東南大學

编译原理课程设计

设计报告

组长：09014222　王铎

成员：09014105　杨阳

09014116 曹心成

09014119　郑锜

东南大学计算机科学与工程学院

二0 17 年 5 月

设计任务名称	SeuLex/SeuYACC		
完成时间	2017/5/30	验收时间	2017/6/3
本组成员情况			
学　号	姓　名	承　担　的　任　务	成　绩
09014222	王铎	1.词法及文法描述文件的解析 2.正规表达式中缀转后缀算法实现 3.生成 NFA 以及确定化 NFA 算法实现 4.最小化 DFA 的算法实现 5.词法与文法分析程序的生成 6.基于 CFG 的 LR(1)下推自动机以及分析表的生成 7.LR(1)到 LALR(1)的映射 8.语法制导翻译的函数实现 9.基本块划分、寄存器分配算法的实现 10.目标代码生成算法的实现 11.设计报告中中间代码与目标代码部分的说明	
09014105	杨阳	1.词法输入文件的解析及输出格式的调整 2.设计报告中 SeuLex 部分数据结构和算法的说明	
09014116	曹心成	1.词法用户定义部分的解析和输出 2.设计报告中模块概要设计的说明	
09014119	郑锜	1.语法制导翻译的初步分析 2.设计报告中 SeuYacc 部分数据结构和算法的说明	

注：本设计报告中各部分如果页数不够，请自行扩页。原则是一定要把报告写详细，能说明本组设计的成果和特色，能够反映小组中每个人的工作。报告中应该叙述设计中的每个模块。设计报告将是评定各人成绩的重要依据之一。

1 编译对象与编译功能

1.1 编译对象

（作为编译对象的C语言子集的词法、语法描述）

1.1.1 C语言子集的词法描述

```
%{
#include <string>
#include <map>
#include <iostream>
int line = 0;
int numCount = 0;
%}

digit [0~9];
letter [A~Za~z];
id {letter}({letter}|{digit})*;
number {digit}+(.{digit}+)?(E[+-]?{digit}+)?;
%%

int {return "int";}
float {return "float";}
double {return "double";}
void {return "void";}
if   {return "if";}
else   {return "else";}
while {return "while";}
return {return "return";}
static {return "static";}
{id} {return "id";}
{number} {++numCount; return "num";}
"+" {return "+";}
"-" {return "-";}
"*" {return "*";}
"/" {return "/";}
"%" {return "%";}
"+=" {return "+=";}
"-=" {return "-=";}
"*=" {return "*=";}
"/=" {return "/=";}
"%=" {return "%=";}
"&&" {return "&&";}
"||" {return "||";}
"!" {return "!";}
"<=" {return "<=";}
```

```
">=" {return ">=";}
"=" {return "=";}
"!=" {return "!=";}
"==" {return "==";}
"<" {return "<";}
">" {return ">";}
"(" {return "(";}
")" {return ")";}
"{" {return "{";}
"}" {return "}";}
"[" {return "[";}
"]" {return "]";}
";" {++line;return ";";}
"," {return ",";}
%%

void calLine() {
	std::cout << "number of semicolon : " << line << std::endl;
}

void getNumCount() {
	std::cout << "count of num : " << numCount << std::endl;
}

void main() {
	calLine();
	getNumCount();
}
```

1.1.2 C语言子集的语法描述

```
%right '=' '+=' '-=' '*=' '/='
%left '==' '!='
%left '<' '>' '<=' '>='
%left '+' '-'
%left '*' '/'
%left '||'
%left '&&'
%right '!'
%left 'else'
%%

S : program {}
  ;
program : globalData stmts {}
```

```
      | stmts    {}
            ;
globalData : globalStmts    {
            }
           ;
globalStmts : globalStmts globalStmt {}
            | globalStmt {}
            ;
globalStmt : 'static' var_decl ';' {}
            ;
stmts      : stmts    M stmt {
                  backpatch($1.nextlist, $2.instr);
                  $$.nextlist = $3.nextlist;
                  }
      | stmt {
                  $$.nextlist = $1.nextlist;
                  }
      ;
stmt : '{' stmts    '}' {
                  $$.nextlist = $2.nextlist;
                  }
      | fun_define    {
                  returnToGlobalTable();
                  }
      | if_stmt {
                  $$.nextlist = $1.nextlist;
                  }
      | while_stmt {
                  $$.nextlist = $1.nextlist;
                  }
      | var_decl ';' {
                  $$.nextlist = $1.nextlist;
                  }
      | expr_stmt ';' {
                  }
      | 'return' expr ';' {
                  emit("return",$2.place,"","");
                  setOutLiveVar($2.place);
                  }
      ;
fun_define      : fun_decl_head BlockLeader '{' stmts '}' {
                              $$.name = $1.name;
                              }
            ;
```

```
fun_decl_head : type_spec 'id' '(' ')' {
                                $$.name = $2.lexeme;
                                createSymbolTable($2.lexeme, $1.width);
                                addFunLabel(nextInstr, $2.lexeme);
                                }
            | type_spec 'id' '(' param_list ')' {
                                $$.name = $2.lexeme;
                                createSymbolTable($2.lexeme, $1.width);
                                addToSymbolTable($4.itemlist);
                                addFunLabel(nextInstr, $2.lexeme);
                                }
              ;
param_list : param_list ',' param {
                        $$.itemlist = $1.itemlist ||    $3.itemlist;
                        }
        | param {
                        $$.itemlist = $1.itemlist;
                        }
      ;
param : type_spec 'id' {
                        $$.itemlist = makeParam($2.lexeme,$1.type,$1.width);
                        }
        | type_spec 'id' '[' int_literal ']' {
                        $$.itemlist = makeParam($2.lexeme,array($4.lexval,$1.type),$4.lexval *
$1.width);
                        }
      ;
int_literal : 'num' {
                            $$.lexval = $1.lexeme;
                            }
                ;
static_var_decl : 'static' var_decl {
                            //$$.code = $2.code;
                            }
      ;
var_decl     : type_spec 'id' {
                        enter($2.lexeme,$1.type,$1.width);
                        }
            | type_spec 'id' '=' expr {
                        p = enter($2.lexeme,$1.type,$1.width);
                        emit("",$4.place,"",p);
                        }
            | type_spec 'id' '[' int_literal ']' {
                        enter($2.lexeme,array($4.lexval,$1.type),$4.lexval * $1.width);
```

```
                        }
                    ;
type_spec    : 'int' {
                        $$.type = "int";
                        $$.width = "2";
                        }
                    | 'double' {
                        $$.type = "double";
                        $$.width = "2";
                        }
                    | 'void' {
                        $$.type = "void";
                        $$.width = "0";
                        }
                    ;
expr_stmt : 'id' '=' expr {
                        p = lookupPlace($1.lexeme);
                        if (p.empty()) error();
                        emit("",$3.place,"",p);
                        }
          ;
expr : expr '+' expr {
                        $$.place = newtemp($1.place);
            emit("ADD", $1.place, $3.place, $$.place);
                        }
        | expr '-' expr {
                        $$.place = newtemp($1.place);
            emit("SUB", $1.place, $3.place, $$.place);
                        }
        | expr '*' expr {
                        $$.place = newtemp($1.place);
            emit("MUL", $1.place, $3.place, $$.place);
                        }
        | expr '/' expr {
                        $$.place = newtemp($1.place);
            emit("DIV", $1.place, $3.place, $$.place);
                        }
        | '(' expr ')' {
                        $$.place = $2.place;
                        }
        | 'id'     {
            $$.place = lookupPlace($1.lexeme);
                    }
        | 'id' '(' arg_list ')' {
```

```
            p = gen(paramStack.size());
            while (!paramStack.empty()) {
                  emit("param", paramStack.top(),"","");
                  paramStack.pop();
            }
            emit("call", p, $1.lexeme,"");
            enter("#","int",2);
            $$.place = newtemp("#");
            emit("","#","",$$.place);
            //$$.place = "#";
          }
       | 'num' {
            $$.place = addNum($1.lexeme);
          }
      ;
arg_list    : arg_list ',' expr {
                        paramStack.push($3.place);
                        }
              |   expr   {
                        paramStack.push($1.place);
                        }
              | {
                        //paramStack.clear();
              }
      ;
if_stmt : 'if' BlockLeader '(' logic_expr ')' M stmt {
            backpatch($4.truelist, $6.instr);
            $$.nextlist = merge($4.falselist, $7.nextlist);
            }
      | 'if' BlockLeader '(' logic_expr ')' M stmt N 'else' M stmt {
            backpatch($4.truelist, $6.instr);
            backpatch($4.falselist, $10.instr);
            $$.nextlist = merge(merge($7.nextlist, $8.instr), $11.nextlist);
            }
      ;
while_stmt : 'while' BlockLeader M '(' logic_expr ')' M stmt {
            backpatch($8.nextlist, $3.instr);
            backpatch($5.truelist, $7.instr);
            $$.nextlist = $5.falselist;
            emit("j","","",$3.instr);
            }
      ;
logic_expr     : logic_expr '&&' M logic_expr {
                        backpatch($1.truelist, $3.instr);
```

```
                    $$.truelist = $4.truelist;
                    $$.falselist = merge($1.falselist, $4.falselist);
                }
            | logic_expr '||' M logic_expr {
                    backpatch($1.falselist, $3.instr);
                    $$.truelist = merge($1.truelist, $4.truelist);
                    $$.falselist = $4.falselist;
                }
            | '!' logic_expr {
                    $$.truelist = $2.falselist;
                    $$.falselist = $2.truelist;
                }
            | '(' logic_expr ')' {
                    $$.truelist = $2.truelist;
                    $$.falselist = $2.falselist;
                }
            | expr rel expr {
                    $$.truelist = makelist(nextInstr);
                    $$.falselist = makelist(nextInstr+1);
                    emit("j"+$2.op, $1.place, $3.place, "_");
                    emit("j","","","_");
                }
            | expr {
                    $$.truelist = makelist(nextInstr);
                    $$.falselist = makelist(nextInstr+1);
                    emit("j!=", $1.addr, "0", "_");
                    emit("j","","","_");
                }
            | 'true' {
                    $$.truelist = makelist(nextInstr);
                    emit("j","","","_");
                }
            | 'false' {
                    $$.falselist = makelist(nextInstr);
                    emit("j","","","_");
                }
            ;
M : {$$.instr = "LABEL_" + gen(nextInstr);}
  ;
N : { $$.instr = makelist(nextInstr);
        emit("j","","","_");
    }
  ;
BlockLeader : {
```

```
        addLeader(nextInstr);
    }
  ;
rel : '<' {$$.op = "<";}
    | '>' {$$.op = ">";}
    | '<='{$$.op = "<=";}
    | '<='{$$.op = ">=";}
    | '=='{$$.op = "==";}
    | '!='{$$.op = "!=";}
    ;
```

