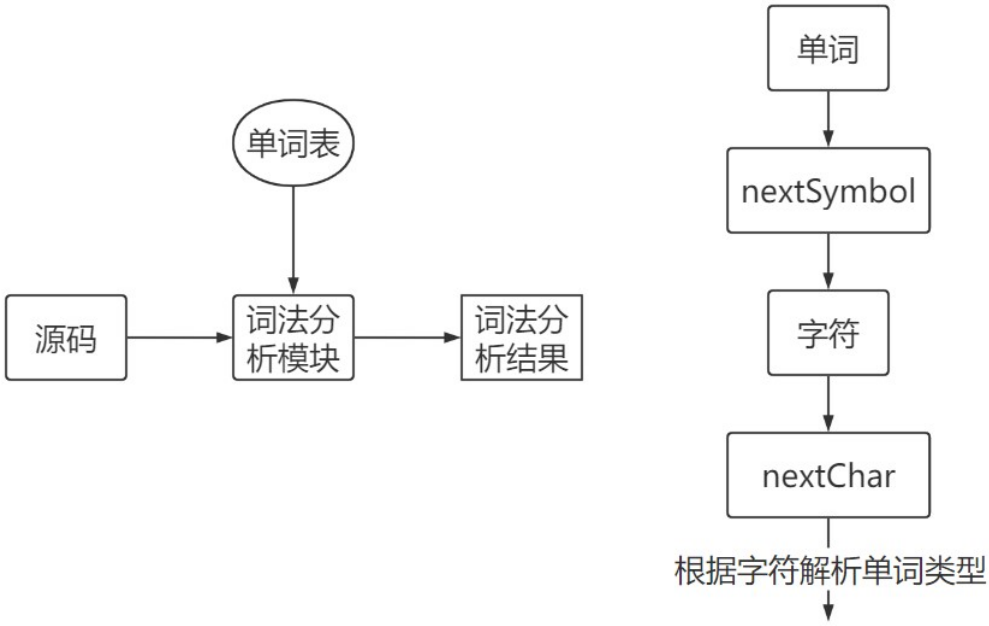
2021词法分析阶段设计文档

编码前设计

词法分析主要目的是将整个代码解析并输出每个单词的类别。

架构设计

主要架构有单词表和词法分析模块。单词表主要储存词法分析类别码的类型，词法分析模块则将代码进行解析并输出词法分析后的结果。



编码后设计

架构设计

在解析 Ident 时有可能是关键词，因此加入了关键词列表判断该单词是否是关键词，若是则将单词归类为对应的单词类别，否则归类为 Ident。

数据结构设计

使用 `map` 存储关键词表。

模块

构造了语法分析器的类 LexicalAnalyzer，主要解析源码并输出词法分析的结果。词法分析模块结构如下：

```
class LexicalAnalyzer {
public:
    LexicalAnalyzer(string buffer);
    void initialSymbols(); // 初始化单词表
    void initialTokens();  // 初始化关键词表
    void printSrc();
    void analyze();        // 解析源码
```

```
    void nextSymbol();     // 解析下一个单词
    char nextChar();       // 下一个字符
    bool checkUnary(string sym); // 检查是不是 UnaryExp
    char viewNextChar();   // 查看下一个单词
    void output();         // 输出词法分析结果
private:
    int ptr;    // 源码指针
    string src; // 源码
    map<string, symbol> tokens; // 关键词表
    vector<pair<symbol, string>> symbols; // 解析结果
};
```

nextSymbol

该函数主要功能为解析单词，若遇到单词的字符前缀或后缀相同，例如 `!`、`!=` 和 `==`、`<=`、`>=`，将他们转换到中间状态根据下一个字符决定怎么解析该单词；否则直接解析该单词。

