PL0 编译器

202000130143 郑凯饶
2023.5.16

一、 实验目的

应用编译原理所学知识，理论联系实际，设计 PL0 语言编译器。

二、 实验环境

- 硬件环境：

笔记本：Dell Latitude 5411
CPU：Intel(R) Core(TM) i5-10400H CPU @ 2.60GHz(8GPUs),~2.6GHz
内存：32.0GB

- 软件环境：

操作系统：Windows 10 家庭中文版
集成开发环境：CLion 2022.3 x64
编译工具：CMake 3.24
编译器：MinGW 11.2.0
生成器：ninja

- C++标准：C++14

三、 实验内容

1. 词法分析

(1) 任务要求
由于实验需要在 OJ 平台上提交评测,所有 PL0 编译器实现遵循 OJ 平台上的标准。

给定一个 PL0 语言源程序，将其从字符流转换为词语流。具体地，需要过滤掉源程序中的空白字符（空格、tab、换行符），识别关键字、数字以及运算符。

PL0 任意一个词法单元均不会超过 10 个字符单元，否则，认定为词法错误，输出 Lexical Error。

PL0 仅仅支持无符号整数。因此，对于-123，词法会将其分为词语-和 123。

PL0 关键词表：

CONST
VAR
PROCEDURE
BEGIN
END
ODD
IF
THEN
CALL
WHILE
DO
READ
WRITE

PL0 标识符由上下文无关文法定义：

<标识符> → <字母> {<字母>|<数字>}
<字母> → A|B|C...X|Y|Z
<数字> → 0|1|2...7|8|9

PL0 运算符表：

=
:=
+
-
*
/
#
<
<=
>
>=

PL0 分隔符：

;
,
.
(
)

(2) 解决方案

首先对符号对象进行抽象，符号通过名字和 id 唯一标识，常量和变量拥有数值

number，标识符存储于 value，由于 PL0 支持嵌套定义，为了进行非局部量的访问设置 level，最后变量以及程序标识符需要 address 用于变量访问以及后面的 CALL 调用。

```
/*
 * 符号类
 */
class Symbol
{
    std::string name = "";
    int id = -1;
    std::string value = "";
    int number = -1;
    int level = -1;
    int address = -1;
};
```

Symbol 类的初始化：
通过 explicit 关键字对方法的使用范围进行限定，只能用于初始化而不能用于隐式类型转换。预防隐式转换带来的难以觉察的错误。

```
explicit Symbol(std::string _name, int _id): name(std::move(_name)), id(_id) {}
explicit Symbol(std::string _name, int _id, int _number): name(std::move(_name)),
id(_id), number(_number) {}            // 常量声明
explicit Symbol(std::string _name, int _id, std::string _value):
name(std::move(_name)), id(_id), value(_value) {}      // 变量声明，value 存储变量名
```

之后通过哈希表定义所有的符号对象，将输入字符串和符号进行绑定，用于快速查找，并将哈希表放入命名空间 SYMBOL 中以防命名空间污染。

```
const std::unordered_map<std::string, Symbol> KEYWORD_MAP({
    /* 关键字 */
    {"begin", Symbol("begin",1)},
    {"end", Symbol("end", 2)},
    {"if", Symbol("if", 3)},
    {"then", Symbol("then", 4)},
    {"else", Symbol("else", 5)},
    {"const", Symbol("const", 6)},
    {"procedure", Symbol("procedure", 7)},
    {"var", Symbol("var", 8)},
    {"do", Symbol("do", 9)},
    {"while", Symbol("while", 10)},
    {"call", Symbol("call", 11)},
```

```
    {"read", Symbol("read", 12)},
    {"write", Symbol("write", 13)},
    {"odd", Symbol("odd", 14)},
});
```

词法分析器的设计：
主要是对输入流 inputStream 的处理,get 从 inputStream 中获取一个词语,peek 查看但是不获取 inputStream 的一个词语,isEOF 方法判断输入流是否处理结束。

```
/**
 * 词法分析器
 */
class Lexer
{
public:
    Lexer(std::istream &_inputStream) : inputStream(_inputStream)
    {}
    Symbol getSymbol();
    bool isEOF();

private:
    std::istream &inputStream;
    char get();
    char peek();
};
```

核心方法为 getSymbol，方法的工作流程：

a. isEOF 判断是否到达文件末；
b. 滤掉单词间的空格、Tab 以及换行符；
c. 再一次 isEOF 判断是否到达文件末；
d. Peek 查看首字符是否是字母，若是则可能是标识符或者关键字类型，get 获取往后的所有数字或者字符，拼接得到字符串，在关键字表中查找，若找到返回对应的 Symbol，否则为标识符；
e. Peek 查看首字符是否是数字，如是为常量类型，get 获取后面的所有数字，拼接得到一个数；
f. 判断是否是算符或者界符，在哈希表中查找并返回，否则为非法符号！

2. 语法分析

(1) 任务要求
前面的词法分析器提供词语流，以合适的方式对词语流进行分析，并生成一个语法树，或者宣告语法错误。

PL0 语言的语法可以通过上下文无关文法进行描述：

<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>|<空语句>
<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>)
<加减运算符> → +|-
<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(<标识符>{,<标识符>})
<字母> → A|B|C…X|Y|Z
<数字> → 0|1|2…7|8|9
<空语句> → epsilon