1.2 编译功能

（所完成的项目功能及对应的程序单元）

1.2.1 SeuLex

1. Lex 输入文件的解析

   对 C 语言子集词法描述文件 RE.l 进行解析，对应的程序单元为

   1) bool SeuLex::readREFile(const std::string& reFile)
   2) bool SeuLex::readHeadInformation()
   3) bool SeuLex::readRegularDefinition()
   4) bool SeuLex::readTranslationRule()。

2. 正规表达式的解析

   对一个正规表达式进行中缀转后缀的形式转换，对应的程序单元为

   1) std::list<ElementType> SeuLex::convertPost(std::list<ElementType>& list)

3. 一个正规表达式到 NFA 的转换算法实现,以及多个 NFA 的合并

   将后缀形式的正规表达式通过 Thompson 算法转换为 NFA，并通过两两合并的方式转换为只有一个开始状态的 NFA，对应的程序单元为

   1) bool SeuLex::createNFA()
   2) void SeuLex::PUSH(ElementType chInput)
   3) bool SeuLex::CONCAT();
   4) bool SeuLex::STAR();
   5) bool SeuLex::UNION();
   6) bool SeuLex::QUESTION();
   7) bool SeuLex::PLUS();

4. NFA 的确定化和最小化算法实现

   依据ε-闭包算法将 NFA 确定化变为 DFA，并基于自上而下的方式对等价状态进行划分，将 DFA 最小化为 DFA^0，对应的程序单元为

   1) DFAState SeuLex::Closure(DFAState T)
   2) DFAState SeuLex::Move(DFAState T, ElementType input)
   3) bool SeuLex::createDFA()
   4) bool SeuLex::optimizeDFA()

5. 返回状态与返回内容的对应

   将词法描述文件中每个返回状态的不同动作和返回内容进行对应，对应的程序单元为

   1) void SeuLex::outputAction()
   2) void SeuLex::outputMinDFA()

6. SeuLex 应用

利用 DFA[0] 和状态动作生成词法分析程序，对目标文件进行解析生成 Token 序列，对应的程序单元为

1) void SeuLex::generateParsingProgram()

2) lexHelp.h

3) lex.h

1.2.1 SeuYacc

1. Yacc 输入文件的解析

对 C 语言子集文法描述文件 Cminus.y 进行解析，包括说明部分、语法规则、程序段，对应的程序单元为

1) bool SeuYacc::readGrammer(const std::string grammerFile)

2) bool SeuYacc::readPriority()

2. 上下文无关文法到对应 LR(1)文法的下推自动机的构造

对 CFG 进行基于 LR(1)的下推自动机构造，包括 First 和 Follow 集的计算，Closure 和 Goto 算法的实现，对应的程序单元为

1) void SeuYacc::nullableAnalyze()

2) LR1State SeuYacc::Closure(LR1State result)

3) LR1State SeuYacc::GOTO(LR1State h, ElementType edge)

4) std::set<ElementType> SeuYacc::First(ElementType X)

5) std::set<ElementType> SeuYacc::Follow(ElementType X)

6) void SeuYacc::initTransition()

3. LR(1)文法的下推自动机到相应分析表的构造

基于下推自动机构建 LR(1)分析表，包括基于优先级的二义性冲突解决，对应的程序单元为

1) void SeuYacc::initParseTable()

2) ACTION_TYPE SeuYacc::conflictSolve(ElementType shiftOP, ElementType reduceOP)

4. LR(1)总控程序的构造（查表程序）

利用查找分析表在进行自底向上的规约，在 yaccHelp.h 对应的程序单元为

1) bool yyreduce(std::list<Token> _tokenList)

以及在 SeuYacc.cpp 中的

2) void SeuYacc::outputTable()

5. LR(1)到 LALR(1)的映射

将 LR(1)状态进行同芯状态合并，转化为 LALR(1)的下推自动机和分析表，对应的程序单元为

1) bool SeuYacc::transformIntoLALR()

6. 符号表的构建与相应管理程序

对全局变量和局部变量建立符号表以记录其类型，位置等信息，为后续的中间代码和目标代码生成做准备，在 supportFuntion.h 文件中，对应的程序单元为

1) std::string enter(strType name, strType type, unsigned int offset)

2) void createSymbolTable(strType name, unsigned int returnSize)

3) void addToSymbolTable(strType itemlist)

4) std::string lookupPlace(strType name)

7. 语义动作程序的加入

将文法文件中的语义动作程序进行翻译，采用后缀翻译方案的语法制导翻译，边进行规约边进行翻译，生成四元式形式的中间代码，对应的程序单元为

1) void SeuYacc::translateAction()

以及在 supportFuntion.h 文件中

2) std::string makeParam(strType name, strType type, unsigned int offset)

3) std::string newtemp(std::string var)

4) void emit(strType op, strType arg1, strType arg2, strType des)

5) std::string makelist(int i)

6) std::string merge(const std::string& p1, const std::string& p2)

7) void backpatch(std::string p, std::string label)

8. 目标代码的生成

利用中间代码与符号表，对四元式进行基本块划分、变量活跃信息与待用信息的计算、函数调用与局部变量的栈区动态分配、寄存器分配算法的实现，生成基于 80x86 的汇编代码，在 CodeGenerate.h 文件中，对应的程序单元为

1) void fillVarState(int beginIndex, int endIndex, const VarSetType& outLiveVar, VarSetType& inLiveVar)

2) int getEmptyReg()

3) int storeToGetReg()

4) void Assembly_function_prework(const Quadruple& code)

5) void GenerateAssembly(const Quadruple& code)

6) void outputAssemblyCode()

7) void tranlateIntoAssembly()

以及在 yaccHelp.h，

8) bool yyreduce(std::list<Token> _tokenList)

9. SeuYacc 的应用

利用 SeuLex 生成的词法分析程序得到 Token 序列，在利用分析表和语义动作生成语法分析程序，利用该程序对目标文件进行自底向上的规约动作，同时基于语法制导翻译生成中间代码，再通过算法生成目标代码，对应的程序单元为

1) void SeuYacc::generateParsingProgram()

2) yacc.cpp

3) yaccHelp.h

2. 主要特色

2.1 SeuLex

1.正规表达式中可以出现*|+？~等运算

2.在词法描述中，用户可以加上自身定义的函数以方便调用

2.2 SeuYacc

1.文法描述中支持语义动作，用户可以自定义以生成中间代码或实现特点功能

2.文法说明部分支持操作符的结合性和优先级的定义，以解决冲突

3.进行 LALR(1)的映射，以提升查表规约的效率

4.程序支持生成四元式形式的中间代码，供将来的代码优化模块使用

5.程序支持简单的 X86 汇编代码生成，包括跳转循环等基本语句

3 概要设计与详细设计

（由总到分地介绍 SeuLex 和 SeuYACC 的设计，包括模块间的关系，具体的算法等。采用面向对象方法的，同时介绍类（或对象）之间的关系。在文字说明的同时，尽可能多采用规范的图示方法。）

3.1 概要设计

（以描述模块间关系为主）

3.1.1 SeuLex 模块关系

模块	输入	输出	调用关系
词法描述文件解析	RE.l	lex.h RE vector	调用正规表达式解析模块
正规表达式解析	RE vector	post RE Action vector	
生成与合并 NFA	post RE Vector of Action	FA Stack Final State Set Action map	正规表达式解析完成后运行
确定化 NFA	FA Stack Final State Set Action map	DFA final state set DFA state Transfer DFA acion map	生成与合并 NFA 模块完成后运行
最小化 DFA	DFA final state set DFA state Transfer DFA acion map	DFA^0 final state set DFA^0 state Transfer DFA^0 acion map	确定化 NFA 模块完成后运行
词法分析程序生成	Vector of Action DFA state Transfer	lex.h actionLex.h tableLex.h	