在解析 Ident 时需查找关键词表检验该标识符是不是关键词，若是则归类为对应的关键词，否则一律归类为标识符。

checkUnary

该函数是检查该字符串是不是一元符号，例如 `+`、`-`、`*` 等字符，若是则返回 true。

debug 经历

判别注释的逻辑写错，造成有 `/***/` 的情况会退出注释状态。

无法解析前缀为 `_` 的 Ident。

2021 语法分析阶段设计文档

将源码中根据语法进行分析，从最开始的非终结符自顶向下逐个分析所有符号。

编码前设计

由于要使用左递归自顶向下分析，而实验规定文法中无法进行左递归分析，所以将文法稍微进行修改

```
乘除模表达式 MulExp → UnaryExp | MulExp   ('*' | '/' | '%') UnaryExp
转换为    => MulExp → UnaryExp {('*' | '/' | '%') UnaryExp}

加减表达式 AddExp → MulExp | AddExp ('+' | '-') MulExp
转换为 => AddExp → MulExp {('+' | '-') MulExp}

关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
转换为 => RelExp → AddExp {('<' | '>' | '<=' | '>=') AddExp}

相等性表达式 EqExp → RelExp | EqExp ('==' | '!=') RelExp
转换为    => EqExp → RelExp {('==' | '!=') RelExp}

逻辑与表达式 LAndExp → EqExp | LAndExp '&&' EqExp
转换为    => LAndExp → LAndExp {'&&' EqExp}

逻辑或表达式 LOrExp → LAndExp | LOrExp '||' LAndExp
转换为    => LOrExp → LOrExp {'||' LAndExp}
```

一般上有 `{}` 或 `first(<文法>)` 的文法：（仅列部分文法）

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef
声明 Decl → ConstDecl | VarDecl
变量定义 VarDef → Ident { '[' ConstExp ']' } | Ident { '[' ConstExp ']' } '=' InitVal
语句 Stmt → LVal '=' Exp ';' | [Exp] ';' | Block |
           'if' '(' Cond ')' Stmt [ 'else' Stmt ] |
           'while' '(' Cond ')' Stmt |
           'break' ';' | 'continue' ';' | 'return' [Exp] ';' |
           LVal '=' 'getint''('')'';' |
           'printf''('FormatString{','Exp}')'';'
```

需要提前查看终结符或使用回溯，而这里设计的编译器使用的是向前看的策略解决这类问题。

编码后设计

语法分析设计部分中使用递归下降法进行自顶向下分析程序语法。分析结束后将结果保存在 `out` 容器内。

```
class GrammarAnalyzer {
public:
    GrammarAnalyzer(vector<pair<symbol, string>> symbols);
    void analyze();
    void output();
private:
    vector<pair<symbol, string>> src;
    vector<string> out;
    pair<symbol, string> sym;

    int ptr;
    pair<symbol, string> nextSymbol();
    pair<symbol, string> viewNextSymbol();

    pair<symbol, string> viewNextSymbol(int i);
    void _CompUnit();       // CompUnit 非终结符
    void _Decl();           // Decl 非终结符
    void _ConstDecl();      // ConstDecl 非终结符
    void _BType();          // BType 非终结符
    void _ConstDef();       // ConstDef 非终结符
    void _ConstInitVal();   // ConstInitVal 非终结符
    void _VarDecl();        // VarDecl 非终结符
    void _VarDef();         // VarDef 非终结符
    void _InitVal();        // InitVal 非终结符
    void _FuncDef();        // FuncDef 非终结符
    void _MainFuncDef();    // MainFuncDef 非终结符
    void _FuncType();       // FuncType 非终结符
    void _FuncFParams();    // FuncFParams 非终结符
    void _FuncFParam();     // FuncFParam 非终结符
    void _Block();          // Block 非终结符
    void _BlockItem();      // BlockItem 非终结符
    void _Stmt();           // Stmt 非终结符
    void _Exp();            // Exp 非终结符
    void _Cond();           // Cond 非终结符
    void _LVal();           // LVal 非终结符
    void _PrimaryExp();     // PrimaryExp 非终结符
    void _Number();         // Number 非终结符
```