PL0 语言允许嵌套定义函数，但嵌套不允许超过 3 层，否则视为语法错误。

请忽略语义错误，诸如重复声明的变量，使用未声明的变量等。在语法分析阶段，这些程序暂时还是正确的。

生成语法树，输出树的括号表示形式，节点名称有如下约定：

上下文无关文法中的节点	对应的替换词语
程序	PROGRAM
分程序	SUBPROG
常量说明部分	CONSTANTDECLARE

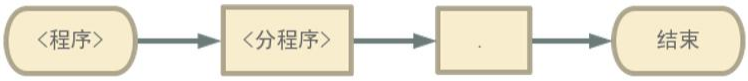
上下文无关文法中的节点	对应的替换词语
常量定义	CONSTANTDEFINE
无符号整数	< 这是一个叶子节点，用其本身替代 >
变量说明部分	VARIABLEDECLARE
标识符	< 这是一个叶子节点，用其本身替代 >
过程说明部分	PROCEDUREDECLARE
过程首部	PROCEDUREHEAD
语句	SENTENCE
赋值语句	ASSIGNMENT
复合语句	COMBINED
条件	CONDITION
表达式	EXPRESSION
项	ITEM
因子	FACTOR
加减运算符	< 这是一个叶子节点，用其本身替代 >
乘除运算符	< 这是一个叶子节点，用其本身替代 >
关系运算符	< 这是一个叶子节点，用其本身替代 >
条件语句	IFSENTENCE
过程调用语句	CALLSENTENCE
当型循环语句	WHILESENTENCE
读语句	READSENTENCE
写语句	WRITESENTENCE
空语句	EMPTY

(2) 解决方案

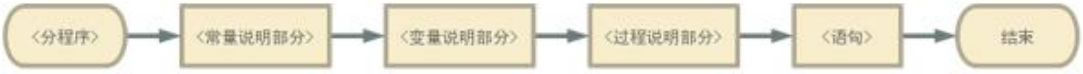
A. 递归下降地进行语法分析

采用递归下降子程序法进行语法分析，每个产生式设计一个函数解析。对应的程序框图如下：

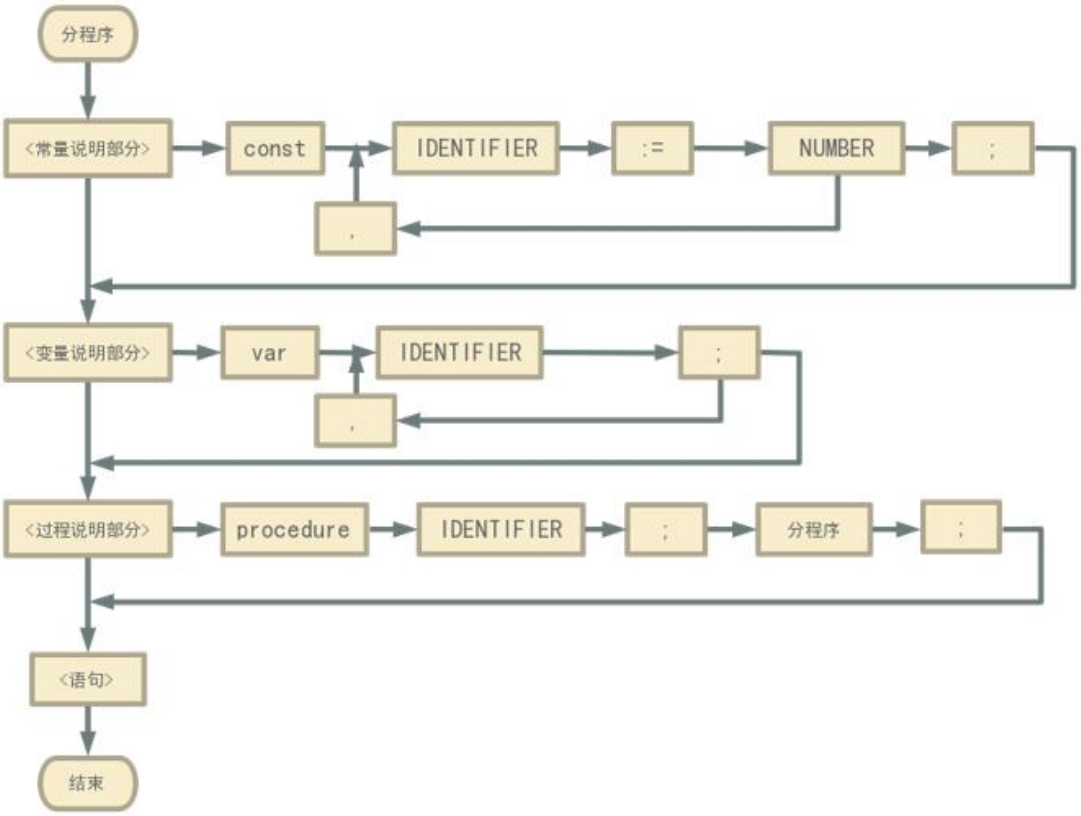
a. <程序>→<分程序>.