3.1.2 SeuYacc 模块关系

模块	输入	输出	调用关系
描述文件解析	Cminus.y	Table of nonterminal Table of terminal Map of priotity Rules of left Associative	
下推自动机构造	Production Vector Table of terminal	LR(1) State Transition	若描述文件解析模块执行成功
分析表构造	LR(1) Transition	LR1 Parse Table	若描述文件解析模块执行成功
LR(1)到 LALR(1)的映射	LR(1) Parse Table LR(1) State Transition	LALR(1) Parse Table LALR(1) State Transition	若映射失败，则将输出清空
语义动作程序 目标代码生成	Production Action	actionYacc.h	目标代码随规约动作生成
基本块划分 控制流图	Middle Code	FlowGraph	
目标代码生成	Basic Block	Assembly Code	对每个基本块进行代码生成
文法分析程序生成		yacc.cpp actionYacc.h tableYacc.h	调用语义动作程序模块

3.2 详细设计

（以描述数据结构及算法实现为主）

3.2.1 数据结构—Token

Definition: list<Token> _tokenList;

```
struct Token {
    std::string _lexecal;
    std::string _attribute;
    unsigned int _innerCode;
    unsigned int _line;
    unsigned int _offset;
};
```

遇到的 Token 值都会存在_tokenList 中，因为它只可能往里面添加元素不可能删减元素，所以这里为了快速添加操作直接将 Token 列表作为一个 list 来实现。

Token 的第一个数据成员是它的词素，第二个数据成员是其属性包括关键字、数字和 ID。在词法分析的过程中，词法分析器根据属性值将词素和某种词法单元的模式匹配，并被识别为该种词法单元的一个实例。如果 Token 的属性不是一个关键字，第三个数据成员为它所对应的内部码。对于关键字来说，_innerCode 的值恒为 0。第四个数据成员_line 存储 Token 在源文件中出现的行数，第五个数据成员_offset 存储 Token 在该行中的偏移量。这两个数据成员是为了在词法和语法分析过程中更加方便地输出错误信息，在正常的 Token 表示中并不是必要的。

3.2.2 数据结构—正规表达式

Definition: typedef unsigned int ElementType;

```
typedef std::string Action;
std::map<std::string, std::list<ElementType> > _REs;
std::vector<std::list<ElementType> > _postREs;
std::vector<Action> _actionMap;
std::map<char, ElementType> _inputRE;
std::map<ElementType, char> _outputRE;
const ElementType EPSILON = 0;
const ElementType RIGHT_BRACE_OP = 1;
const ElementType LEFT_BRACE_OP = 2;
const ElementType CONCAT_OP = 3;
const ElementType UNION_OP = 4;
const ElementType STAR_OP = 5;
const ElementType QUESTION_OP = 6;
const ElementType PLUS_OP = 7;
unsigned int _reIndex = 7;
ElementType _minREelement = 8;
```

选择 map 数据结构设置_inputRE 存储出现的字符并且把一个用 unsigned int 来定义的 ElementType 值作为它的关键码，也称为内部码。同时为了输出和方便检查，也利用 map 数据结构设置了_outputRE 通过关键码获取相应字符。

在 C++ STL 库中，map 模板在一些版本中是用红黑树来实现的，因此理论上来说，它的检查和插入操作的效率都是 O(logN)，N 为元素的数量。同理，接下来的报告中也存在由于同样的原因选择 map 而不是 vector 来实现数据存储的情况，不再赘述。

当读完 RE.l 之后，正规定义存储在_REs 中。_REs 的 key 是头字符串，对应的 value 值为正则定义所定义的具体的字符序列。它们都是中缀表达式，在 NFA 的构造过程中，会将它们转化为后缀表达式并存储在_postREs 中。_postREs 中的后缀表达式序列生成 NFA 并进行归并，不需要存储 key 值，所以直接用 vector 对后缀表达式进行存储即可。对于正规表达式中的动作 op，我们用一个常量值来定义它们，常量值代表着动作的优先级并用到后缀转换中。具体的算法见 3.2.3。当一个正规表达式需要返回动作时，将其所要返回的动作存储在_actionMap 中，利用 vector 结构进行存储，下标即为对应的 action ID 值。

3.2.3 算法实现—正规表达式中缀转后缀

INPUT:字母表 Σ 上的一个中缀正规表达式 r

OUTPUT: 一个后缀表达式 r'

METHOD:

定义优先级：LEFT_BRACE < CONCAT < UNION < STAR = QUESTION = PLUS

从中缀正规表达式第一个元素开始，对每一个元素 k，如果 k 为：

a. UNION : 把栈中优先级大于等于 k 的操作符都出栈，将元素 k 入栈

b. CONCAT : 把栈中优先级大于等于 k 的操作符都出栈，将元素 k 入栈

c. STAR : 将其添加到后缀正规表达式中

d. PLUS : 将其添加到后缀正规表达式中

e. QUESTION : 将其添加到后缀正规表达式中

f. LEFT_BRACE : 将元素 k 入栈

g. RIGHT _BRACE : 将操作符依次出栈直到有一个 LEFT_BRACE 操作符出栈

h. 输入字符 : 将其添加到后缀正规表达式中

当一个操作符从栈中弹出时，将其添加到后缀正规表达式中直到符号栈为空。

IMPLEMENT:

SeuLex.cpp line 482-529

std::list<SeuLex::ElementType> SeuLex::convertPost(std::list<ElementType> & list)

3.2.4 数据结构—NFA

```
Definition:   typedef typename std::deque<State*> FA;
              typedef typename std::set<State*> DFAState;
              typedef unsigned int IDType;
              State* _NFA_start;
              IDType _nextStateID = 0;
              std::stack<FA> _operandStack;
              std::set<IDType> _NFAfinalStateSet;
              std::map<IDType, IDType> _NFAAction;
struct State {
    struct Edge {
      State* _next;
      ElementType _edge;
    };
    std::list<Edge> _transition;
    IDType _id;
};
```

状态的结构体中包括一个转换边的 list 和一个唯一的状态 ID。每一个正规表达式都有其对应的 NFA 并且会被_NFA_start 开始状态通过 ε 边合并。_operandStack 用于单个 NFA 的构造和合并操作的迭代。具体算法见 3.2.5。当每个单独的 NFA 的一个终态有返回操作时，其状态 ID 被存储在_NFAfinalStateSet 中并且相应的返回动作存储在_NFAAction 中，_NFAAction 选择用 map 结构，key 为 NFA 状态 ID，value 值为对应的 action ID。

3.2.5 算法实现—NFA 生成与合并

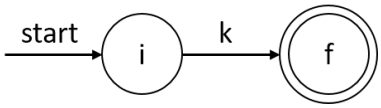
INPUT: 一个后缀正规表达式 r

OUTPUT: 一个接受 L(r)的 NFA

METHOD:

从后缀正规表达式的第一个元素开始，对于元素 k：

如果 k 是一个输入字符，创建两个新状态和一个从开始状态到结束状态的 k 转换边并将这个 NFA 入栈；



如果是 CONCAT 操作符，依次出栈两个元素分别为 A、B（B 先出栈 A 后出栈）。从 A 的终止状态添一条 ε 边到 B 的初始状态，将新生成的这个 NFA 入栈；

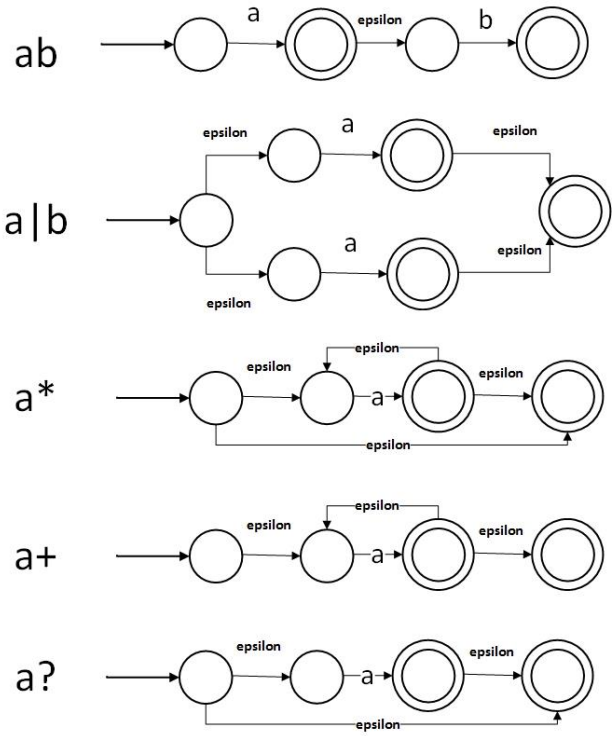
如果是 UNION 操作符，依次出栈两个元素分别为 A、B（B 先出栈 A 后出栈）。创建两个新的状态作为新的 NFA 的初始和结束状态。创建一条 ε 边从初始状态连接到 A、B 的初始状态，创建一条 ε 边从 A、B 的结束状态连接到新的结束状态，将新生成

的这个 NFA 入栈；

如果是 STAR 操作符，出栈一个元素 A，创建两个新的状态并将其插入到状态双端队列的两侧作为新的初始和结束状态，对 A 的初始和结束状态之间添加 ε 边并对新的终止和结束状态之间用 ε 边连接，将新生成的 NFA 入栈；

如果是 PLUS 操作符，出栈一个元素 A，创建两个新的状态并将其插入到状态双端队列的两侧作为新的初始和结束状态，对 A 的初始和结束状态之间添加 ε 边，将新生成的 NFA 入栈；

如果是 QUESTION 操作符，出栈一个元素 A，创建两个新的状态并将其插入到状态双端队列的两侧作为新的初始和结束状态，对新的终止和结束状态之间用 ε 边连接，将新生成的 NFA 入栈。