```
    void _UnaryExp();       // UnaryExp 非终结符
    void _UnaryOp();        // UnaryOp 非终结符
    void _FuncRParams();    // FuncRParams 非终结符
    void _MulExp();         // MulExp 非终结符
    void _AddExp();         // AddExp 非终结符
    void _RelExp();         // RelExp 非终结符
    void _EqExp();          // EqExp 非终结符
    void _LAndExp();        // LAndExp 非终结符
    void _LOrExp();         // LOrExp 非终结符

    bool isExp();           // 查看当前终结符是否是 Exp
    bool isBlockItem();     // 查看当前终结符是否是 BlockItem
    void _ConstExp();       // ConstExp 非终结符
    void _Ident();          // Ident 终结符
    void pushPair();

    void FormatString();    // FormatString 终结符
};
```

由于在编码时有些地方的判断会比较复杂，因此加了两个函数 `isExp` 和 `isBlockItem` 把可能的终结符都放在该函数上。

```
bool GrammarAnalyzer::isExp() {
    return sym.tkn == SEMICN  || sym.tkn == IDENFR   || sym.tkn == PLUS     ||
           sym.tkn == MINU    || sym.tkn == NOT      || sym.tkn == LPARENT  ||
           sym.tkn == INTCON;
}

bool GrammarAnalyzer::isBlockItem() {
    return sym.tkn == INTTK   || sym.tkn == CONSTTK  || isExp() ||
           sym.tkn == IFTK    || sym.tkn == WHILETK  || sym.tkn == CONTINUETK ||
           sym.tkn == BREAKTK || sym.tkn == PRINTFTK || sym.tkn == RETURNTK ||
           sym.tkn == LBRACE;
}
```

debug 经历

在 `Stmt` 上对语法上有些理解错误

```
语句 Stmt → LVal '=' Exp ';' | [Exp] ';' | Block |
           'if' '(' Cond ')' Stmt [ 'else' Stmt ] |
           'while' '(' Cond ')' Stmt |
           'break' ';' | 'continue' ';' | 'return' [Exp] ';' |
           LVal '=' 'getint''('')'';' |
           'printf''('FormatString{','Exp}')'';'
```

由于 `Stmt` 上 `Exp` 和 `LVal` 中有 `ident` ，之前没判断直接分析成 `LVal` 的情况，因此需要向前看有没有 `=` 在 `Stmt` 内，若有则分析到 `LVal` 即可停止，否则将 `Exp` 的分析都输出到 `output.txt` 上。

2021 错误处理阶段设计文档

错误处理功能主要将程序中的语法错误或语义错误都返回给用户知道，而该实验需实现：

错误类型	错误类别码	解释
非法符号	`a`	格式字符串中出现非法字符报错行号为 所在行数。
名字重定义	`b`	函数名或者变量名在**当前作用域**下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为 所在行数。
未定义的名字	`c`	使用了未定义的标识符报错行号为 所在行数。
函数参数个数不匹配	`d`	
函数参数类型不匹配	`e`	
无返回值的函数存在不匹配的 `return` 语句	`f`	报错行号为 `return` 所在行号。
有返回值的函数缺少 `return` 语句	`g`	只需要考虑函数末尾是否存在 `return` 语句，**无需考虑数据流**。报错行号为函数**结尾的** `}` 所在行号。
不能改变常量的值	`h`	为常量时，不能对其修改。报错行号为 所在行号。
缺少分号	`i`	报错行号为分号**前一个非终结符**所在行号。
缺少右小括号 `)`	`j`	报错行号为右小括号**前一个非终结符**所在行号。
缺少右中括号 `]`	`k`	报错行号为右中括号**前一个非终结符**所在行号。
`printf` 中格式字符与表达式个数不匹配	`l`	报错行号为 `printf` 所在行号。
在非循环块中使用 `break` 和 `continue` 语句	`m`	报错行号为 `break` 与 `continue` 所在行号。

编码前设计

处理错误处理可分为语法错误和语义错误，除了 a, i, j, k 以外，其他错误都归类为语义错误。语法错误可以在递归下降时发现并及时补正，若有语义上的错误则需要检查源码的上下文判别各种的语义错误如 `break` 和 `continue` 语句是否在 `while` 模块内。为了方便查看上下文，因此使用了抽象语法树 (Ast) 搜查上下文。

为了方便搜查上下文，在这里设计的语法树结构中把一些必要的讯息给整合起来，例如 `Decl` 中存放数据类型、维度、变量名、是否为常量或参数等。以下都是各结点的设计：

