::pair<int, QString>>{

 std::make_pair(PRODUCTION_TYPE_UNTERMINAL, "PROCEDURE_MORE"),

 std::make_pair(PRODUCTION_TYPE_UNTERMINAL, "PROCEDURE_HEADER"),

 std::make_pair(PRODUCTION_TYPE_UNTERMINAL, "BLOCK"),

 std::make_pair(PRODUCTION_TYPE_TERMINAL, ";")},
                         [=](ProductionChar &left, std::vector<ProductionChar>
&right)->void{
                           //qDebug() << "归约：PROCEDURE_PART -> PROCEDURE_MORE
PROCEDURE_HEADER BLOCK ;\n";

                           //qDebug() << "beginAddr:" <<
right[2].getAttribute("beginAddr")[0];
                           //qDebug() << "size:" << right[2].getAttribute("size")
[0];

                           int value1 = right[2].getAttribute("beginAddr")[0];
```

```
                            int t1 = this->symbolTableManage.refillAddres(this-
>symbolTableManage.getTopTableName(), value1);
                            this->reFill(t1, value1);

                            int value2 = right[2].getAttribute("size")[0];
                            int t2 = this->symbolTableManage.refillSize(this-
>symbolTableManage.getTopTableName(), value2);
                            this->reFill(t2, value2);

                            this->symbolTableManage.popDeleteTable();
                          });
//...
```

这里插入了表达式**PROCEDURE_PART -> PROCEDURE_MORE PROCEDURE_HEADER BLOCK ;**。第一个参数是左部名称，第二个参数是右部类型和名称，第三个参数使用lambda表达式声明了在使用该表达式进行归约是执行的翻译操作，这个匿名函数的参数代表了一个表达式的左部和右部序列。注意，插入表达式时生成的表达式仅作为信息使用，而匿名函数的参数是在语法语义分析过程中生成的，是存储在语法符号栈中的，注意区分。

buildParsingTable()创建了SLR分析表。在build()函数中运行了全部创建过程，创建完毕后，运行runTranslator方法，它用来执行语法分析，并顺带进行语义分析。

```
//parser.cpp
//...
void Parser::runTranslator()
{
  std::vector<LRState> stateStack;
  std::vector<ProductionChar> symbolStack;
  std::vector<ProductionChar> inputStack;

  stateStack.push_back(0);
  symbolStack.push_back(ProductionChar(PRODUCTION_TYPE_TERMINAL, "$", "$"));
  inputStack.push_back(ProductionChar(PRODUCTION_TYPE_TERMINAL, "$", "$"));
  for(auto tokenPoint = tokenList->rbegin();tokenPoint != tokenList->rend();++
tokenPoint)
  {
    auto& token = *tokenPoint;
    QString tokenName = token.getString();
    if(token.isIdent()) tokenName = "identifier";
    if(token.isNumber()) tokenName = "number";
    auto tmp = ProductionChar(PRODUCTION_TYPE_TERMINAL, token.getString(),
tokenName);
    tmp.setAttribute("line", Attribute{(int)token.getLine()});
    inputStack.push_back(tmp);
  }

  while(true)
  {
    //printState(stateStack);

    LRState nowState = stateStack[stateStack.size() - 1];
    ProductionChar nowChar = inputStack[inputStack.size() - 1];



    ParsingTableItem item = this->searchParsingTable(nowState, nowChar);
```

```
    //qDebug() << "nowState: " << (int)nowState << "\n";
    //qDebug() << "nowChar TokenName: " << nowChar.getTokenName() << "\n";
    //qDebug() << "item type: " << (int)item.type << ", item id:" << item.id <<
"\n";


    if(item.type == ParsingTableItemType::STATE)
    {
      stateStack.push_back(item.id);
      symbolStack.push_back(nowChar);
      inputStack.pop_back();
      //printSymbol(symbolStack);
    }
    else if(item.type == ParsingTableItemType::PRODUCTION)
    {
      const Production &nowProductionInfor = productionList[item.id];

      ProductionChar newChar(PRODUCTION_TYPE_UNTERMINAL,
nowProductionInfor.getLeft().getName(),
nowProductionInfor.getLeft().getTokenName());

      std::vector<ProductionChar> nowRight;

      for(int i = symbolStack.size() - nowProductionInfor.getRight().size();i <
symbolStack.size();++ i)
      {
        nowRight.push_back(symbolStack[i]);
      }

      auto f = nowProductionInfor.getfunc();
      f(newChar, nowRight);

      for(int i = 0;i < nowProductionInfor.getRight().size();++ i)
      {
        symbolStack.pop_back();
        stateStack.pop_back();
      }
      symbolStack.push_back(newChar);
      //printSymbol(symbolStack);