IMPLEMENT:

SeuLex.cpp	line 677-708	bool SeuLex::createNFA();
SeuLex.cpp	line 534-549	bool SeuLex::PUSH();
SeuLex.cpp	line 553-570	bool SeuLex::CONCAT();
SeuLex.cpp	line 574-596	bool SeuLex::STAR();
SeuLex.cpp	line 600-624	bool SeuLex::UNION();
SeuLex.cpp	line 628-648	bool SeuLex::QUESTION();
SeuLex.cpp	line 652-672	bool SeuLex::PLUS();

ADDITION:

对每个函数来说都可能会产生新的状态，在设计时在类 SeuLex 的析构函数中会对这些空间进行释放。采用 shared_ptr 是这个问题的另一种解决办法但是因为它一度是基于 Boost 库使用的，所以我们没有采取这种方法。另外，java 平台支持垃圾自动回收技术可以很好地解决这方面的问题。

3.2.6 数据结构—DFA

Definition: typedef typename std::set<State*> DFAState;
typedef typename std::map<ElementType, IDType> StatesTransfer;
std::map<DFAState, DFAState> _ClosureMap;
std::map<DFAState, IDType> _DFAStateMap;
std::set<IDType> _DFAfinalStateSet;
std::vector<StatesTransfer> _DFAStateTranfer;
std::map<IDType, IDType> _DFAAction;

通过 CLOSURE 算法，数个 NFA 状态会被合并成一个 DFA 状态，所以用集合来存储 NFA 来保证 NFA 不会重复。_DFAStateMap 的 key 为对应的 DFAState，通过 map 将其映射到 0 开始计数的新的 DFA 状态 ID 上。当一个 DFA 状态包含了一个 NFA 的终态时，它就是一个终态，会被保存在_ DFAfinalStateSet 中并且在_DFAAction 中存储当以该终态 ID 为 key 时对应的 action ID 值。_DFAStateTranfer 向量存储状态转换信息，其中向量下标对应着 DFA 的状态 ID。_ClosureMap 则是用来存储进行 ε 闭包算法前后的 DFA 状态对应关系，以避免重复多次的计算，提升效率。

3.2.7 算法实现—NFA 确定化

INPUT:一个有开始状态和结束状态集 F 的 NFA

OUTPUT: 一个有开始状态和结束状态集 F'的 DFA

METHOD:

1.DFA 的开始状态为 NFA 的开始状态的 ε 闭包；

2.对于每一个新产生的 DFA 状态，对每个输入字符执行以下操作：

(1)通过输入字符移动到新创建的状态，这里可能会移动到多个 NFA 状态，生成一个 NFA 状态集 T；

(2)通过对(1)的结果进行求 ε 闭包创建新的状态加入到 T。在 ε 闭包的过程中我们可能得到一个已经存在在我们集合 T 中的状态。通过 ε 闭包扩展会产生一个新的 NFA 状态集 T'，由这个状态集组成了新的 DFA 状态。注意这里是从一个或者多个 NFA 状态构造一个单独的 DFA 状态。

3.对每个新创建的 DFA 状态，执行步骤 2。

4.DFA 的终止状态是至少包含了 N 的一个终止状态的状态集的集合。

IMPLEMENT:

SeuLex.cpp line 758-821
bool SeuLex::createDFA();

SeuLex.cpp line 713-736
SeuLex::DFAState SeuLex::Closure(DFAState T)

SeuLex.cpp line 742-752
SeuLex::DFAState SeuLex::Move(DFAState T, ElementType input)

3.2.8 数据结构—DFA^0

Definition: typedef typename std::map<ElementType, IDType> StatesTransfer;
std::set<IDType> _minDFAfinalStateSet;
std::vector<StatesTransfer> _minDFAStateTranfer;
std::map<IDType, IDType> _minDFAAction;

因为大多数状态是等价的，所以分析器需要将它们合并成一个状态来降低状态转换的规

模。和 DFA 类似，_minDFAStateTranfer 向量存储状态转换信息，其中向量下标对应着 DFA^0 的状态 ID，_minDFAfinalStateSet 也存储了 DFA^0 的终态 ID，_minDFAAction 中存储当以该 DFA^0 终态 ID 为 key 时对应的 action ID 值.

另外，DFA 和 DFA^0 都最多只会返回一个动作。否则，正规表达式就会具有二义性。

3.2.9 算法实现—DFA 最小化

INPUT: 一个 DFA D，状态集合为 S，输入字母表为 Σ，开始状态为 s_0，终止状态集为 F。

OUTPUT: 一个 DFA D'，它和 D 接受相同的语言，且状态数最少。

METHOD:

1. 首先构造包含两个组 F 和 S-F 的初始划分Π，这两个组分别为 D 的终止状态组合和非终止状态组。
2. 应用下面的过程来构造新的分化Π_{new}：

   最初，令 Π_{new} =Π;

   对（Π中的每个组 G）{

   将 G 分划为更小的组，使得两个状态 s 和 t 在同一小组中当且仅当对于所有的输入符号 a，状态 s 和 t 在 a 上的转换都到达Π中的同一组；

   /*在最坏的情况下，每个状态各自组成一组*/

   在 Π_{new} 中将 G 替换为对 G 进行分划得到的那些小组；

   }
3. 如果 Π_{new} =Π，令 Π_{final} =Π并接着执行步骤 4；否则，用 Π_{new} 替换Π并重复步骤 2。
4. 在分划 Π_{final} 的每个组中选取一个状态作为该组的代表。这些代表构成了最小状态 DFA D'的状态。D'的其他部分按如下步骤构建：

   (a) D'的开始状态是包含了 D 的开始状态的组的代表。

   (b) D'的终止状态时那些包含的 D 的终止状态的组的代表。注意每个组中要么只包含终止状态，要么只包含非终止状态，因为我们从一开始就把这两类状态分开了，而步骤 2 中的过程总是通过分解已经构造得到的组来得到新的组。

   (c) 令 s 是 Π_{final} 中某个组 G 的代表，并令 DFA D 中在输入 a 上离开 s 的转换到达状态 t。令 r 为 t 所在组 H 的代表。那么在 D'中存在一个从 s 到 r 在输入 a 上的转换。注意在 D 中，组 G 中的每一个状态必然在输入 a 上进入组 H 中的某个状态，否则，组 G 应该已经被步骤 2 分割成更小的组了。

IMPLEMENT:

SeuLex.cpp line 827-976

bool SeuLex::optimizeDFA()

ADDITION:

为了分化组 G，选择初始状态 k 作为分割符来构成子分组，状态等价于 k 的构成一个分组、其余的为另一个分组。也就是说，它是一个二叉树。为了解决回头看问题，只有遍历一次中不在出现分裂才停止算法。这个算法比较暴力并且费时。也许自底向上进行合并可能会有更好的效率。

3.2.10 数据结构—文法标识符

Definition: typedef unsigned int ElementType;

typedef unsigned int IDType;

std::map<std::string, ElementType> _nonterminalTable;

std::map<std::string, ElementType> _terminalTable;

std::map<ElementType, bool> _nonterminalNullable;
std::map<ElementType, bool> _leftAssociative;
std::map<ElementType, unsigned int> _priority;
std::map<ElementType, std::set<ElementType> > _firstMap;
std::vector<std::string> _word;

类型 ElementType 表示元素类型，我们暂时采用的整型 int,定义这个类型是为了修改方便。类型 IDType 表示 ID 类型，作用同上。

因为非终结符和终结符使用频率很高，所以在程序中，所有的非终结符和终结符都用无符号整型数来表示。虽然使用 string 会更好让人理解，但是使用 string 会使存储空间和计算时间的成本增加。所以在我们的程序中，string 表达式存储在_word 向量中，只是为了供输出函数使用。

选择 map 数据结构是为了语法符号的快速存在性检查。在 C++ STL 中，map 是用红黑树实现的,所以构造时时间会有一点长但是只需要一次。元素查找的时间复杂度是 O(nlogn)。

所以我选择 std::map<std::string, ElementType> 数据结构实现 _nonterminalTable 和 _terminalTable。

对于非终止的可空属性，我使用 map <unsigned int，bool>来存储_nonterminalNullable 的数据。 每个非终结符都将在其中，并且当它可以导出空序列时，它的 bool 值将为 true，而相反为 false。 它在添加产生式时被分配，并且将在可空分析成员函数中完成赋值。

考虑到终结符的左/右结合属性，我使用 map<unsigned int, bool>来存储_leftAssociative 的数据。当终结符是左结合时，它的 bool 值是 true，右结合时是 false。它在读语法文件的附加条件时被赋值，缺省值是左结合。同时，它也用来解决冲突问题。

对于优先级问题,我使用 map<ElementType, unsigned int>来存储。优先级为 0 表示最低，随值增加优先级增加。它在读语法文件的附加条件时被赋值，越后面的条件优先级越高。

对于 map<ElementType, std::set<ElementType> > _firstMap，它的值是对< A, first(A)>。对于每个语法符号 A，计算 first(A)是为了 LR(1)项中的预测符。因为 first 集在构造状态转换图时要重复用到，所以我用这种方式存储。当需要但又没有计算过时，计算 first 集并且存储，当需要并且已在 map 中时，只需要返回集合。这样做平衡了时间成本和存储空间因为它只在需要的时候计算。

3.2.11 数据结构—产生式

Definition: std::deque<ProductionItem> _productionVector;
std::map<unsigned int, std::string> _productionAction;

```
struct ProductionItem {
        ElementType _head;   // nonterminal for production head
        std::list<ElementType> _bodyList;// nonterminal and terminal for production body
        size_t _bodyLength;
        IDType _index;
        ElementType _op;
 };
```

设计结构体 ProductionItem 是为了表示上下文无关文法的单个产生式，_head 表示产生式的头部（非终结符）。产生式的主体部分_bodyList 存储为 list 数据结构，因为在大多数场景中，它是顺序访问的，并且增加新的元素的成本也不高。_index 数据成员表示它在产生式向量中的位置。_op 代表着这个产生式的操作。在大多数情况下，这是无用的，但是在分析表中解决冲突时，_op 又是必要的，因为移进-规约冲突和规约-规约冲突需要由用户提供的

附加条件解决，比如操作优先级和左/右结合方式。

3.2.13 算法实现—First

INPUT:语法符号 X

OUTPUT:从 X 导出的终结符集合

METHOD：

1. If X is a terminal, then FIRST(X) = {X}.

2. If X is a nonterminal and X->Y_1 Y_2 ... Y_k is a production for some k>=1, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is , Y_1 ... $Y_{i-1}\Rightarrow_* \epsilon$. If ϵ is in FIRST($Y_j$) for all j = 1, 2, ..., k, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1\Rightarrow_* \epsilon$, then we add FIRST($Y_2$), and so on.