```
Ast -> 语法树，存放各种结点的数据结构
```

```
Program -> 存放多个 ProgramItem 的一个容器
ProgramItem -> 程式内的讯息，如 Decl 和 Func
Decl -> 存放定义变量/常量的信息，如数据类型 (BType)、维度 (Dims) 、变量名 (name)、是否为
        常量、参数和指针
Func -> 存放函数定义的信息，如返回类型 (returnType)、函数名 (name)、参数 (fParams) 和程
        式块 (Block)
Block -> 存放多个 BlockItem 的一个容器
BlockItem -> 程式块内的讯息，如 Decl 和 Stmt
Stmt -> 程式中的语句，这里分为程式块 (Block)、表达式语句 (ExpStmt)、返回语句 (ReturnStmt)、条
件语句 (CondStmt)、循环语句 (LoopStmt)
ExpStmt -> 表达式语句，存放着表达式 (Exp)
ReturnStmt -> 返回语句，存放着 return 和返回值 (Exp)
CondStmt -> 条件语句，存放着 if 和 while 语句，else 语句必须在 if 语句之前
LoopStmt -> 循环语句，存放着 break 和 continue
Exp -> 表达式结点，表达式可分为一元表达式 (UnaryExp) 、二元表达式 (BinaryExp)、格式化字符
       串 (FormatString)
UnaryExp -> 一元表达式，存放着一元符号 (UnaryOp)、表达式 (Exp)
BinaryExp -> 二元表达式，存放着二元符号和左表达式 (lhs) 和右表达式 (rhs)
FormatString -> 格式化字符串，存放着格式化字符串的结点
LVal -> 左值表达式，是一元表达式 (UnaryExp) 的一种，存放变量名 (name) 和使用维度 (dims)
AssignExp -> 赋值表达式，是二元表达式 (BinaryExp) 的一种，存放 LVal 为 lhs，Exp 为
             rhs
CallExp -> 调用表达式，是一元表达式的一种，存放函数符号 (func) 和实参 (rParams)
```

编码后设计

以下为 `Ast` 的所有类定义的属性与方法：

```
class Ast {
public:
    Ast();
    explicit Ast(Program* p, ErrorHandling* err);
    void traverse();
private:
    Program* program;
};

class ProgramItem {
public:
    ProgramItem();
    virtual void traverse(int lev);
};

class BlockItem {
public:
    BlockItem();
    virtual void traverse(int lev);
};

class Decl : public ProgramItem, public BlockItem {
public:
    Decl();
    Decl(Symbol _bType, Symbol _name);
    void setType(Symbol _bType);
    Type getType();
    void setConst();
    bool isConst();
```

```
    void addDim();
    void addDim(Exp* size);
    int getDim();
    std::vector<int> getDims();
    void addParam();
    void setName(Symbol _name);
    Symbol getName();
    void addInitVal(Exp* init);
    std::vector<int> getInitVal();
    void traverse(int lev) override;
private:
    std::vector<Exp*> initVal;
    int level;
    bool Const, pointer, param;
    Type bType;
    Symbol name;
};

class Stmt : public BlockItem {
public:
    Stmt();
    virtual void traverse(int lev);
};

class Block : public Stmt {
public:
    Block();
    void addItem(BlockItem *item);
    void setLevel(int lev);
    int getLevel();
    void addLoop();
    bool isLoop();
    void setLBrace(Symbol sym);
    Symbol getLBrace();
    void setRBrace(Symbol sym);
    Symbol getRBrace();
    ReturnStmt* evalReturn();
    std::vector<LoopStmt*> evalLoop();
    void checkLoop();
    void traverse(int lev) override;
private:
    std::vector<BlockItem*> block_items;
    Symbol lBrace, rBrace;
    int level;
    bool loop;
};

class Func : public ProgramItem {
public:
    Func();
    Func(Symbol _returnType, Symbol _name, Block* _block);
    Func(Symbol _returnType, Symbol _name, Block* _block, std::vector<Decl*> v);
    Symbol getName();
    std::vector<Decl*> getParams();
    Type getType();
    void checkReturn();
    void checkLoop();
    void traverse(int lev) override;
```