b. <分程序>→ [<常量说明部分>][<变量说明部分>][<过程说明部分>]<语句>



c. 以此类推画出所有产生式对应的程序框图，进一步，可以将多个程序框图合并到一起方便查看程序的全貌：



将取词逻辑封装成一个方法：

```
// 取下一个 词
void Parser::advance() {
    if ("illegal" != nowSymbol.name)
        // 通过唯一标识符号 name 进行比较匹配
    {   // nowSymbol 初始化为 illegal
        if (!nowSymbol.value.empty())
            syntaxTree.merge(nowSymbol.value, 1);
        else if (-1 != nowSymbol.number)
            syntaxTree.merge(std::to_string(nowSymbol.number), 1);
        else {
            if (nowSymbol.id >= 15 && nowSymbol.id <= 24) {
```

```
                // 加减，乘除，关系运算符是叶子节点
                syntaxTree.merge(nowSymbol.name, 1);
            } else { syntaxTree.merge(nowSymbol.name); }
        }
    }
    // 原先在 main 函数中的取词逻辑
    nowSymbol = lexer.getSymbol();  // getSymbol 函数中变量名过长也会导致词法错误
    // 发生词法错误之后不需要进行接下来的分析
    if (nowSymbol.name == "illegal") LexicalError = true;
    if (nowSymbol.name == "$") {}
    if (LexicalError) {
        exit(1);
    }
    logs("LOG[Parser: advance] ", nowSymbol);
}
```

其中 log 是我设计的调试类（在附录中说明）。syntaxTree 是语法树，如果当前符号是标识符则将 value 合并进入语法树，如果是常量将 number 合并，否则将 name 合并。

一个函数处理一个产生式。
例如递归解析产生式：〈当型循环语句〉 → while〈条件〉do〈语句〉，首先跳过 while，递归解析<条件>，之后，下一个词必须是 do，通过 ASSERT 进行断言，失败输出语法错误，并报告具体的异常信息 EXCEPTION（在附录中说明），之后递归解析<语句>。

```
// 〈当型循环语句〉 → while〈条件〉do〈语句〉
bool Parser::whileStatement() {
    logs("LOG[Parser: whileStatement]");
    if ("while" == nowSymbol.name) {
        syntaxTree.begin("WHILESENTENCE");

        /* 语义分析 */
        advance();                          // 跳过 'while'

        /* 中间代码生成 */
        int whileBeginAddress = getCodeAddress();

        /* 语义分析 */
        condition();
        ASSERT("do" == nowSymbol.name, EXCEPTION::MISSING_DO); // 缺少 DO
        advance(); // 跳过 'do'
```

```
        /* 中间代码生成 */
        Code *jpc = new Code(CODE::JPC);// 条件转移代码，地址回填
        codeTable.push_back(jpc);

        /* 语义分析 */
        statement();
        // 执行完<语句>，无条件跳转回条件判断处
        codeTable.push_back(new Code(CODE::JMP, whileBeginAddress)); // 无条件跳
转
        // <条件>不满足，跳转至<语句>之后
        jpc->setA(getCodeAddress());

        syntaxTree.end();
        return true;
    }
    return false;
}
```

B. 语法树的生成：

需要在递归下降分析的过程中构建语法树，在上面的例子中我使用了 syntaxTree.begin("WHILESENTENCE")以及 syntaxTree.end()两个方法。

a. 语法树的节点（语法基本单元）：

```
class SyntaxTreeNode
{
public:
    std::string name;
    bool isLeaf;
    std::list<SyntaxTreeNode*> children;
    explicit SyntaxTreeNode(std::string name, bool isLeaf = 0):
name(std::move(name)), isLeaf(isLeaf) {}
};
```

节点的名称 name，是否是叶子节点 isLeaf，节点的孩子节点 childrens，通过 list 容器存储。同样通过 explicit 修饰构造函数，传入字符串参数使用移动语义，原先的字符串不再使用。

b. 语法树：

```
class SyntaxTree
{
    std::list<SyntaxTreeNode*> syntaxTreeStack;  // 语法树 栈
    SyntaxTreeNode* root;                        // 语法树 根
public:
    void merge(Symbol* symbol);
```

```
    void merge(std::string name,  bool isLeaf = 0); // 增加 isLeaf 标识叶子节点
    void begin(const std::string &name);
    void end();
    void print();
    void print(SyntaxTreeNode* now, const std::string& prefix, bool isLast);
};
```

syntaxTreeStack 是一个栈，存储当前活跃的节点(还未完成解析的节点)，begin 以及 end 方法对这个栈进行操作。

```
void SyntaxTree::begin(const std::string &name)
{   // 往语法树中添加一个语法单元
    SyntaxTreeNode *syntaxTreeNode = new SyntaxTreeNode(name);
    if (syntaxTreeStack.empty())
        this->root = syntaxTreeNode;    // 若整个语法树为空，则设置当前语法单元为根
    else    // 否则将该节点作为当前栈顶的儿子节点加入
        syntaxTreeStack.back()->children.push_back(syntaxTreeNode);
    syntaxTreeStack.push_back(syntaxTreeNode);
}

void SyntaxTree::end()
{
    if (syntaxTreeStack.back()->children.empty())
    {   // 若该节点没有儿子，则将该节点弹出，打印时将不再显示
        syntaxTreeStack.pop_back();
        syntaxTreeStack.back()->children.pop_back();
    } else
    {
        syntaxTreeStack.pop_back();
    }
}
```