3. If X-> ϵ is a production, then add ϵ to FIRST (X).

IMPLEMENT:

SeuYacc.cpp line 577-623

std::set<SeuYacc::ElementType> SeuYacc::First(ElementType X);

3.2.14 算法实现—Follow

INPUT: 语法符号 X

OUTPUT: 跟随 X 的终结符集合

METHOD：

1. Put $ into the Follow(X).

2. If A->αXβ, put FIRST(β) into the Follow(X) except for ε.

3. If A->αX or A->αXβ(ε∈first(β)), put Follow(A) into Follow(X).

IMPLEMENT:

SeuYacc.cpp line 627-688

std::set<SeuYacc::ElementType> SeuYacc::Follow(ElementType X);

ADDITION:

在基于 LR(1) 文法进行分析表构建时，并不需要 Follow 函数，本程序是为了不时之需而实现了该算法。

3.2.15 算法实现—Closure

INPUT: LR(1)项的集合 T

OUTPUT:内部状态扩展后的 LR(1)项的集合 T

METHOD：

SetOfItems CLOSURE(T) {

repeat

for (each item $[A \rightarrow \alpha \cdot B\beta, a]$ in T)

for (each production $B \rightarrow \gamma$ in G')

for (each terminal b in FIRST(βa))

add $[B \rightarrow \cdot\gamma, b]$ to set T;

until no more items are added to T;

}

IMPLEMENT:

SeuYacc.cpp line 493-555

SeuYacc::LR1State SeuYacc::Closure(LR1State T);

3.2.16 算法实现—Goto

INPUT: LR(1)项的集合 T 和语法符号 X

OUTPUT: LR(1)项的集合 W.

METHOD：

SetOfItems GOTO(T,X) {

Initialize J to be the empty set;

for (each item $[A \rightarrow \alpha \cdot X\beta, a]$ in T)

add item $[A \rightarrow \alpha X \cdot \beta, a]$ to set J;

return CLOSURE(J);

}

IMPLEMENT:

SeuYacc.cpp line 559-573

SeuYacc::LR1State SeuYacc::GOTO(LR1State h, ElementType edge);

3.2.17 数据结构—LR(1)项目及下推自动机

Definition: std::vector<TransitionItem> _LR1stateTransition;

```
struct TransitionItem {
    LR1State _lr1state;
    std::map<ElementType, IDType> _transMap;
    IDType _index;
};
struct LR1State {
    list<LR1Item> itemList;
};
struct LR1Item {
    ProductionItem* _prod;
    std::list<ElementType>::iterator _dot; // dot position pointer to the latter (non)terminal
    std::set<ElementType> _predSet;
};
```

下推自动机_LR1stateTransition 用 vector 数据类型，其下标即为 LR(1)状态的标号，其

值与对应 TransitionItem 对象的_index 相等。每个 LR(1)状态转移情况用 TransitionItem 对象来表示，包含一个 LR(1)状态和转移_transMap。_transMap 的 key 值为转移边，value 值为跳转的状态标号。

对于每个 LR(1)状态包含一个 LR(1)项目的 list 对象。而每一个 LR(1)项目包含其代表产生式的指针，点所在的位置和预测符集合。使用产生式的指针的形式可是节省大量存储空间（享元模式的思想）。

3.2.18 算法实现—LR(1)项目构造

INPUT: 扩展语法 G'

OUTPUT: LR(1) 项的集合.

METHOD:

Initialize C to CLOSURE({$[S' \rightarrow \cdot S, \$]$});

```
repeat
    for (each set of items T in C)
        for (each grammar symbol X)
            if (GOTO(T,X) is not empty and not in C)
                add GOTO(T,X) to C
until no new sets of items are added to C;
```

IMPLEMENT:

SeuYacc.cpp line 331-365

void SeuYacc::initTransition();

3.2.19 数据结构—LR(1)分析表

Definition: std::vector<std::vector<TableItem> > _LR1parseTable;

```
struct TableItem {
    ACTION_TYPE _action;
    IDType _index;     // index of production or goto state id
};
```

结构体 TableItem 包含两个数据成员。_action 是无符号整型值，对应的是类中枚举变量 ACTION_TYPE { SHIFT = 1, REDUCTION, GOTO_STATE, ERROR, ACCEPT }。_index 属性代表不同的数字。对于 SHIFT 和 GOTO_STATE 动作，_index 的值是下一个状态的下标，对于 REDUCTION 动作，它的值是规约项下标，对于 ERROR 动作，它的值是 0，对于 ACCEPT 动作，它的值是 0，因为 ACCEPT 只是一个特殊的规约项。

3.2.20 算法实现—LR(1)分析表构造

INPUT: 扩展语法 G'

OUTPUT:语法 G'的规范 LR(1)分析表（函数 ACTION 和 GOTO）

METHOD:

1. Construct C' = {$I_0, I_1, \ldots, I_n$}, the collection of sets of LR(1) items for G';

2. State I of the parser is constructed from I_i. The parsing action for state i is determined as follows.

i. If $[A \rightarrow \alpha \cdot d\beta, b]$ is in I_i and GOTO(I_i, d) = I_j, then set ACTION[i, d] to "shift j". Here a must

be a terminal.

ii. If $[A \rightarrow \alpha\cdot, b]$ is in I_i, A ≠ S', then set ACTION[i, a] to "reduce $A \rightarrow \alpha$"

iii. If $[S' \rightarrow S\cdot, \$]$ is in I_i, then set ACTION[i, $] to "accept".

If any conflicting action result from the above rules, use additional conditions to solve. Otherwise, the algorithm fails to produce a parser.

3. The goto transitions for state i are constructed for all nonterminals A using the rule: If GOTO(I_i, A) = I_j, then GOTO[i, A] = j.

4. All entries not defined by rule b and c are made "error".

5. The initial state of the parse is the one constructed from the set of items containing $[S' \rightarrow S\cdot, \$]$.

IMPLEMENT:

SeuYacc.cpp line 368-426

void SeuYacc::initParseTable();

SeuYacc.cpp line692-706

SeuYacc::ACTION_TYPE SeuYacc::conflictSolve(ElementType shiftOP, ElementType reduceOP);

ADDITION:

当产生了移进-规约冲突时，我使用冲突解决函数来返回最后的动作（移进或者冲突）。当规约操作优先级高时，返回规约。当它们优先级相同并且规约满足左结合时，返回规约。

3.2.21 数据结构—LALR(1)项目

Definition: std::map<IDType, TransitionItem> _LALR1stateTransition;

```
struct TransitionItem {
    LR1State _lr1state;
    std::map<ElementType, IDType> _transMap;
    IDType _index;
};
```

Map 类型的 LALR(1)下推自动机_LALR1stateTransition，key 值为对应状态的标号，value 值为转移变量，其结构与 LR(1)相同，不赘述。

3.2.22 算法实现—LR(1)到 LALR(1)的映射

INPUT: 扩展语法 G'

OUTPUT: 语法 G'的规范 LALR(1)分析表（函数 ACTION 和 GOTO）

METHOD:

1. Construct C' = {I_0, I_1, …, I_n}, the collection of sets of LR(1) items.
2. For each core present among the set of LR(1) items, find all sets having that core, and replace these sets by their union.
3. Let C'' = {J_0, J_1, …, J_m} be the resulting sets of LR(1) items. The parsing actions for state i are constructed from J_i in the same manner as in LR(1) parsing table construction algorithm. If there is a parsing action conflict, the algorithm fails to produce a parser, and the grammar is said not to be LALR(1).

d. The GOTO table is constructed as follows. If J is the union of one or more sets of LR(1) items, that is , $J = I_1 \cap I_2 \cap \ldots \cap I_k$, then the cores of $GOTO(I_1, X)$, $GOTO(I_2, X)$, ... ,$GOTO(I_k, X)$ are the same, since $I_1, I_2, \ldots, I_k$ all have the same core. Let K be the union of all sets of items having the same core as $GOTO(I_1, X)$, Then GOTO(J, X) = K.