      nowState = stateStack[stateStack.size() - 1];

      item = this->searchParsingTable(nowState, newChar);
      if(item.type != ParsingTableItemType::GOTO)
      {
        //printState(stateStack);
        qDebug() << "Wrong!归约后item的类型不是GOTO! \n";
        qDebug() << "nowState: " << (int)nowState << "\n";
        qDebug() << "newChar TokenName: " << newChar.getTokenName() << "\n";
        qDebug() << "newChar Name: " << newChar.getName() << "\n";
        qDebug() << "newChar Type: " << (int)newChar.getType() << "\n";

        break;
      }
      stateStack.push_back(item.id);
      //this->debugPrintInstruction();
    }
```

```
    else if(item.type == ParsingTableItemType::SUCCESS)
    {
      qDebug() << "Success!\n";
      break;
    }
  }

  //this->debugPrintInstruction();
}
//...
//
```

首先创建状态栈、语法符号栈，并读取token列表获得输入符号栈。每次从状态栈中读取一个状态，根据输入符号栈内容查询SLR分析表，根据读到的表项类型决定下一步的运行动作。如果是STATE类型，则跳转到下一个状态；如果是PRODUCTION类型，则用其指向的产生式归约并执行翻译函数，继续查找GOTO到下一个状态。

语义分析部分我们是用insertProduction函数插入产生式和翻译模式。下面给出翻译模式的伪代码。实际代码过多不一一列举，具体形式可参考上文中调的插入表达式的实例并阅读相关代码。

```
//parser.cpp
//...
/*
   * 程序:E
   * 分程序：BLOCK
   * 常量说明部分：CONST_PART
   * 变量说明部分：VARIABLE_PART
   * 过程说明部分：PROCEDURE_PART
   * 语句：STATEMENT
   * 常量定义：CONST_DEFINE
   * 标识符：identifier
   * 无符号整数：identifier
   * 过程首部：PROCEDURE_HEADER
   * 赋值语句：ASSIGN_STATEMENT
   * 条件语句：CONDITION_STATEMENT
   * 当型循环语句：WHILE_STATEMENT
   * 过程调用语句：CALL_STATEMENT
   * 读语句：READ_STATEMENT
   * 写语句：WRITE_STATEMENT
   * 复合语句：COMPOUND_STATEMENT
   * 表达式：EXPRESSION
   * 条件：CONDITION
   * 关系运算符：RELATION_OPT
   * 项：ITEM
   * 加减运算符：ADD_SUB_OPT
   * 因子：FACTOR
   * 乘除运算符：MULTIPLY_DIVIDE_OPT
   * 空：^
 一些属性名及作用：
 nxq:最后一条指令的地址
 needFillAddr:待重填指令的地址
 beginAddr:过程第一条指令的开始地址
 size:变量个数
 一些方法名及作用：
 getName():返回非终结符的名称（不在Attribute里,identifier的名称就是代码中的id名，number的名称就是代码中的数字的字符串表示）
```

gen(address, type, levelDiff, operand):三地址代码，address表示指令地址，type表示操作类型，levelDiff表示层次差，operand表示操作数

pushNewTable():在符号表栈中压入并新建一个符号表
popDeleteTable():在符号表栈中弹出并删除一个符号表
enterConstant(name, value)
enterVariable(name)
enterProcedure(name, address, size)
search(name):返回一个item，它包含若干属性：name, type(PROCEDURE、VARIABLE、CONSTANT), value(常量值), level(变量、过程所在层次), address(变量相对地址，过程跳转地址), size(过程中变量个数+3)
refillAddress(name, address):查找表中name过程名，回填address
refillSize(name, size):查找表中name过程名，回填size

```
E' -> E
E -> EM BLOCK_FIRST  {gen(0, JMP, 0, BLOCK_FIRST.beginAddr)}
                     重填跳转到主程序入口
EM -> ^ {nxq = 0;
         pushNewTable("0main");
         gen(nxq, JMP, 0, -1); }
        设置第一条指令位置，新建主程序的符号表，填写跳转到主程序执行过程的指令（待重填）
BLOCK_FIRST -> BLOCKM_FIRST STATEMENT {
         BLOCK_FIRST.beginAddr = BLOCKM_FIRST.beginAddr;
         nxq = nxq + 1;
         gen(nxq, OPR, 0, 0);}
         填写第一条指令位置;生成返回指令
BLOCKM_FIRST -> CONST_PART' VARIABLE_PART' PROCEDURE_PART'  {
             nxq = nxq + 1;
             gen(nxq, INT, 0, VARIABLE_PART'.size);
             BLOCKM_FIRST.beginAddr = nxq;}
         生成BLOCKM_FIRST的第一条指令INT，开辟数据空间；设置第一条指令的地址
CONST_PART' -> CONST_PART
CONST_PART' -> ^
VARIABLE_PART' -> VARIABLE_PART    {VARIABLE_PART'.size = VARIABLE_PART.size;}
VARIABLE_PART' -> ^                {VARIABLE_PART'.size = 3;}
PROCEDURE_PART' -> PROCEDURE_PART
PROCEDURE_PART' -> ^