```
private:
    Type returnType;
    Symbol name;
    std::vector<Decl*> FParams;
    Block *block;
};

class Exp {
public:
    Exp();
    void addCond();
    bool isCond();
    virtual void traverse(int lev);
    virtual Type evalType();
    virtual int evalInt();
private:
    bool cond;
};

class ExpStmt : public Stmt {
public:
    ExpStmt();
    void addExp(Exp* e);
    void traverse(int lev) override;
private:
    Exp* exp;
};

class CondStmt : public Stmt {
public:
    CondStmt();
    CondStmt(Symbol name, Exp* e, Stmt* stmt);
    void addElse(Stmt* stmt);
    Symbol getSym();
    Stmt* getIfStmt();
    Stmt* getElseStmt();
    void traverse(int lev) override;
private:
    Symbol sym;
    Exp* condExp;
    Stmt* ifStmt;
    Stmt* elseStmt;
};

class LoopStmt : public Stmt {
public:
    LoopStmt();
    explicit LoopStmt(Symbol name);
    Symbol getSym();
    void traverse(int lev) override;
private:
    Symbol sym;
};

class ReturnStmt : public Stmt {
public:
    ReturnStmt();
    explicit ReturnStmt(Symbol name);
```

```
    void addExp(Exp* e);
    Exp* getExp();
    Symbol getSymbol();
    void traverse(int lev) override;
private:
    Exp* exp;
    Symbol sym;
};

class UnaryExp : public Exp {
public:
    UnaryExp();
    void addOp(std::string _op);
    std::string getOp();
    void setExp(Exp* e);
    Exp* getExp();
    void addSym(Symbol name);
    virtual void traverse(int lev);
    virtual Type evalType();
    virtual int evalInt();
private:
    Symbol sym;
    Exp* exp;
    std::string op;
};

class BinaryExp : public Exp {
public:
    BinaryExp();
    void setVal(Symbol _val);
    Symbol getVal();
    void setLhs(Exp* node);
    void setRhs(Exp* node);
    virtual void traverse(int lev);
    virtual Type evalType();
    virtual int evalInt();
private:
    Symbol val;
    Exp *lhs, *rhs;
};

class FormatString : public Exp {
public:
    FormatString();
    explicit FormatString(Symbol str);
    Symbol getSym();
    void traverse(int lev) override;
private:
    Symbol sym;
};

class LVal : public UnaryExp {
public:
    LVal();
    explicit LVal(Symbol sym);
    void addDim(Exp *exp);
    Symbol getName();
    void traverse(int lev) override;
```

```
    Type evalType() override;
    int evalInt() override;
private:
    Type type;
    std::vector<Exp*> dims;
    Symbol name;
};

class AssignExp : public BinaryExp {
public:
    AssignExp();
    AssignExp(LVal* l, Exp* r);
    void checkConst();
    void traverse(int lev) override;
    Type evalType() override;
    int evalInt() override;
private:
    LVal* lhs;
    Exp* rhs;
};

class CallExp : public UnaryExp {
public:
    CallExp();
    explicit CallExp(Symbol f);
    void addParam(Exp* param);
    std::vector<Exp*> getParams();
    bool isGetInt() const;
    bool isPrintf() const;
    void checkPrintf();
    void traverse(int lev) override;
    Symbol getFunc();
    Type evalType() override;
    int evalInt() override;
private:
    Symbol func;
    std::vector<Exp*> rParams;
};

class Program {
public:
    Program();
    void addError(ErrorHandling* error);
    void addItem(ProgramItem* item);
    std::vector<ProgramItem*> getItems();
    void traverse(int lev);
    ErrorHandling* err;
private:
    std::vector<ProgramItem*> program_items;
};
```

由于这次作业需要将报错的行号报出来，因此在 `Symbol` 类加入 `col` 和 `row`

```
class Symbol {
public:
    Symbol();
    Symbol(SYMBOL symbol, std::string value, int line, int column);
    void print();
    bool operator==(const Symbol& t) const;
    bool operator!=(const Symbol& t) const;