Begin 和 end 方法生成了语法树的中间单元，merge 操作则将来自词语流的一个个词语作为叶子节点添加到中间单元的 children 中，完成对词语的语法分析。

```
void SyntaxTree::merge(std::string name, bool isLeaf)
{
    syntaxTreeStack.back()->children.push_back(new
SyntaxTreeNode(std::move(name), isLeaf));
}
```

之后深度优先遍历语法树，输出语法树的括号表示。

3. 语义分析与目标代码产生

(1) 任务要求
至此已经完成了词法分析器和语法分析器，词法分析器会向语法分析器提供词语流，而后者则按规范推导/归约生成一棵语法树。在本实验中。将用语法制导翻译的方式，完成给定语法的语义分析以及目标代码生成。

对于每个 PL0 语言过程的说明部分制作符号表，填写所在层次、属性，并分配相对地址。例如对于程序

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

应当生成如下符号表

NAME	KIND	PARAMETER1	PARAMETER2
a	CONSTANT	VAL:35	--
b	CONSTANT	VAL:49	--
c	VARIABLE	LEVEL: LEV	ADR: DX
d	VARIABLE	LEVEL: LEV	ADR: DX+1
e	VARIABLE	LEVEL: LEV	ADR: DX+2
p	PROCEDURE	LEVEL: LEV	ADR: <UNKNOWN>
g	VARIABLE	LEVEL: LEV+1	ADR: DX

其中 LEVEL 给出的是层次，DX 是每一层局部量的相对地址。

生成的目标代码是一种假想栈式的计算机的汇编语言，其格式为 f l a
其中 f 为功能码，l 代表层次差，a 代表位移量。
这种假想栈式计算机有一个无限大的栈，以及四个寄存器 IR,IP,SP,BP。
IR，指令寄存器，存放正在执行的指令。
IP，指令地址寄存器，存放下一条指令的地址。
SP，栈顶寄存器，指向运行栈的顶端。
BP，基址寄存器，指向当前过程调用所分配的空间在栈中的起始地址。
共有 8 种目标码
LIT：l 域无效，将 a 放到栈顶
LOD：将于当前层层差为 l 的层，变量相对位置为 a 的变量复制到栈顶
STO：将栈顶内容复制到于当前层层差为 l 的层，变量相对位置为 a 的变量
CAL：调用过程。l 标明层差，a 表明目标程序地址
INT：l 域无效，在栈顶分配 a 个空间
JMP：l 域无效，无条件跳转到地址 a 执行

JPC：l 域无效，若栈顶对应的布尔值为假（即 0）则跳转到地址 a 处执行，否则顺序执行
OPR：l 域无效，对栈顶和栈次顶执行运算，结果存放在次顶，a=0 时为调用返回
你可以自行定义 OPR 中的运算和 a 的对应关系。

PL0 的语义细节，在解决方案中结合实现详细说明。

(2) 解决方案
A. 符号表
采用语法制导翻译，在进行规范推导的过程进行目标代码生成。编译过程中符号表是一个非常重要的中间媒介，用于记录标识符的信息。同时符号表由于过程可以相互嵌套，符号表之间也存在层次关系。

定义符号表类：

```
class SymbolTable
{   // 符号表
public:
    std::unordered_map<std::string, Symbol*> mp;
    std::list<Symbol*> lst;
    bool inTable(const std::string& name);  // 判断符号是否在符号表中出现
    void addSymbol(Symbol* symbol);         // 添加符号项
};
```

符号表的主要内容通过一个 list 存储，为了加速查找过程设置一个哈希表。方法 inTable 通过访问哈希表，判断对应符号是否在符号表中出现。addSymbol 方法向符号表中添加符号。

由于程序执行过程中，需要构建多个过程，因此会出现多张符号表，为了实现对这些符号表的统一管理，设计一个符号表管理类 SymbolTableManager，和语法树类似，设置一个栈，用于存储当前活跃的符号表，另外通过一个结构存储全局所有的符号表，方便后面输出调试。

```
class SymbolTableManager
{
    std::list<SymbolTable*> symbolTableStack;  // 符号表 栈
    std::list<SymbolTable*> symbolTableList;   // 符号表 数组
public:
    SymbolTableManager()
    {
        pushTable();
    }

    void addSymbol(const std::string &symbolName, int id, const std::string &name,
```

```
int number, int level, int address);
    Symbol *getLastProcedure();
    bool inTable(const std::string& name);
    Symbol *getSymbol(const std::string &name);
    void pushTable();
    void popTable();
    void printTables();
};
```