CONST_PART -> const CONST_DEFINE CONST_MORE ;
CONST_MORE ->  ,CONST_DEFINE CONST_MORE
CONST_MORE ->  ^
CONST_DEFINE -> identifier = number {enterConstant( identifier.getName(),
str2int(number.getName()) )}
                  将常量加入栈顶符号表
VARIABLE_PART -> var identifier VARIABLE_MORE ;
{enterVariable(identifier.getName()); VARIABLE_PART.size = VARIABLE_MORE.size +
1;}
                  将变量加入符号表,计算变量个数
VARIABLE_MORE -> , identifier VARIABLE_MORE
{enterVariable(identifier.getName()); VARIABLE_MORE_Left.size = 1 +
VARIABLE_MORE_Right.size;}
                   将变量加入符号表,计算变量个数
VARIABLE_MORE -> ^ {VARIABLE_MORE.size = 3;}
                  设置变量个数初始值

PROCEDURE_PART -> PROCEDURE_MORE PROCEDURE_HEADER BLOCK ;  {
                     refillAddress(getTopTableName(),BLOCK.beginAddr);
                     refillSize(getTopTableName(), BLOCK.size);
                     popDeleteTable();
```

```
                    }
                  回填Address和size;弹出并移除栈顶符号表;
 PROCEDURE_MORE -> PROCEDURE_PART
 PROCEDURE_MORE -> ^
 PROCEDURE_HEADER -> procedure identifier ; {
                   enterProcedure(identifier.getName(), -1, -1);
                   pushNewTable(identifier.getName());}
                   将过程加入符号表;新建一个符号表,压入栈顶
 BLOCK -> BLOCKM STATEMENT {
            BLOCK.beginAddr = BLOCKM.beginAddr;
            nxq = nxq + 1;
            gen(nxq, OPR, 0, 0);
            BLOCK.size = BLOCKM.size}
            填写该BLOCK第一条指令位置;生成返回指令;记录size
 BLOCKM -> CONST_PART' VARIABLE_PART' PROCEDURE_PART'  {
             BLOCKM.size = VARIABLE_PART'.size;
             nxq = nxq + 1;
             gen(nxq, INT, 0, searchSize(getTopTableName, nxq));
             BLOCKM.beginAddr = nxq;}
             填写当前过程变量大小;生成该BLOCK的第一条指令INT，开辟数据空间;设置第一条指令的
地址
 STATEMENT -> ASSIGN_STATEMENT
 STATEMENT -> CONDITION_STATEMENT
 STATEMENT -> WHILE_STATEMENT
 STATEMENT -> CALL_STATEMENT
 STATEMENT -> READ_STATEMENT
 STATEMENT -> WRITE_STATEMENT
 STATEMENT -> COMPOUND_STATEMENT
 STATEMENT -> ^
 ASSIGN_STATEMENT -> identifier := EXPRESSION
                     {
                       item = search(identifier.getName());
                       nxq = nxq + 1;
                       gen(nxq, STO, getNowLevel() - item.getLevel(),
item.getAddress());
                     }

 CONDITION -> EXPRESSION RELATION_OPT EXPRESSION
              {
                nxq = nxq + 1;
                gen(nxq, OPR, 0, RELATION_OPT.type);
              }
 CONDITION -> odd EXPRESSION
              {
                nxq = nxq + 1;
                gen(nxq, OPR, 0, 6);
              }

 EXPRESSION -> EXPRESSION_FIRST_CALCULATE EXPRESSION_MORE_OPT
 EXPRESSION_FIRST_CALCULATE -> EXPRESSION_HEADER ITEM
                               {
                                 if(EXPRESSION_HEADER.type == 1)
                                 {
                                     nxq = nxq + 1;
                                     gen(nxq, OPR, 0, 1);
                                 }
                               }
 EXPRESSION_HEADER -> + {EXPRESSION_HEADER.type = 0;}
```