    SYMBOL sym;
    std::string val;
    int row, col;
private:
};
```

因为发现判断数据类型时需要同时考虑类型和维度，因此这里再定义了 `Type` 类存储数据类型和维度，并定义了 `Type` 类的等价条件。

```
class Type {
public:
    Type();
    explicit Type(Symbol type);
    void addDim(int dim);
    bool operator==(const Type& t);
    bool operator!=(const Type& t);
    Symbol getType() const;
    std::vector<int> getDims() const;
private:
    Symbol type;
    std::vector<int> dims;
};
```

debug 经历

第一次写错误处理发现了很多 bug ，这里就说明一些比较重要的 bug

1. 检查 `break` 和 `continue` 是否在循环语句内

   debug 前是在 `while` 语句连接的 `Block` 内检查 `break` 和 `continue` ，但这个做法无法判断 `Block` 内的 `Block` 存在着 `break` 或 `continue` ，但因这个 `Block` 未被标记为 `while` 连接的，因此会照成这个 `Block` 内的 `LoopStmt` 会被判定为 `m` 错误。

   因此转换成从函数开始检查，分为三个情况：

   - 遇上 `Block` 则进入到 `Block` 内检测，重复该动作
   - 遇上 `while` 语句则直接跳过
   - 遇上 `break` 或 `continue` 时显然就可以发现这个语句是不在 `while` 语句里的，因此判别为 `m` 错误
2. 实参类型判别

   误解了实参维度判断，若实参为 `a[2][2]` 应判为整数，而不是二维数组，其他情况也以此类推。因此寻找出 `a` 的变量定义后，再获取它的维度讯息并根据每个 `[]` 减少实参的维度。

3. `return` 语句检查的是最后一行

   直接从 `Block` 中取出最后一个终结符，但殊不知虽然最后一个终结符基本上不是 `return` ，而是 `;` 。因此把判断改为从尾端开始网上扫描是否存在 `return` 语句。

2021代码生成阶段设计文档

编码前设计

编译器中间代码设计

变量

- 局部变量

```
int main() {
    int a, b = 0;
}
```

```
int %0
int %1 = 0
```

- 全局变量

```
int a, b = 0;
int main() {
    ...
}
```

```
int @1
int @2 = 0
```

表达式

```
z = a*(b + c)
```

```
%2 = %0 + %1
%4 = %3 * %2
```

函数

- 函数声明

```
int foo(int a, int b) { ... }
```

```
int foo()
label:
para int %0
para int %1
...
ret %2
```

- 函数调用（`printf` 和 `getint` 都归类为函数调用）

```
i = tar(x, y);
x = getint();
printf("%d", x);
```

```
push %0
push %1
call tar
i = RET
call getint
%2 = RET
%1 = %2
push %1
call printf
```

- 函数返回

```
return x + y;
```

```
%2 = %0 + %1
ret %2
```

变量和常量

- 局部变量

```
int main() {
    int a, b = 0;
}
```

```
int %0
int %1 = 0
```

- 全局变量

```
int a, b = 0;
int main() {
    ...
}
```

```
int @1
int @2 = 0
```

- 数组声明

  格式为 `alloca <type> (%|@)x <size>`

```
int a[3] = {1,2,3};
int b[3];
```

```
alloca int %0 3
%0[0] = 1
%0[1] = 2
%0[2] = 3
alloca int %1 3
```

- 常量声明

```
const int c = 10; // 常数不做声明
const int arr[3] = {1,2,3};
```

```
alloca const int %0 3
```

分支和跳转

- 标签

```
label:
    %0 = %1 + %2
```

- 条件分值

```
if (a == 1 && b <= 2) { ... }
```

```
cmp a, 1
bne end_if
cmp b, 2
bgt end_if
# if body ...
end_if:
```

- 跳转

```
while ( ... ) {
    if ( ... ) {
        break;
    }
}
```

```
loop_begin:
# some statements ...
goto loop_end
# some statements ...
loop_end:
```

数组

- 数组定义

  格式为 `alloca <type> (%|@)<编号> <size>`