结合代码介绍几个方法：

a. getLastProcedure 方法

过程体的目标代码生成之后，需要返填过程体的入口地址。这个同样在会在符号表中记录，这样之后分析的过程通过 CAL 调用该过程的话，访问符号表获取过程的入口地址，赋值给 CAL 语句的 a 域。因此在每个分程序中我们需要知道当前活跃的是哪个过程，通过反序遍历栈（正是为了这一步，在实现时使用 list 容器而不是 stack）获取最近的过程标识符。

```
Symbol *SymbolTableManager::getLastProcedure()
{   // 获取最近出现的“程序”关键字
    for (auto it = symbolTableStack.rbegin(); it != symbolTableStack.rend(); it++)
{    // fixed
        for (auto it2 =  (*it)->lst.rbegin(); it2 != (*it)->lst.rend(); it2++) {
            if ("procedure" == (*it2)->name) {
                return *it2;
            }
        }
    }
    return nullptr;
}
```

有一点要注意，在 CAL 调用时过程体的入口并不一定已经确定，CAL 可以调用祖先过程（包含当前过程的外层过程），这是合法的！如下图程序

```
1   const k=10;
2   var a, b;
3   procedure f1;
4       var b,c;
5       procedure f2;
6           var c,d;
7           begin
8           write(a, a);
9           if a < k then call f1;
10          end;
11      begin
12      a := a + 1;
13      write(a);
14      call f2;
15      end;
16  begin
17  a := a + 1;
18  write(a);
19  call f1;
20  end.
```

因此，我们需要利用目标代码的 a 域构建一个链表，并在入口地址确定之后回填该链表。

```
    if (procedure != nullptr)                                    // 不是主函数，那
该过程的起始语句的地址要保存在符号表
    {
        // backpatch 操作
        int t = procedure->address, nxt;
        while (t != -1) {
            nxt = codeTable[t]->a;
            codeTable[t]->a = getCodeAddress() - 1;
            t = nxt;
        }
        procedure->address = getCodeAddress() - 1;  // fixed
    }   // 起始语句即上一句 INT 指令
```

b. inTable 方法

调用当前符号表的 inTable 方法进行查找。这个主要用于防止同一个过程中局部变量的重复声明。

c. getSymbol 方法

在所有活跃符号表中查找符号，这个符号可以是非局部的。同样为了保证访问的是最内层的变量，采取反序遍历的方式。

```
Symbol *SymbolTableManager::getSymbol(const std::string &name)
{   // lookup，在<有效>符号表中查找某一个变量
    // 保证：访问的变量是递归过程中最接近的变量 fixed
    for (auto it = symbolTableStack.rbegin(); it != symbolTableStack.rend(); it++)
{
        auto res = (*it)->mp.find(name);
        if (res != (*it)->mp.end()) {
            return res->second;
        }
    }
    return nullptr;
}
```

B. 目标代码

目标代码一共有 8 类，为之设计枚举类，枚举类作为强类型对编码更加友好。（前面我直接使用字符串表达词语，在编码过程中犯了很多细节错误！）

```
namespace CODE
{
    enum CODE_TYPE
    {
        LIT, // 将常数放到运栈顶，a 域为常数
        LOD, // 将变量放到栈顶。a 域为变量在所说明层中的相对位置，l 为调用层与说
```

```
明层的层差值
        STO, // 将栈顶的内容送到某变量单元中。a 域为变量在所说明层中的相对位置，
l 为调用层与说明层的层差值
        CAL, // 调用过程的指令。a 为被调用过程的目标程序的入口地址，l 为层差
        INT, // 为被调用的过程（或主程序）在运行栈中开辟数据区。a 域为开辟的个数
        JMP, // 无条件转移指令，a 为转向地址
        JPC, // 条件转移指令，当栈顶的布尔值为非真时，转向 a 域的地址，否则顺序
执行
        OPR  // 关系和算术运算。具体操作由 a 域给出。运算对象为栈顶和次顶的内容
进行运算，结果存放在次顶。a 域为 0 时是退出数据区
    };
```

设计 OPR 指令：

```
enum OP_TYPE
{
    RET = 0,    // 过程调用结束后，返回调用点并退栈
    NEG = 1,    // 栈顶元素取反
    ADD = 2,    // 弹出次栈顶与栈顶相加，结果进栈
    SUB = 3,    // 弹出次栈顶减去栈顶，结果进栈
    MUL = 4,    // 弹出次栈顶、栈顶，相乘结果进栈
    DIV = 5,    // 弹出次栈顶除以栈顶，结果值进栈
    LEQ = 6,    // 栈顶两个元素弹出，判次栈顶小于等于栈顶结果进栈
    LS = 7,     // 栈顶两个元素弹出，判次栈顶小于栈顶结果进栈
    EQU = 8,    // 弹出栈顶两元素判相等结果入栈顶
    NEQ = 9,    // 弹出栈顶两元素判不等结果入栈顶
    GT = 10,    // 栈顶两个元素弹出，判次栈顶大于栈顶结果进栈
    GEQ = 11,   // 栈顶两个元素弹出，判次栈顶小于等于栈顶结果进栈
    WRITE = 14,// 栈顶值输出至屏幕
    LINE = 15,  // 屏幕输出换行
    READ = 16,  // 从命令行读一个数到栈顶
    ODD = 17,   // 弹出栈顶，判断是否为奇数，结果进栈
};
```