```
 EXPRESSION_HEADER -> - {EXPRESSION_HEADER.type = 1;}
 EXPRESSION_HEADER -> ^ {EXPRESSION_HEADER.type = 0;}
 EXPRESSION_MORE_OPT -> EXPRESSION_MORE_OPT_CALCULATE EXPRESSION_MORE_OPT
 EXPRESSION_MORE_OPT_CALCULATE -> ADD_SUB_OPT ITEM
                                  {
                                    nxq = nxq + 1;
                                    gen(nxq, OPR, 0, ADD_SUB_OPT.type);
                                  }
 EXPRESSION_MORE_OPT -> ^
 ITEM -> FACTOR ITEM_MORE_OPT
 ITEM_MORE_OPT -> ITEM_MORE_OPT CALCULATE_ITEM

 CALCULATE_ITEM -> MULTIPLY_DIVIDE_OPT FACTOR
                   {
                     nxq = nxq + 1;
                     gen(nxq, OPR, 0, MULTIPLY_DIVIDE_OPT.type);
                   }
 ITEM_MORE_OPT -> ^

 FACTOR  -> identifier      {item = search(identifier.getString());nxq = nxq
+1;gen(nxq, LOD, getNowLevel() - item.level, item.address);} 将操作数放到栈顶
 FACTOR  -> number          {nxq = nxq + 1;gen(nxq, LIT, 0,
number.getString()).toInt());} 将常数放到栈顶
 FACTOR  -> ( EXPRESSION )
 ADD_SUB_OPT -> + {ADD_SUB_OPT.type = 2;}
 ADD_SUB_OPT -> - {ADD_SUB_OPT.type = 3;}
 MULTIPLY_DIVIDE_OPT -> * {MULTIPLY_DIVIDE_OPT.type = 4;}
 MULTIPLY_DIVIDE_OPT -> / {MULTIPLY_DIVIDE_OPT.type = 5;}
 RELATION_OPT -> = {RELATION_OPT.type = 8;}
 RELATION_OPT -> # {RELATION_OPT.type = 9;}
 RELATION_OPT -> < {RELATION_OPT.type = 10;}
 RELATION_OPT -> >= {RELATION_OPT.type = 11;}
 RELATION_OPT -> > {RELATION_OPT.type = 12;}
 RELATION_OPT -> <= {RELATION_OPT.type = 13;}
 COMPOUND_STATEMENT -> begin STATEMENT STATEMENT_MORE end
 STATEMENT_MORE -> ; STATEMENT STATEMENT_MORE
 STATEMENT_MORE -> ^

 CONDITION_STATEMENT -> if CONDITION then TRUE_OUT STATEMENT
                        {
                          gen(TRUE_OUT.nxq, JPC, 0, nxq + 1)
                        }
 TRUE_OUT -> ^
             {
               nxq = nxq + 1;
               gen(nxq, JPC, 0, -1);
               TRUE_OUT.nxq = nxq;
               TRUE_OUT.address = nxq + 1;
             }
 CALL_STATEMENT  -> call identifier
                   {
                     nxq = nxq + 1;
                     item = search(identifier.getName());
                     gen(nxq, CAL, getNowLevel() - item.getLevel(),
searchAddress(identifier.getName(), nxq))
                   }

 WHILE_STATEMENT -> WHILE_BEGIN while CONDITION do TRUE_OUT STATEMENT
```

```
                  {
                    nxq = nxq +1;
                    gen(nxq, JMP, 0, WHILE_BEGIN.nxq);
                    gen(TRUE_OUT.nxq, JPC, 0, nxq + 1);
                  }
 WHILE_BEGIN -> ^
{
  WHILE_BEGIN.nxq = nxq + 1;
}
 READ_STATEMENT -> READ_FIRST READ_MORE )
 READ_FIRST -> read( identifier
              {
                item = search(identifier.getName())
                nxq = nxq + 1;
                gen(nxq, OPR, 0, 16);
                nxq = nxq + 1;
                gen(nxq, STO, getNowLevel() - item.getLevel(),
item.getAddress());
              }
 READ_MORE -> , identifier READ_MORE
              {
                item = search(identifier.getName())
                nxq = nxq + 1;
                gen(nxq, OPR, 0, 16);
                nxq = nxq + 1;
                gen(nxq, STO, getNowLevel() - item.getLevel(),
item.getAddress());
              }
 READ_MORE -> ^
 WRITE_STATEMENT -> WRITE_FIRST WRITE_MORE )
 WRITE_FIRST -> write ( EXPRESSION
              {
                nxq = nxq + 1;
                gen(nxq, OPR, 0, 14)
                nxq = nxq + 1;
                gen(nxq, OPR, 0, 15)
              }
 WRITE_MORE -> , EXPRESSION WRITE_MORE_M WRITE_MORE
 WRITE_MORE_M -> ^
              {
                nxq = nxq + 1;
                gen(nxq, OPR, 0, 14)
                nxq = nxq + 1;
                gen(nxq, OPR, 0, 15)
              }
 WRITE_MORE -> ^
 //...
```

4.4 解释程序（虚拟机程序）

4.4.1 Instruction类