```
int a[3] = {1,2,3};
int b[3];
```

```
alloca int %0 3
%0[0] = 1
%0[1] = 2
%0[2] = 3
alloca int %1 3
```

- 数组读取

```
int a[3][2];
int x = a[2][1];
```

```
la %7, %0
%8 = 2 * 2
%8 = %8 + 1
%9 = %8 * 4
%9 = %7 + %9
load %10, %9
int %6 = %10
ret 0
```

- 数组存储

```
int a[3][2];
int x = 0;
a[2][1] = x; // a[3][2]
```

```
int %6 = 0
la %7, %0
%8 = 2 * 2
%8 = %8 + 1
%9 = %8 * 4
%9 = %7 + %9
load %10, %9
%10 = %6
store %10, %9
```

控制语句

- `while` 语句

```
while (condition) stmt
```

```
label_1:
    // while condition code
    Beqz %1, label_3
label_2:
    // while statement code
    Jump label_1
label_3:
    // while end code
```

- `if` 语句

```
if (cond1) stmt1;
if (cond2) stmt2;
else stmt3;
```

```
label_1:
    // if condition code
    Beqz %1, label_2
label_2:
    // if end code (empty)
label_3:
    // if condition code
    Beqz %2, label_5
label_4:
    // if statement code
label_5:
    // else statement code
label_6:
    // if end code
```

条件语句

- `&&`

```
cond1 && cond2
```

```
// cond1 code
Beqz %1, end_label
// cond2 code
Beqz %2, end_label
```

- `||`

```
cond1 || cond2
```

```
// cond1 code
Beqz %1, body_label
// cond2 code
Beqz %2, body_label
```

编码后设计

中间代码生成

中间代码设计分为三大层次，`IrFunc`、`BasicBlock`、`Inst`，并使用数据结构侵入式链表存储。

`IrFunc` 设计

```
class IrFunc {
    int varId; // 局部变量 id
    std::string funcName; // 函数名
    std::vector<IrParam*> params; // 参数
    IntrusiveLinkedList<BasicBlock> blocks; // 基本块链表
    Type returnType; // 返回类型

    std::map<Decl*, Variable*> table; // 函数符号表
};
```

`BasicBlock` 设计

```
class BasicBlock {
    BasicBlock *prev, *next; // 链表指针

    int label_id;
    IntrusiveLinkedList<Inst> insts; // 中间指令
    std::vector<BasicBlock*> pred, succ; // 基本块的前驱和后继
};
```

`Inst` 设计

```
class Inst {
    Inst *prev, *next; // 链表指针
};
```

实现时把中间代码都分类成以下的中间指令，并都继承父类 `Inst`

- `BinaryInst`

  ```
  class BinaryInst : public Inst {
      BinaryOp op;
      Variable *var, *lhs, *rhs;
  };
  ```

- `AssignInst`

  ```
  class AssignInst : public Inst {
      Variable *lhs, *rhs;
  };
  ```

- `ReturnInst`

```
class ReturnInst : public Inst {
    Variable* var;
};
```

- `CallInst`

```
class CallInst : public Inst {
    Symbol sym;
    IrFunc* func;
    std::vector<Variable*> params;
};
```

- `DeclInst`

```
class DeclInst : public Inst {
    Variable *var, *init;
    std::vector<Variable*> inits;
};
```

- `GetReturnInst`

```
class GetReturnInst : public Inst {
    Variable* var; // 接收变量
};
```

- `BranchInst`

```
class BranchInst : public Inst {
    BranchOp op;
    Variable* var;
    int label_id;
};
```

- `JumpInst`

```
class JumpInst : public Inst {
    int label_id;
};
```

- `NotInst`

```
class NotInst : public Inst {
    Variable *var, *not_var;
};
```

- `LoadAddrInst`

```
class LoadAddrInst : public Inst {
    Variable *var, *base;
};
```

- `LoadInst`

```
class LoadInst : public Inst {
public:
    Variable *dst, *addr;
};
```

- `StoreInst`