IMPLEMENT:

SeuYacc.cpp line 430-489

```
bool SeuYacc::transformIntoLALR();
```

3.2.23 数据结构—分析栈

Definition: deque<StackItem> stack;

```
struct StackItem {
    IDType _state;
    ElementType _symbol;
    std::map<std::string, std::string> _definitionMap;
};
```

在自底向上规约的过程中，使用分析栈 stack。其每个元素 StackItem 包括当前状态标号，文法符号和供语法制导翻译所用的属性 map。由于当规约时需要提取栈里特定位置的属性值，因此采用 deque 来模拟栈的行为。

3.2.24 算法实现—LR 规约

INPUT:输入字符串 w 和语法的 LR 分析表（函数 ACTION 和 GOTO）

OUTPUT:如果 w 在 L(G)中，输出一个自底向上分析的规约；否则，输出错误提示

METHOD:

Initially, the parser has s0 on its stack, where so is the initial state, and w$ in the input buffer. The parser then executes the program in below.

```
Let a be the first symbol of w$;
  while (1) {
    let s be the state on top of the stack;
    if (ACTION[s, a] = shift t) {
            push t onto the stack;
            let a be the next input symbol;
    } else if (ACTION[s, a] = reduce A->β) {
            do SDT actions
            pop |β| symbols off the stack;
            let state t now be on the top of stack;
            push GOTO[t, A] onto the stack;
            output the production A->β;
    } else if (ACTION[s, a] = accept) break;
    else call error-recovery routine;
}
```

IMPLEMENT:

yaccHelp.h line 32-113

```
bool yyreduce(std::list<Token> _tokenList);
```

3.2.25 数据结构—符号表

```
Definition:   std::vector<SymbolTable*> symbolTables;
struct SymbolTable {
    std::string _funName;
    unsigned int _returnSize = 0;
    unsigned int _variableOffset = 0;
    unsigned int _paramOffset = 0;
    int _beginIndex = -1;
    int _endIndex = -1;
    struct varState {
        bool _live;
        int _nextUse;
        bool _inM;
        int _inR;
        bool _unsigned;
        unsigned int _offset;
        unsigned int _space;
        std::string _name;
        std::string _type;
        std::string _place;
    };
    std::vector<varState> _field;
    std::set<int> _leaders;
}
```

全局变量和每个函数的局部变量都各自生成一张符号表，对应于 SymbolTable 的一个对象。_funName 为该函数的函数名（由于难度限制，并未采用函数签名的机制，同时全局符号表的_funName 为“global”）。Size 的计量单位为字节，_returnSize 为该函数返回值所占的字节数，_variableOffset 为局部变量所占全部的字节数，_paramOffset 为函数实参所占的全部字节数。_beginIndex 和_endIndex 分别是该函数对应的中间代码的入口序号和出口序号（不包含后者）。_leaders 为该函数所有基本块的入口序号。_field 为该函数所有变量的域，以定义的顺序依次存放。

对于每个变量对应一个 varState 对象，_unsigned 表示是否是无符号数（true 表示是无符号数），_offset 表示该变量相对于变量存储空间基址 Base 的偏移量（注意 X86 机型为小端存储模型），即[Base-_offset]用来访问该变量,_space 表示该变量所占用的字节数,_name 为变量名，_type 为类型，_place 为本程序对应的动态栈区分配模型的变量访问地址，具体见“算法实现—栈式存储管理”。

为了在代码生成中进行寄存器分配，需要为每个四元式的每个变量附加两个信息（活跃 Live 和待用 Next-Use）。而填写这两项信息的算法（见“算法实现—变量活跃与待用信息计算”）需要符号表中为每个变量增加两栏，用于暂存活跃与待用信息。因此 varState 中的_live 和_nextUse 即为对应信息。同时寄存器分配算法需要了解各个变量在寄存器中或是内存中,因此增加_inM 和_inR 数据成员，_inM 等于 true 时表示该变量在内存中有副本，_inR 等于-1 时表示该变量不在寄存器中，当-1<_inR<REGISTER_NUM 时，表示变量在数值对应的寄存器中。

由于难度较高，本版本并未支持块内变量，而且支持的变量大小为 2 个字节（如用 AX

寄存器进行存储计算)。临时变量与局部变量处理相同，由于不额外设计常量表，因此数值量也存放在该符号表中，不过不占用空间，即_space 为 0。

3.2.26 算法实现—标号回溯

METHOD:

在对转移语句进行语法制导翻译的过程中，有若干种的翻译方式，本程序基于修改 LR 文法的 SDT 的自底向上语法分析，通过保存 TRUE-CHAIN 和 FALSE-CHAIN 来进行标号的回填，具体文法见程序所给的 Cminus.y 文件。对同一个出口的跳转指令合并成链，在该出口中间代码生成时将其标号进行回填，结合中间代码标号存储结构 labelMap 即可实现。

IMPLEMENT:

supportFunction.h　line 165-167

```
std::string makelist(int i)
```

supportFunction.h　line 170-172

```
std::string merge(const std::string& p1, const std::string& p2)
```

supportFunction.h　line 174-197

```
void backpatch(std::string p, std::string label)
```

3.2.27 数据结构—四元式

```
Definition: std::vector<Quadruple> middleCode;
            std::map<int, std::string> labelMap;
struct Quadruple {
    int _type;
    std::string _label;
    std::string _op;
    std::string _des, _arg1, _arg2;
    bool _typeDes = false, _typeArg1 = false, _typeArg2 = false;
    bool _liveDes, _liveArg1, _liveArg2;
    int _nextDes, _nextArg1, _nextArg2;
};
```

对于每一个四元式形式的中间代码对应一个 Quadruple 对象，所有四元式存放在数组 middleCode 中，数组下标即为其标号。_type 表示该四元式的类型码，如无条件跳转语句_type 等于 20，加法语句_type 等于 10 等，具体数值参见 Quadruple 的源代码，设置_type 变量是为了在对不同四元式进行翻译时进行区分，提高效率。_label 为该四元式的标签，无标签时为空；_op 为操作符，如“ADD”,“j<”等；_des 为目的操作数，_arg1 为第一操作数，_arg2 为第二操作数，即为 _des = _arg1 _op _arg2。_typeDes, _typeArg1, _typeArg2 分别表示它们对应的变量或是常量，默认为 false 即为变量，当为常量时不做翻译动作。_liveDes, _liveArg1, _liveArg2 分别为该变量是否还会在基本块内被引用，false 表示不会被引用。_nextDes, _nextArg1, _nextArg2 表示会在哪个四元式被引用，数值为四元式标号，无引用则为-1。计算规则见“算法实现—变量活跃与待用信息计算”。

labelMap 为记录中间代码标签的数据结构。Key 为中间代码的标号，value 为对应该标号的标签。在语法制导翻译的过程中，对应标号的中间代码可能尚未生成，因此将标签信息进行保存，在翻译完成后统一更新。

3.2.28 数据结构—基本块

Definition: std::map<int, BasicBlock> flowGragh;

```
struct BasicBlock {
    int _begin;
    int _end;
    std::vector<int> _predecessors;
    std::vector<int> _successors;
    std::set<std::string> _inLiveVar;
    std::set<std::string> _outLiveVar;
};
```

基本块是一段程序中一段顺序执行的语句序列，只有一个入口与一个出口，内部不存在跳转指令跳出基本块的范围。

对于四元式序列的基本块划分算法较为简单。首先求出各个基本块的入口语句，满足一下三种情况其一：1)程序的第一个语句；2)能有条件转移语句或无条件跳转语句到达的语句；3)紧跟在条件转移语句后面的语句。然后对于每一条入口语句，构建其所属的基本块，由该入口语句到下一个入口语句（不包含后者）之间的语句组成。

在表示每一个基本块的 BasicBlock 对象中，_begin 为入口语句的四元式标号，_end 为出口语句的标号，_predecessors 为该基本块的前继（基本块标号的集合），_successors 为后继，_inLiveVar 和_outLiveVar 分别为入口和出口的活跃变量集。本程序中取该基本块的_begin 作为其标号，同时为控制流图 flowGragh 的 key 值。

3.2.29 数据结构—汇编代码

Definition: std::vector<Assembly> assemblyCode;
std::map<size_t, std::string> AssemblyLabelMap;

```
struct Assembly {
    std::string _op;
    std::string _des;
    std::string _arg;
    std::string _label;
};
```

本编译器的目标代码是基于 X86 的汇编代码。每条汇编代码为一个 Assembly 对象，其中_op 为指令的操作，_arg 为源操作数，_des 为目标操作数，_label 为该指令的标签。即为 _label : _op _des _arg。两个操作数不能同时为存储器寻址的方式。对于跳转语句，_arg 为空，_des 为目标指令的标签。对于 RET,PUSH,POP 等只有一个变量的指令，将变量置于_des 中，_arg 置空。

AssemblyLabelMap 是记录汇编代码的标签信息，key 为汇编代码的标号，value 为对应此标号的标签。由于一句中间代码会对应多条目标代码，因而在做中间代码翻译的过程中记录对应的标签信息，在完成翻译工作后统一更新目标代码的标签信息。

3.2.30 算法实现—变量活跃与待用信息计算

INPUT: 基本块入口四元式序号 begin,出口 end,出口活跃变量集 outLiveVar

OUTPUT: 填入信息的四元式序列,基本块入口活跃变量集 inLiveVar

METHOD:

将符号表中与本基本块有关的各变量的 Next-Use 栏全部置为空

根据基本块出口的活跃变量集，将相应变量的 Live 置 true
FOR i = end TO begin DO BEGIN
　　取序号为 i 的四元式 A = B op C
　　将符号表中变量 A,B,C 的 Live 与 Next-Use 填到变量 A,B,C 的附加信息两栏内
　　清除符号表中 A 的 Live 和 Next-Use 项
　　将符号表中 B,C 的 Live 置为 true, Next-Use 置为 i
END
注：如果四元式为 A= op B 或者 A = B，则执行步骤相同，只是不涉及 C
　　对于 PUSH，POP 等只有一个操作数的指令类似，忽略立即数
IMPLEMENT:
　　CodeGenerate.h　line 26-83
　　void fillVarState(int begin, int end, const VarSetType& outLiveVar, VarSetType& inLiveVar)

3.2.31 数据结构—寄存器和变量地址描述

　　Definition:　std::vector<std::set<std::string> >* RValue;
　　　　std::vector<int>* RNextUse;

为了在目标代码生成过程中进行寄存器分配，需要随时掌握各寄存器的使用情况（是空闲，或是已分配给某个变量，或是几个变量共用一个寄存器）以及这些变量的待用信息，因此为每个寄存器附加两项信息：RValue 和 RNextUse。数组的下标对应寄存器的编号，从 0 到 REGISTER_NUM-1。RValue 中填写 R 寄存器中寄存了哪些变量，比如 RValue[Ri]={}, RValue[Ri]={X}, RValue[Ri]={X,Y，…},。在 RNextUse 中填写 R 寄存器中变量的待用信息，为所存变量中待用四元式序号最小的值（即最快会被用到，都不会被用到时为-1）。

在生成目标代码时，还需要了解某变量是否在寄存器中，若在，则不必从内存调入寄存器。同时腾出寄存器时需要知道该寄存器中的变量是否在内存中有副本，若有，则不必生成将寄存器中的内容保存到内存的指令。相关的变量地址描述数据存储在符号表中，见“数据结构—符号表”

3.2.32 算法实现—寄存器分配

METHOD:

由于难度所限，本程序并未对循环等结构进行寄存器的分配优化，即没有计算执行代价来调整策略充分发挥运算速度。本程序仅采用最为简单的分配原则。

对于四元式 A = B op C,本程序假设要求变量 B 一定要在寄存器中，变量 C 可在寄存器中或内存中，结果 A 必在寄存器中。这是一种简单的模型机指令系统，对于真实的情况由于过于复杂本程序不进行讨论。

1.要为变量 B 分配寄存器时，先查寄存器描述数组 RValue 看是否有空寄存器，若有，则直接分配，若没有，按 storeToGetReg 方法：

(1) 寄存器 Ri 中的变量在内存中有副本，即对应的_inM==true

(2) 寄存器 Ri 中的内容在最远的将来使用，即 RNextUse 的值最小，并生成指令

　　MOV M, Ri　　// 腾出寄存器 Ri,仅但 Ri 的内容在内存中无副本时生成，同时修改
　　　　// 相应的 M 的变量地址描述，inR 清空，inM 为真

利用 storeToGetReg 方法获得一个可用的寄存器 regB，接下来生成指令

　　MOV regB, B　　// 将 B 的内容载入寄存器 regB

这样 B 的值在寄存器 regB 中了，紧接着执行 RValue[regB] = {B}, RNextUse[regB] = B.nextUse,修改符号表中的变量地址描述数组，即 B._inR = Ri, B._inM = false。

2.要为变量 A 分配寄存器，方法是

(1) 选择原存放 B 的寄存器 regB 来存放 A

如果 RValue[regB]={B}且 B.live==false 或者四元式为 A = A op C 的形式（此时 B==A），可以直接生成目标指令

OP regB, C // regB 改存 A

然后修改两个变量地址描述，B._inR 重置，B._inM 不变，A._inR=regB，A._inM=false，表示 B 不在寄存器中，但在内存中，A 仅在寄存器中。同时 RValue[regB]={A},修改 RNextUse 的值。

(2) 另外选择一个寄存器存放 A

如果 1)的条件不满足，当存在空闲的寄存器 regA，则直接分配给 A，生成指令

MOV regA, regB

OP regA, C // 此时 B 仍在 regB 中，而 A 在 regA 中

然后 RValue[regA]={A},更新 RNextUse 的值，A._inR=regB，A._inM=false。

(3) 如果(1)和(2)的条件都不满足，那么采用 storeToGetReg 的方法获得可用寄存器 regA，生成指令

MOV regA, A

OP regA, C

然后修改对应的 A 的变量地址描述，并修改 RValue[regA]和 RNextUse[regA]。

3. 若 C 在寄存器中，根据 C 的活跃与待用信息修改 RValue 和 RNextUse，以便腾出不用的寄存器给后续的代码使用。

对于其他形式的四元式，仿造上述算法即可生成目标代码。特别指出对于 A=B 形式的四元式，如果 B 的值在寄存器中，此时不需要生成目标代码，只需在 RValue[regB]中增加一个 A，把 regB 同时分配给 A 和 B，修改 A._inR，同时如果后续 B 不在被引用，可以删去相应的 B 的信息。注意的是此时由于没有生成目标代码，为防止指令标签位置出错，生成“JMP $2”指令以进行过渡。

一旦处理完一个基本块的所有的四元式，如果其属于出口活跃集变量，则需要将其保持入内存中，腾空所有寄存器为下一个基本块做准备。对于循环时的公共变量可以采用算法以优化寄存器分配效率，由于时间限制，本程序不进行进一步实现了。

IMPLEMENT:

CodeGenerate.h line 85-91

int getEmptyReg()

CodeGenerate.h line 93-124

int storeToGetReg()

CodeGenerate.h line127-267

void Assembly_A_BopC(const Quadruple& code)

CodeGenerate.h line 270-315

void Assembly_A_opB(const Quadruple& code)

CodeGenerate.h line 318-349

void Assembly_A_B(const Quadruple& code)

CodeGenerate.h line 351-354

void Assembly_j(const Quadruple& code)

CodeGenerate.h line 357-414

void Assembly_jrop(const Quadruple& code)

3.2.33 算法实现—栈式存储管理

METHOD:

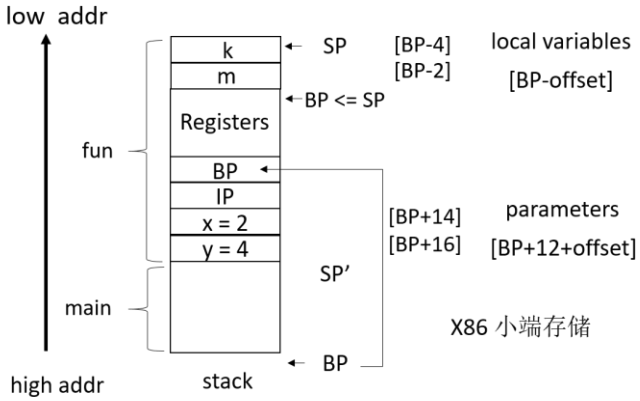
由于内存空间有限，因而进行函数调用时，采用栈区进行动态分配空间供局部变量存储以及回溯的信息存储。参考 C 语言的活动记录与函数的调用约定__stdcall，构建适合本程序的管理规则。

1.对于参数传递，依据声明顺序从右向左压入栈区，较为符合实参存放的顺序

2.函数调用时（程序会自动压入返回的地址 IP），压入原先的栈基址 BP，再依次压入寄存器保存现场，再将当前栈顶指针 SP 赋给 BP 成为现行函数的新栈基址，移动 SP 给函数分配局部变量总大小的空间

3.访问局部变量时，采用[BP-offset]的基址寻址方式；访问实参时，采用[BP+offset]的基址寻址方式；访问全局变量时，采用 varName 的形式，直接定位到静态全局数据区。

4.函数退出时，首先将返回值赋给寄存器 AX（本程序为简单起见，利用 AX 传递函数返回值，实际情况中更为负责），将 BP 赋给 SP，再依次弹出寄存器恢复现场，弹出 BP 恢复原有的基址，程序自动弹出调用地址 IP。最后由 callee 自己清理堆栈中的参数，即对应的汇编代码为 RET _paramOffset，弹出压入栈区的实参。



symbol table of fun

name	type	offset	space
x	int	2	2
y	int	4	2
m	int	2	2
k	int	4	2

```
//suppose : int var is of 2B space
int fun(int x, int y) {
    int m = x + y;
    int k = m + y;
    return k;
}
void main() {
    int a = fun(2,4);
}
```

IMPLEMENT:

CodeGenerate.h　line 417-422

　　void Assembly_param(const Quadruple& code)

CodeGenerate.h　line 425-443

　　void Assembly_call(const Quadruple& code)

CodeGenerate.h　line 445-468

　　void Assembly_function_prework(const Quadruple& code)

CodeGenerate.h　line 471-504

　　void Assembly_function_prework(const Quadruple& code)

3.2.34 算法实现—目标代码生成

METHOD:

本程序采取的基本思路是在规约过程中进行语法制导翻译，以生成四元式形式的中间代码，接着根据符号表和寄存器分配算法进行目标代码的生成。

主体是函数 tranlateIntoAssembly，其对目标函数进行基本块的划分，进而创建控制流图，利用“变量活跃与待用信息计算的算法”，对每个基本块计算各个基本块之间的入口和出口

活跃变量集，根据前继和后继的关系进行传递。随后按照基本块的顺序调用 GenerateAssembly 函数对每个四元式进行翻译，生成目标代码。完成所有基本块翻译过后，在 outputAssemblyCode 函数内，按照 X86 汇编代码格式进行数据段，堆栈段和代码段的输出。