```
//instruction.h
enum class InstructionType
{
  LIT = 0,
  LOD = 1,
```

```
  STO = 2,
  CAL = 3,
  INT = 4,
  JMP = 5,
  JPC = 6,
  OPR = 7,
};

class Instruction
{
public:
  Instruction();
  Instruction(InstructionType type, int levelDiff, int operand);

  InstructionType getType() const;
  void setType(InstructionType newType);

  int getLevelDiff() const;
  void setLevelDiff(int newLevelDiff);

  int getOperand() const;
  void setOperand(int newOperand);

  void print();

private:
  InstructionType type;
  int levelDiff;
  int operand;
};

typedef std::vector<Instruction> InstructionList;
```

InstructionType枚举类定义了指令类型，包括LIT、LOD、STO、CAL、INT、JMP、JPC、OPR八种。Intruction类包含了三个私有成员：类型type，层次差levelDiff，操作数operand。还包括了一些列获取和修改这些私有成员的函数。使用typdef定义了InstructionList类型，它其实是vector<Instruction>类型的别名。

4.4.2 VirtualMachine类

```
//virtualmachine.h
//...
class VirtualMachine
{
public:
  VirtualMachine();

  void restart(const std::list<int> &_screenInputBuffer);
  void runStep(int step);
  void runUntilStop();

  void sendScreenInputBuffer(const std::list<int>& _screenInputBuffer);
  void sendInstructionList(const InstructionList* _instructionList);

  void printInfor();
  void debugRun();
```

```
private:
  std::vector<int> stack;
  Instruction I;
  int P;
  int T;
  int B;
  QString screenOutputBuffer;
  std::list<int> screenInputBuffer;

  const int stackBottom = -100000;

  const InstructionList* instructionList;

  int runOneStep();

  void getInstruction();
  void runInstruction();
  void addP();

  int getStackTop();


  void LIT(int levelDiff, int value);
  void LOD(int levelDiff, int value);
  void STO(int levelDiff, int value);
  void CAL(int levelDiff, int value);
  void INT(int levelDiff, int value);
  void JMP(int levelDiff, int value);
  void JPC(int levelDiff, int value);
  void OPR(int levelDiff, int value);
};
//...
```

VirtuaMachine定义了一个执行目标代码的虚拟机。四个私有成员I、P、T、B分别模拟了指令寄存器，程序计数器，栈顶寄存器和基地址寄存器，vector<int>类型的stack模拟了运行栈，字符串类型的screenOutputBuffer模拟了输出，InstructionList*类型的instructionList记录了语法语义分析生成的代码序列。restart()方法用于重置虚拟机信息，重新开始程序运行。部分方法通过方法名可知其作用，实现相对简单，不再赘述。部分核心方法会在后面详细介绍。

```
//virtualmachine.cpp
//...
void VirtualMachine::LIT(int levelDiff, int value)
{
  this->stack.push_back(value);
}

void VirtualMachine::LOD(int levelDiff, int value)
{
  int now = this->B;
  while(levelDiff)
  {
    now = stack[now];
    -- levelDiff;
  }
  stack.push_back(stack[now + value]);
}
```

```
void VirtualMachine::STO(int levelDiff, int value)
{
  int now = this->B;
  while(levelDiff)
  {
    now = this->stack[now];
    -- levelDiff;
  }
  this->stack[now+value] = getStackTop();
  this->stack.pop_back();
}

void VirtualMachine::CAL(int levelDiff, int value)
{
  int SL = B;
  while(levelDiff)
  {
    SL = stack[SL];
    -- levelDiff;
  }
  stack.push_back(SL);
  stack.push_back(B);
  stack.push_back(P);
  B = T + 1;
  P = value;
}

void VirtualMachine::INT(int levelDiff, int value)
{
  this->T += value;
  while(this->stack.size() < this->T +1)
    this->stack.push_back(0);
}

void VirtualMachine::JMP(int levelDiff, int value)
{
  this->P = value;
}

void VirtualMachine::JPC(int levelDiff, int value)
{
  if(!getStackTop())
  {
    this->P = value;
  }
  stack.pop_back();
}

void VirtualMachine::OPR(int levelDiff, int value)
{
  if(value == 0)
  {
    T = B - 1;
    P = stack[T + 3];
    B = stack[T + 2];
    while(stack.size() > T + 1) stack.pop_back();
  }
  else if(value == 1)
```