```
class StoreInst : public Inst {
public:
    Variable *addr, *val;
    StoreInst();
    StoreInst(Variable* val, Variable* addr);

    std::string show() override;
};
```

中间代码中的数据表示都为 `Variable` ，以下为 `Variable` 的设计

`Variable` 设计

```
class Variable {
    int id; // id
    Type bType; // 变量数据类型
    bool is_global, is_addr; // 表示该 Variable 是全局和数组地址
    std::vector<Variable*> dims; // 变量维度
};
```

`Variable` 也细分成以下的类，都继承 `Variable` 父类

- `Constant`

  `Constant` 可以是常数或常量字符串，并由 `type` 决定

```
class Constant : public Variable {
    int value; // 数值
    SYMBOL type; // 数据类型
    std::string str; // 字符串
};
```

- `IrParam`

  实参类型

```
class IrParam : public Variable {
    Constant* constant;
    Variable* var;
};
```

- `IrArray`

```
class IrArray : public Variable {
    bool isConst; // 是否为常量
    int base, size; // 基地址和数组大小
};
```

- `IrPointer`

  `IrPointer` 主要用在数组传参上

```
class IrPointer : public IrParam {
public:
    bool isConst; // 是否为常量
    int base; // 地址
};
```

目标代码生成

代码生成作业实现的是生成 `Mips` 汇编代码，目标代码生成主要在 `Generator` 上实现，根据中间指令，生成对应的汇编代码。

- `BinaryInst`

  该指令有四个情况：

  - 若 `lhs` 和 `rhs` 都为常量

    直接在编译器里计算

  - 若 `lhs` 为常量

    若该运算符满足交换律，则将 `lhs` 作为立即数计算，否则需 `li` 到寄存器上

  - 若 `rhs` 为常量

    将 `rhs` 作为立即数计算

  - 若都不为常量

    将 `lhs` 和 `rhs` 从内存拿到寄存器中，并根据 `op` 的类型进行计算。

- `AssignInst`

  该指令有两个情况

  - 若 `rhs` 为常量

    直接 `li` 到 `lhs`

  - 若 `rhs` 不为常量

    从内存 `lw` 并赋值到 `lhs`

- `ReturnInst`

  若有返回值将 `var` 传到 `v0` 里，再 `jr $ra`

- `CallInst`

  该指令分为三个情况

  - `getint` 函数

    将 `v0` 设为 5 并执行 `syscall` 。

  - `printf` 函数

根据参数类型设置 `v0` 并执行 `syscall` 输出，参数可以是字符串常量及普通变量。

- 其他函数

  将实参填入到该函数定义的位置，再将当前 `$ra` 寄存器存入栈内，并腾出被调用函数的空间，最后跳转到函数的第一个 `BasicBlock`

- `DeclInst`

  若有初始化，则将对应的值存入该变量的内存。

- `GetReturnInst`

  将 `v0` 的值存入该变量的内存。

- `BranchInst`

  将变量从内存取出到寄存器，并根据 op 决定跳转条件跳转到制定的 `label`。

- `JumpInst`

  直接跳转到制定 `label` 。

- `NotInst`

  检测 `Variable` 是否为 0 即可。

- `LoadAddrInst`

  将 `base` 的地址放入 `var` 内。

- `LoadInst`

  将 `addr` 地址的值放入 `dst` 内。

- `StoreInst`

  将 `val` 的值放入 `addr` 地址的内存里。

debug 经历

一开始实现的 bug 很多，只叙述几个重要的 bug

- break continue

  在 while 代码的设计里分为三个 label，即 while condition, while body 和 while end，一开始认为执行完 while condition 后就不会再使用这三个 label，因此 break 的时候找不到 label 而导致程序崩溃。因此这些 label 会用栈进行维护，每进入一个 while 程序就会将这三个 label 压栈，退出时弹栈。

- 指针取地址

  数组取地址时使用的是 `la` ，但因为指针存储的是数组的地址，若直接 `la` 会取出指针的地址，而不是数组的基地址。

- NotInst

  当执行 `!x` 时，需要一个临时变量保存结果，而不是直接将 `x` 设为 `!x` 。