例子（1）：

在语法分析过程中同时进行目标代码的生成，同样以当型循环语句作为例子。解析完<条件>之后，此时条件的结果被放置于栈顶，因此生成一条条件跳转语句 JPC，等待回填，因此<语句>还未解析完成，无法确定<语句>之后下一句的地址。解析完<语句>，首先生成一个无条件跳转语句，跳转至前面生成的条件跳转语句处进行判断，之后将<语句>之后下一句的地址回填至条件跳转语句。（黄色标记部分为目标代码生成）

```
// 〈当型循环语句〉 → while〈条件〉do〈语句〉
bool Parser::whileStatement() {
```

```
    logs("LOG[Parser: whileStatement]");
    if ("while" == nowSymbol.name) {
        syntaxTree.begin("WHILESENTENCE");

        /* 语义分析 */
        advance();                              // 跳过 'while'

        /* 中间代码生成 */
        int whileBeginAddress = getCodeAddress();

        /* 语义分析 */
        condition();
        ASSERT("do" == nowSymbol.name, EXCEPTION::MISSING_DO); // 缺少 DO
        advance(); // 跳过 'do'

        /* 中间代码生成 */
        Code *jpc = new Code(CODE::JPC);// 条件转移代码，地址回填
        codeTable.push_back(jpc);

        /* 语义分析 */
        statement();
        // 执行完<语句>，无条件跳转回条件判断处
        codeTable.push_back(new Code(CODE::JMP, whileBeginAddress)); // 无条件跳
转
        // <条件>不满足，跳转至<语句>之后
        jpc->setA(getCodeAddress());

        syntaxTree.end();
        return true;
    }
    return false;
}
```

例子（2）：
再看一个 call 调用的解析，从符号表获取 procedure 符号，判断 level - symbol->level >= 2 是否成立，若成立则这是一个内层过程调用外层过程的 call，此时 procedure->address 还未产生，我们将此时符号表的 address 数值赋给该 call 指令的 a 域，将符号表 address 数值更新为该 call 指令的地址(增长链表)。否则，直接将 procedure->address 赋予 call 指令的 a 域。

```
// 〈过程调用语句〉 → call〈标识符〉
bool Parser::callStatement() {
    logs("LOG[Parser: callStatement]");
```

```
    if ("call" == nowSymbol.name) {
        syntaxTree.begin("CALLSENTENCE");

        /* 语义分析 */
        advance();                                  // 跳过 'CALL'
        ASSERT("identifier" == nowSymbol.name, EXCEPTION::MISSING_IDENTIFIER);//
异常处理：缺少标识符
        std::string procName = nowSymbol.value;
        // fixed
        Symbol *symbol = symbolTable.getSymbol(procName);
        ASSERT(nullptr != symbol, EXCEPTION::NOT_A_PROCUDURE);
        // CALL 调用 标识符必须是 procedure 类型
        ASSERT("procedure" == symbol->name, EXCEPTION::NOT_A_PROCUDURE); // 并非
过程
        advance();                                  // 跳过 '标识符'

        /* 中间代码生成 */
        // 如何执行 CALL 语句？参见 vm.cpp
        // 执行 CAL 主要是构建该过程的活动链、静态链以及确定返回地址
        // 调用自己的孩子则 level - symbol->level 为 0

        // 实际上该符号表已经从活动栈中弹出，不会访问到
        ASSERT(level - symbol->level >= 0, EXCEPTION::CALL_INDIRECTLY_CONTAIN);

        if (level - symbol->level >= 2) {
            int t = symbol->address;
            symbol->address = getCodeAddress(); // 指向下面将产生的 CAL 语句
            codeTable.push_back(new Code(CODE::CAL, level - symbol->level, t));
// 利用 a 区块构造链表
        }   // 如果是孩子调用祖先，此时祖先地址还未确定，等待回填
        else {
            codeTable.push_back(new Code(CODE::CAL, level - symbol->level,
symbol->address));
        }
        syntaxTree.end();
        return true;
    }
    return false;
}
```

其他产生式类似，在代码中体现。

4. 代码解释执行

(1) 任务要求

完成目标代码生成部分之后，已经可以将一个 PL0 语言程序生成为目标代码。但是，假想计算机的机器指令并不能在现代计算机上直接运行，为此需要完成一个解释器来执行对应的机器指令。

编译器从标准输入读取 PL0 源程序，并将编译后的机器码输出到标准输出。

解释器需要从 program.code 读取机器码，从标准输入读取 read 指令对应的数字，并将 write 指令对应的数字输出到标准输出。

(2) 解决方案

定义解释器 VM 如下

```
class VM
{
public:
    VM(std::vector<Code *> codeTable) : codeTable(std::move(codeTable))
    {}
    void run();

private:
    std::vector<Code*> codeTable;
    int stack[400010];  // 太大会导致栈溢出
    int top;        // SP，栈顶寄存器，指向运行栈的顶端
    int base;       // BP，基址寄存器，指向当前过程调用所分配的空间在栈中的起始地址
    int pc;         // IP，指令地址寄存器，存放下一条指令的地址
    Code* code;     // IR，指令寄存器，存放正在执行的指令
    int getBaseAddress(int nowBaseAddress, int levelDiff);
    void opr();
};
```