```
{
  stack[stack.size() - 1] = -1 * getStackTop();
}
else if(value >= 2 && value <= 5)
{
  int l1 = getStackTop();
  stack.pop_back();
  int l2 = getStackTop();
  stack.pop_back();
  int ans = 0;

  if(value == 2) ans = (l2 + l1);
  else if(value == 3) ans = (l2 - l1);
  else if(value == 4) ans = (l2 * l1);
  else if(value == 5) ans = (l2 / l1);

  stack.push_back(ans);
}
else if(value == 6)
{
  int l1 = getStackTop();
  stack.pop_back();
  if(l1 & 1)
    stack.push_back(1);
  else
    stack.push_back(0);
}
else if(value == 7)
{

}
else if(value >= 8 && value <= 13)
{
  int l1 = getStackTop();
  stack.pop_back();
  int l2 = getStackTop();
  stack.pop_back();
  int ans = 0;
  if(value == 8) ans = (l2 == l1);
  else if(value == 9) ans = (l2 != l1);
  else if(value == 10) ans = (l2 < l1);
  else if(value == 11) ans = (l2 >= l1);
  else if(value == 12) ans = (l2 > l1);
  else if(value == 13) ans = (l2 <= l1);

  stack.push_back(ans);
}
else if(value == 14)
{
  this->screenOutputBuffer += QString::number(getStackTop());
  stack.pop_back();
}
else if(value == 15)
{
  this->screenOutputBuffer += "\n";
}
else if(value == 16)
{
```

```
    stack.push_back(screenInputBuffer.front());
    screenInputBuffer.pop_front();
  }
}
//...
```

上述代码描述了各个指令的运行过程。我们只介绍几个复杂指令的执行。如前所述，基地址寄存器B记录了当前过程运行栈栈底，指向静态链。LOD函数和STO函数通过层次差，经由静态链跳转到对应层次，根据偏移量获取或修改变量。CAL指令根据层次差找到调用过程所在层次，将静态链指向调用层次的栈底基地址用于变量访存。动态链指向当前栈。依次压入静态链、动态链和返回地址。之后将基地址寄存器设置为当前栈顶寄存器指向的地址加一，程序计数器跳转至制定过程的第一条指令。在PL/0生成的目标代码中，每一个过程的第一条语句都是INT，它负责开辟数据区，INT开辟内存单元的主要作用就是将栈顶寄存器指向它该指向的位置。在这之前，新过程运行栈已经提前压入了3个大小的存储信息，在加上该过程变量个数，就是我们在翻译过程中确定的INT参数大小，INT据此开辟足够大的存储区存储，同时将栈顶寄存器指向应有位置。相应的，OPR 0 0指令充当了返回的作用。它将栈顶寄存器指向基地址寄存器所存地址减一的位置，这是上一个过程的栈顶所在位置，同时把程序计数器指向返回地址，基地址寄存器指向动态链所指地址，完成返回。

注意，在翻译过程中，我们合理使用指令保证了每一个正确语义语义执行结束后，运行栈只包含静态链、动态链、返回地址和变量信息，不包含其它指令过程中运行的信息。因此上述操作不会受到其它复杂数据的干扰。

```
// virtualmachine.cpp
// ...
int VirtualMachine::runOneStep()
{
  if(P < 0) return -1;
  this->getInstruction();
  this->addP();
  this->runInstruction();

  return 1;
}
void VirtualMachine::getInstruction()
{
  this->I = (*(this->instructionList))[P];
}
void VirtualMachine::runInstruction()
{
  int levelDiff = I.getLevelDiff();
  int value = I.getOperand();
  auto type = I.getType();
  if(type == InstructionType::LIT)
  {
    LIT(levelDiff, value);
  }
  else if(type == InstructionType::CAL)
  {
    CAL(levelDiff, value);
  }
  else if(type == InstructionType::INT)
  {
    INT(levelDiff, value);
  }
  else if(type == InstructionType::JMP)
  {
```

```
    JMP(levelDiff, value);
  }
  else if(type == InstructionType::JPC)
  {
    JPC(levelDiff, value);
  }
  else if(type == InstructionType::LOD)
  {
    LOD(levelDiff, value);
  }
  else if(type == InstructionType::OPR)
  {
    OPR(levelDiff, value);
  }
  else if(type == InstructionType::STO)
  {
    STO(levelDiff, value);
  }

}

void VirtualMachine::addP()
{
  ++ this->P;
}
void VirtualMachine::runStep(int step)
{
  for(int i = 0;i < step;++ i)
  {
    if(runOneStep() < 0) break;
  }
}

void VirtualMachine::runUntilStop()
{
  while(runOneStep() >= 0);
}
// ...
```

runOneStep()方法执行了一个完整的指令，包括取值、程序计数器+1和运行指令三个操作。运行结束回返回-1。我们还提供了runStep(int step)方法用于一次运行多步，以及runUntilStop()方法用于运行到程序结束。

5. 实验结果展示

5.1 测试代码

我们使用一个复杂的PL/0程序进行展示。