IMPLEMENT:

CodeGenerate.h line 507-552

void GenerateAssembly(const Quadruple& code)

CodeGenerate.h line 555-602

void outputAssemblyCode()

CodeGenerate.h line 604-736

void tranlateIntoAssembly()

4 使用说明

4.1 **SeuLex 与 SeuYacc 使用说明**

Step 1: 使用 cmd 命令行进入 CompilerGen.exe 所在目录

Step 2: 输入 CompilerGen RE.l Cminus.y 回车

（RE.l 为词法描述文件，Cminus.y 为文法描述文件）

目录下出现 tableLex.h tableYacc.h actionLex.h actionYacc.h lex.h yacc.cpp 等文件，是编译器所使用到的用户定义的部分所生成的文件，结合目录下的所需文件夹里的文件，运行 yacc.cpp 即可生成编译器

注：目录下已给出一个可用的编译器 Compiler.exe

4.2 Compiler 使用说明

Step 1: 使用 cmd 命令行进入 CompilerGen.exe 所在目录

Step 2: 输入 Compiler test.cpp 回车 (test.cpp 为所要编译的目标文件)

目录下出现 token_list.txt 分词序列，reduce_sequence.txt 规约序列, symbol_table_file.txt 符号表，middle_code.txt 中间代码 目标代码 code.asm 文件

5 测试用例与结果分析

5.1 测试用例

测试源文件代码如下 source.cpp

```
static int p;

int funct(int k) {
	int b;
	b = k + k;
	return b;
}

int minus(int k, int m) {
	int n;
	n = k - m;
	return n;
}

void main() {
	int a;
	int b;
	int c;
	c = minus(c,c);
	a = funct(c);
	b = c + funct(a) - 1;
	a = a - 2 + b;
	b = a + b;
	if (a < b || a != c) {
		a = b;
		a = a + b;
	}
	a = a + b;
	if (a > b && !(a - b != c + b)) {
		a = b;
		b = a - b;
		if (a > b) {
			a = b + a;
		}
		else {
			a = b - a;
			a = b + a;
		}
	}
	while (a < b) {
```

```
            a = b + b;
            b = c;
        }
        a = b;
}
```

5.2 结果分析

对应此测试用例生成的汇编代码 code.asm 如下

```
.MODEL SMALL
.STACK
.DATA
    p   DD 00000000H
.CODE
                funct PROC
                PUSH BP
                PUSH BX
                PUSH CX
                PUSH DX
                PUSH SI
                PUSH DI
                MOV BP,SP
                SUB SP,4
                MOV AX,[BP+14]
                ADD AX,[BP+14]
                MOV SP,BP
                POP DI
                POP SI
                POP DX
                POP CX
                POP BX
                POP BP
                RET 2
                funct ENDP
                minus PROC
                PUSH BP
                PUSH BX
                PUSH CX
                PUSH DX
                PUSH SI
                PUSH DI
                MOV BP,SP
                SUB SP,4
                MOV AX,[BP+14]
                SUB AX,[BP+16]
```

```
        MOV SP,BP
        POP DI
        POP SI
        POP DX
        POP CX
        POP BX
        POP BP
        RET 4
        minus ENDP
main : MOV AX,@DATA
        MOV DS,AX
        MOV BP,SP
        SUB SP,42
        MOV AX,[BP-6]
        PUSH AX
        PUSH AX
        MOV [BP-6],AX
        CALL minus
        MOV [BP-10],AX
        MOV AX,[BP-10]
        PUSH AX
        MOV [BP-6],AX
        CALL funct
        MOV [BP-12],AX
        MOV AX,[BP-12]
        PUSH AX
        MOV [BP-2],AX
        CALL funct
        MOV [BP-14],AX
        MOV AX,[BP-6]
        MOV BX,AX
        ADD BX,[BP-14]
        MOV [BP-18],BX
        SUB [BP-18],1
        MOV CX,[BP-18]
        MOV DX,2
        SUB DX,[BP-2]
        ADD DX,CX
        MOV SI,DX
        ADD SI,CX
        MOV [BP-2],DX
        MOV [BP-4],SI
        MOV AX,[BP-2]
        CMP AX,[BP-4]
```

```
                JL LABEL_30
                JMP LABEL_28
LABEL_28 : MOV AX,[BP-2]
                CMP AX,[BP-6]
                JNE LABEL_30
                JMP LABEL_33
LABEL_30 : MOV AX,[BP-4]
                ADD AX,[BP-4]
                MOV [BP-2],AX
LABEL_33 : MOV AX,[BP-2]
                ADD AX,[BP-4]
                MOV [BP-2],AX
                MOV AX,[BP-2]
                CMP AX,[BP-4]
                JG LABEL_37
                JMP LABEL_53
LABEL_37 : MOV AX,[BP-2]
                MOV BX,AX
                SUB BX,[BP-4]
                MOV CX,[BP-6]
                MOV DX,CX
                ADD DX,[BP-4]
                MOV [BP-30],BX
                MOV [BP-32],DX
                MOV AX,[BP-30]
                CMP AX,[BP-32]
                JNE LABEL_53
                JMP LABEL_41
LABEL_41 : MOV AX,[BP-4]
                MOV BX,AX
                SUB BX,[BP-4]
                MOV [BP-2],AX
                MOV [BP-4],BX
                MOV AX,[BP-2]
                CMP AX,[BP-4]
                JG LABEL_46
                JMP LABEL_49
LABEL_46 : MOV AX,[BP-4]
                MOV BX,AX
                ADD BX,[BP-2]
                MOV [BP-2],BX
                JMP LABEL_53
LABEL_49 : MOV AX,[BP-4]
                MOV BX,AX
```

```
                SUB BX,[BP-2]
                MOV CX,AX
                ADD CX,BX
                MOV [BP-2],CX
LABEL_53 :      MOV AX,[BP-2]
                CMP AX,[BP-4]
                JL LABEL_55
                JMP LABEL_59
LABEL_55 :      MOV AX,[BP-4]
                ADD AX,[BP-4]
                MOV BX,[BP-6]
                JMP LABEL_53
LABEL_59 :      MOV AX,[BP-4]
                MOV AH,4CH
                INT 21H
                END main
```

利用汇编器和 MASM 软件进行汇编代码运行，程序退出前数据情况如下，结果与预期相符，运行正确。

```
AX=0000  BX=0000  CX=FFFF  DX=0001  SP=03D6  BP=0400  SI=0000  DI=0000
DS=0B44  ES=0B22  SS=0B46  CS=0B32  IP=0127   NV UP EI PL ZR NA PE NC
0B32:0127 8B46FC        MOV     AX,[BP-04]                      SS:03FC=0000
-t

AX=0000  BX=0000  CX=FFFF  DX=0001  SP=03D6  BP=0400  SI=0000  DI=0000
DS=0B44  ES=0B22  SS=0B46  CS=0B32  IP=012A   NV UP EI PL ZR NA PE NC
0B32:012A B44C          MOV     AH,4C
-u
0B32:012A B44C          MOV     AH,4C
0B32:012C CD21          INT     21
```

6 课程设计总结（包括设计的总结和需要改进的内容）

在课程设计的过程当中，我们利用到了大量编译原理的知识，还极大地锻炼了编程与算法设计的能力。

首先我们小组选择 C++作为编程语言，主要原因是考虑到 C++是偏向于底层的编程语言，运行效率很高，并且可以直接使用 STL 提供的大量基础的数据结构和算法，使得开发效率极高。

其次是对于词法分析与语法分析，由于采用紧耦合的方式，由语法分析程序对词法分析得出的 Token 序列进行自底向上的规约，两者之间传递的数据结构与格式需要有一定的配合。在存储自动机结构方面，我们采用了若干 map 类型的变量以提升查找的效率。同时对于 Closure 和 First 等支撑函数，我们采取了适当的结构来存储结构，避免多次重复计算。

在面对语法制导翻译时，我们尝试了许多种方案，原因在于生成中间代码的形式与符号表的结构直接影响后续代码优化和目标代码生成。这帮助我们对于不同的方法有了更加深刻的认识。由于目前手中的工具只支持我们运行 8086 汇编代码，因此我们选择以 x86 作为目标代码的形式。这牵涉到的问题远超自己先前设想的数量和难度，比如如何分配寄存器以提升运行效率，比如函数调用和变量存储空间的动态分配问题。

由于时间和精力限制，我们完成的编译器还存在非常多的问题和不足。对于需要多个寄存器配合的乘除等操作目前无法进行处理；对于数组和指针等较为复杂的类型还无法进行处理；浮点数转化为汇编代码的形式还没制定；没有对中间代码进行优化导致有大量重复而无效的计算；寄存器分配算法较为基础，没有针对执行代价进行计算优化；等等。当学的越多，越发现自己不知道的事情比想象中多很多。真正要实现一个编译器，我们现在所掌握的知识完完全全不值一提。

在自己寻找资料进行自学探索的过程中，对于编译原理有了不一样的感觉。不仅仅停留在原理上的理解推导，更重要的是如何踏踏实实地用代码实现，这完全是两个层次的能力区别。同时对于计算机整个体系有了进一步的认知，这才是我们所学到最重要的东西。

7 教 师 评 语

签名：_____

附：包含源程序、可运行程序、输入数据文件、输出数据文件、答辩所做 PPT 文件、本设计报告等一切可放入光盘的内容的光盘。