CodeTable 存储目标代码，对应代码空间，pc 是指令地址寄存器，指向代码空间中下一条要执行的指令；stack 是动态栈式空间，用于存储活动记录。在 PL0 语言中活动记录主要包括动态链、静态链以及返回地址，还有变量、临时变量。Top 以及 base 指向栈空间，其中 top 指向栈的顶部，base 指向当前过程活动记录的起始地址（基地址）。

重点讲讲几条比较复杂的指令：

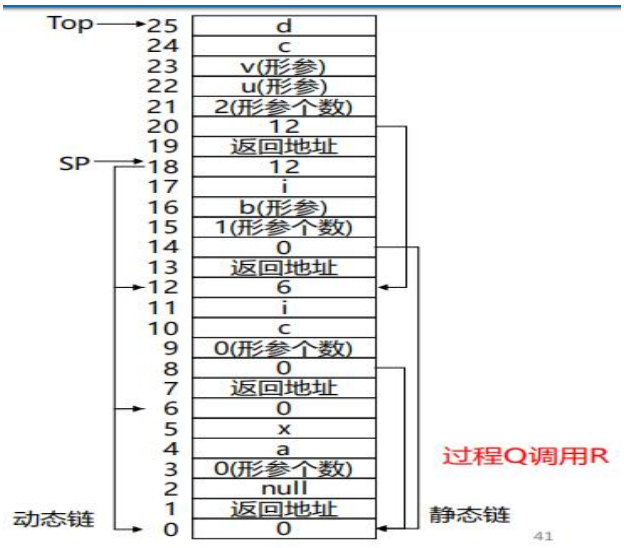
A. LOD

由于 PL0 支持嵌套定义函数（过程），因此对于一个过程涉及到非局部量（即属于外层函数的变量）的访问，这个变量只能从栈中获取（编译结束之后符号表就结束了它的使命，但在编译过程中通过它我们确定了调用者和被调用符号的层次差，以及调用符号在过程中的相对地址，这两个量被记录在访问变量 STO 以及使用变量 LOD 指令的 l 域和 a 域）。

那么如何通过 l，a 确定变量在栈中的位置？这就要讲到静态链。首先明确一点，这个变量一定在栈中。因为 PL0 的语义设定，函数只可以调用自己，和自己同层次但是先声明的函数以及被自己直接包含的函数（这个函数会在自己符号表中声明）。因此只有函数调用被自己直接包含的函数时可以引起 LEVEL 的增加，因此一个过程的直接外层过程一定先于自己被调用，其活动记录一定在栈中。

而内层过程的静态链是其直接外层过程的基地址。访问静态链得到直接外层过程的基地址，加上相对地址就可以访问到直接外层过程的局部量（自己的非局部量）。对于层次差为 l 的变量，我们顺着静态量找 l 次就可以找到被调用符号所在过程的活动记录的基地址。实现如下（静态链、动态链和返回地址的次序和 ppt 上不同但不影响，只要保证在活动记录首部即可）

```
int VM::getBaseAddress(int nowBaseAddress, int levelDiff)
{   // 获取对应层的基地址 base
    // 过程调用栈中的数据分布如下:
    // base+3:变量 a
    // base+2:返回地址
    // base+1:静态链
    // base:动态链
    while (levelDiff--)
        nowBaseAddress = stack[nowBaseAddress + 1]; // 通过静态链访问非局部量
    return nowBaseAddress;
}
```



LOD 指令的解析：

```
    // 变量送栈顶
case CODE::LOD:
    // 将于当前层层差为 l 的层，变量相对位置为 a 的变量复制到栈顶
    stack[top++] = stack[getBaseAddress(base, code->getL()) + code->getA()];
    break;
```

B. STO

和 LOD 类似，将栈顶值送往变量。

```
    // 栈顶送变量
case CODE::STO:
    // 将栈顶内容复制到于当前层层差为 l 的层，变量相对位置为 a 的变量
    stack[getBaseAddress(base, code->getL()) + code->getA()] = stack[--top];
    break;
```

C. CAL

主要是创建新过程的几个重要活动记录。

动态链：将当前基地址 base 赋值给新过程的动态链，用于过程结束之后返回。

静态链：前面讲了静态链的使用以及作用，这里进行静态链的构建。如果 l=0，则是当前函数调用自己的直接内层函数，静态链赋值为 base；如果 l=1，则是函数调用和自己同层次的函数，那么它们拥有相同的静态链，因此跳一次获取到值；如果 l=2，则是函数调用自己的直接外层函数，通过当前函数的静态链跳一次就会到达之前调用的直接外层函数（和当前调用不是一个过程，但是拥有相同的定义），再跳 1 次获取到值。以此类推。

```
    // 调用
case CODE::CAL:
    stack[top] = base;    // 动态链
    if (code->getL() <= -1) {
        exit(1);
    }
    else {
        stack[top + 1] = getBaseAddress(base, code->getL()); // 静态链
    }
    stack[top + 2] = pc; // 返回地址
    base = top;           // 不修改 top，前面已将 address+3，生成 code 后会产生 INT
语句，修改 top 值
    pc = code->getA();
    break;
```