```
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
```

该程序读入7个数，计算第一、二个数的成绩并输出，计算第三个数除以第四个数并输出商和余数，计算第五个数和第六个数的最小公约数并输出，计算第七个书的阶乘并输出。

5.2 词法分析结果

结果过多我们不全部列出，仅列出一部分：

```
[ "var" , line: 1 , type: "var" ]
[ "x" , line: 1 , type: "ident" , id: "x" ]
[ "," , line: 1 , type: "," ]
[ "y" , line: 1 , type: "ident" , id: "y" ]
[ "," , line: 1 , type: "," ]
[ "z" , line: 1 , type: "ident" , id: "z" ]
[ "," , line: 1 , type: "," ]
[ "q" , line: 1 , type: "ident" , id: "q" ]
[ "," , line: 1 , type: "," ]
[ "r" , line: 1 , type: "ident" , id: "r" ]
[ "," , line: 1 , type: "," ]
[ "n" , line: 1 , type: "ident" , id: "n" ]
[ "," , line: 1 , type: "," ]
[ "f" , line: 1 , type: "ident" , id: "f" ]
[ ";" , line: 1 , type: ";" ]
[ "procedure" , line: 3 , type: "procedure" ]
[ "multiply" , line: 3 , type: "ident" , id: "multiply" ]
[ ";" , line: 3 , type: ";" ]
[ "var" , line: 4 , type: "var" ]
[ "a" , line: 4 , type: "ident" , id: "a" ]
[ "," , line: 4 , type: "," ]
[ "b" , line: 4 , type: "ident" , id: "b" ]
[ ";" , line: 4 , type: ";" ]
[ "begin" , line: 5 , type: "begin" ]
[ "a" , line: 6 , type: "ident" , id: "a" ]
[ ":=" , line: 6 , type: ":=" ]
[ "x" , line: 6 , type: "ident" , id: "x" ]
[ ";" , line: 6 , type: ";" ]
[ "b" , line: 7 , type: "ident" , id: "b" ]
[ ":=" , line: 7 , type: ":=" ]
[ "y" , line: 7 , type: "ident" , id: "y" ]
[ ";" , line: 7 , type: ";" ]
[ "z" , line: 8 , type: "ident" , id: "z" ]
[ ":=" , line: 8 , type: ":=" ]
[ "x" , line: 8 , type: "ident" , id: "x" ]
[ "", line: 8 , type: "" ]
[ "y" , line: 8 , type: "ident" , id: "y" ]
[ "end" , line: 9 , type: "end" ]
[ ";" , line: 9 , type: ";" ]
[ "procedure" , line: 11 , type: "procedure" ]
[ "divide" , line: 11 , type: "ident" , id: "divide" ]
[ ";" , line: 11 , type: ";" ]
[ "var" , line: 12 , type: "var" ]
[ "w" , line: 12 , type: "ident" , id: "w" ]
[ ";" , line: 12 , type: ";" ]
[ "begin" , line: 13 , type: "begin" ]
[ "r" , line: 14 , type: "ident" , id: "r" ]
[ ":=" , line: 14 , type: ":=" ]
```

```
[ "x" , line: 14 , type: "ident" , id: "x" ]
[ ";" , line: 14 , type: ";" ]
[ "q" , line: 15 , type: "ident" , id: "q" ]
[ ":=" , line: 15 , type: ":=" ]
[ "0" , line: 15 , type: "number" , value: 0 ]
[ ";" , line: 15 , type: ";" ]
[ "w" , line: 16 , type: "ident" , id: "w" ]
[ ":=" , line: 16 , type: ":=" ]
[ "y" , line: 16 , type: "ident" , id: "y" ]
[ ";" , line: 16 , type: ";" ]
[ "while" , line: 17 , type: "while" ]
[ "w" , line: 17 , type: "ident" , id: "w" ]
[ "<=" , line: 17 , type: "<=" ]
[ "r" , line: 17 , type: "ident" , id: "r" ]
[ "do" , line: 17 , type: "do" ]
[ "w" , line: 17 , type: "ident" , id: "w" ]
[ ":=" , line: 17 , type: ":=" ]
[ "2" , line: 17 , type: "number" , value: 2 ]
[ "", line: 17 , type: "" ]
[ "w" , line: 17 , type: "ident" , id: "w" ]
[ ";" , line: 17 , type: ";" ]
[ "while" , line: 18 , type: "while" ]
[ "w" , line: 18 , type: "ident" , id: "w" ]
[ ">" , line: 18 , type: ">" ]
[ "y" , line: 18 , type: "ident" , id: "y" ]
[ "do" , line: 18 , type: "do" ]
[ "begin" , line: 19 , type: "begin" ]
[ "q" , line: 20 , type: "ident" , id: "q" ]
[ ":=" , line: 20 , type: ":=" ]
[ "2" , line: 20 , type: "number" , value: 2 ]
[ "", line: 20 , type: "" ]
[ "q" , line: 20 , type: "ident" , id: "q" ]
[ ";" , line: 20 , type: ";" ]
...
```

5.23 语法语义分析结果

我们给出最终生成的目标代码。

```
0 : JMP  0 97
1 : INT  0 5
2 : LOD  1 9
3 : STO  0 4
4 : LOD  1 8
5 : STO  0 3
6 : LOD  1 9
7 : LOD  1 8
8 : OPR  0 4
9 : STO  1 7
10 : OPR  0 0
11 : INT  0 4
12 : LOD  1 9
13 : STO  1 5
```

```
14 :  LIT  0 0
15 :  STO  1 6
16 :  LOD  1 8
17 :  STO  0 3
18 :  LOD  0 3
19 :  LOD  1 5
20 :  OPR  0 13
21 :  JPC  0 27
22 :  LIT  0 2
23 :  LOD  0 3
24 :  OPR  0 4
25 :  STO  0 3
26 :  JMP  0 18
27 :  LOD  0 3
28 :  LOD  1 8
29 :  OPR  0 12
30 :  JPC  0 52
31 :  LIT  0 2
32 :  LOD  1 6
33 :  OPR  0 4
34 :  STO  1 6
35 :  LOD  0 3
36 :  LIT  0 2
37 :  OPR  0 5
38 :  STO  0 3
39 :  LOD  0 3
40 :  LOD  1 5
41 :  OPR  0 13
42 :  JPC  0 51
43 :  LOD  1 5
44 :  LOD  0 3
45 :  OPR  0 3
46 :  STO  1 5
47 :  LOD  1 6
48 :  LIT  0 1
49 :  OPR  0 2
50 :  STO  1 6
51 :  JMP  0 27
52 :  OPR  0 0
53 :  INT  0 4
54 :  LOD  1 9
55 :  STO  1 3
56 :  LOD  1 8
57 :  STO  0 3
58 :  LOD  1 3
59 :  LOD  0 3
60 :  OPR  0 9
61 :  JPC  0 79
62 :  LOD  1 3
63 :  LOD  0 3
64 :  OPR  0 10
65 :  JPC  0 70
```

```
66 :  LOD  0 3
67 :  LOD  1 3
68 :  OPR  0 3
69 :  STO  0 3
70 :  LOD  0 3
71 :  LOD  1 3
72 :  OPR  0 10
73 :  JPC  0 78
74 :  LOD  1 3
75 :  LOD  0 3
76 :  OPR  0 3
77 :  STO  1 3
78 :  JMP  0 58
79 :  LOD  1 3
80 :  STO  1 7
81 :  OPR  0 0
82 :  INT  0 3
83 :  LOD  1 4
84 :  LIT  0 1
85 :  OPR  0 12
86 :  JPC  0 96
87 :  LOD  1 4
88 :  LOD  1 3
89 :  OPR  0 4
90 :  STO  1 3
91 :  LOD  1 4
92 :  LIT  0 1
93 :  OPR  0 3
94 :  STO  1 4
95 :  CAL  1 82
96 :  OPR  0 0
97 :  INT  0 10
98 :  OPR  0 16
99 :  STO  0 9
100 :  OPR  0 16
101 :  STO  0 8
102 :  CAL  0 1
103 :  LOD  0 7
104 :  OPR  0 14
105 :  OPR  0 15
106 :  OPR  0 16
107 :  STO  0 9
108 :  OPR  0 16
109 :  STO  0 8
110 :  CAL  0 11
111 :  LOD  0 6
112 :  OPR  0 14
113 :  OPR  0 15
114 :  LOD  0 5
115 :  OPR  0 14
116 :  OPR  0 15
117 :  OPR  0 16
```

```
118 :  STO  0 9
119 :  OPR  0 16
120 :  STO  0 8
121 :  CAL  0 53
122 :  LOD  0 7
123 :  OPR  0 14
124 :  OPR  0 15
125 :  OPR  0 16
126 :  STO  0 4
127 :  LIT  0 1
128 :  STO  0 3
129 :  CAL  0 82
130 :  LOD  0 3
131 :  OPR  0 14
132 :  OPR  0 15
133 :  OPR  0 0
```

5.4 目标代码的执行

我们给出了如下输入：

```
8  19  36  9  72  48  5
输出结果如下：
"152\n4\n0\n24\n120\n"
程序运行正确
```