D. OPR RET

过程结束，将栈顶指针还原为基地址（原过程栈空间分配到这），pc 指针设置为返回地址，并通过静态链找到原过程的基地址。

```
case OP::RET:
    top = base;
    pc = stack[base + 2];// 返回地址
    base = stack[base];  // 动态链
    break;
```

四、 实验结论分析与体会

通过实现 PL0 编译器，我基本理解了高级程序语言的编译过程，从词法分析，到语法分析，再到通过语法制导翻译程序的语义动作，输出一个假想计算机的目标代码。最后通过实现一个简易的解释器执行目标代码。这个实验我倾其所有，用上了大学三年积淀的编程素养以及专业知识。这个实验不仅仅用到编译原理课程所学，还涉及到许多数据结构与算法，操作系统和汇编知识。为了达到一个高效且规范的实现，我还用上了许多 C++11 以及 C++14 特性，以及正在学习的软件工程中提及的设计范式。

永远铭记这段指尖飞舞的青春。

五、 附录

1. 设计范式

一个优秀的工程不仅要前期设计方便、稳定，还要具有良好的可读性、可维护性，因此在设计 PL0 编译器时遵循了以下设计原则：

（1）项目组织原则：项目用良好的项目组织架构、目录管理架构，做到资源元件和代码文件分离，文档和编码分离。

（2）顶层设计原则：由外而内地进行设计，基于一个总体架构设计子模块。

（3）面向对象设计原则：采用面向对象设计的工程，具有易维护、质量高、效率高、易拓展等等优点，打造高内聚低耦合的系统。

2. 调试类

通过宏定义 DEBUG 启用。

```
#ifdef DEBUG
#include <iostream>
void dbg() { std::cout << "\n"; }
template<typename T, typename... A>
void dbg(T a, A... x) { std::cout << a << ' '; dbg(x...); }
#define logv(x...) std::cout << #x << " -> "; dbg(x);
```

```
#define logs(x...) dbg(x);
#else
#define logs(...)
#define logv(...)
#endif
```

3. 异常处理

同样是通过枚举类型描述各种异常情况，并使用一个哈希表存储异常对应的异常输出，通过自定义断言函数 ASSERT。

```
namespace EXCEPTION {
    enum ExceptionEnum{
        EXTRA_CHARACTERS,
        MISSING_SEMICOLON,
        MISSING_IDENTIFIER,
        DUPLICATE_IDENTIFIER,
        MISSING_EQUAL,
        MISSING_CEQUAL,
        MISSING_NUMBER,
        MISSING_THEN,
        MISSING_DO,
        MISSING_LBR,
        MISSING_RBR,
        MISSING_END,
        MISSING_SEMIC,
        NOT_A_VAR,
        NOT_A_PROCUDURE,
        NEVER_DECLARE,
        TOO_MUCH_NESTING,
        CALL_INDIRECTLY_CONTAIN,
        ERROR_USING_PROCEDURE
    };
```

```
void ASSERT(bool _, EXCEPTION::ExceptionEnum parserExceptionEnum)
{ // TODO: 更详细的异常信息、更方便的异常处理
    if (_) return;
    // 发生语法错误
#ifdef LOCAL_JUDGE
    std::cout << "--------------------" <<
EXCEPTION::EXCEPTION_ENUM_MAP[parserExceptionEnum] << std::endl;
#endif
    exit(1);
}
```

4. 本地调试

由于提交要求的输入输出形式不便于调试，又不想频繁地更改注释（频繁的提交是必然的），我通过一个本地调试宏定义两个版本的代码，在 CMakeLists.txt 中定义该宏。

```
cmake_minimum_required(VERSION 3.24)
project(PLO_Compiler23)
set(CMAKE_CXX_STANDARD 17)

set(SOURCE_FILES
        src/lexer.cpp
        src/lexer.h
        src/symbol.h
        src/utils.cpp
        src/utils.h
        src/symbolTable.h
        src/parser.h
        src/syntaxTree.cpp
        src/code.h
        src/parser.cpp
        src/exceptionHandler.h
        src/symbolTable.cpp
        src/vm.h
        src/vm.cpp)

add_definitions(-DLOCAL_JUDGE) # 本地调试宏
add_executable(PLO_Compiler23 ${SOURCE_FILES} src/main.cpp)
```

5. 项目编译脚本

```
#! /bin/bash
g++ -c lexer.cpp -o lexer.o -O2
g++ -c utils.cpp -o utils.o -O2
g++ -c syntaxTree.cpp -o syntaxTree.o -O2
g++ -c parser.cpp -o parser.o -O2
g++ -c symbolTable.cpp -o symbolTable.o -O2
g++ -c vm.cpp -o vm.o -O2

# Compiler
g++ lexer.o utils.o syntaxTree.o parser.o symbolTable.o vm.o main.cpp -o Compiler
-lm

# Interpreter
g++ lexer.o utils.o syntaxTree.o parser.o symbolTable.o vm.o interpreter.cpp -o
```

```
Interpreter -lm
```

此处粘贴你的实现代码
这一部分的代码不会被助教审阅，但可能会被审查评分依据的教授查阅。

您可以自行添加您需要提及的内容
您可以增加额外的章节来使描述清晰