

编译实验课程设计申优文档

19231133 王雯清

写在前面

本文档对于编译器的每个部分的实现进行了说明，大体上分为数据结构、算法思路和难点分析三个部分。

目录

- [写在前面](#)
- [目录](#)
- [词法分析](#)
 - [数据结构](#)
 - [算法流程](#)
 - [难点分析](#)
- [语法分析](#)
 - [数据结构](#)
 - [ASTNode](#)
 - [算法流程](#)
 - [难点分析](#)
- [符号表生成](#)
 - [数据结构](#)
 - [Symbol](#)
 - [SymbolTable](#)
 - [算法流程](#)
 - [难点分析](#)
- [函数表生成](#)
 - [数据结构](#)
 - [Function](#)
 - [FunctionTable](#)
 - [算法流程](#)
 - [难点分析](#)
- [错误处理](#)
 - [数据结构](#)
 - [Error](#)
 - [算法流程](#)
 - [难点分析](#)
- [中间代码生成](#)
 - [中间代码语法简述](#)
 - [类型](#)
 - [语句](#)
 - [mem2reg 与 phi](#)
 - [数据结构](#)
 - [Value](#)
 - [LLVMIR](#)
 - [BasicBlock](#)
 - [Edge](#)
 - [算法流程](#)
 - 0. 判断代码是否存在错误
 - 1. 含有 ALLOCA, LOAD, STORE 的中间代码
 - 2. mem2reg

- a. 划分基本块
- b. 建立深度优先搜索树
- c. 无用基本块删除
- d. 计算半必经点

 前置概念 (参考链接1, 参考链接2)

 算法流程

- e. 计算直接支配点
- f. 计算支配边界
- g. 插入 phi 节点
- h. 变量重命名

 难点分析

中间代码优化

死代码删除

- 一般代码删除
- 跳转代码删除

常量折叠

函数内联

- 1. 判断函数是否内联
- 2. 函数内联实现

 返回语句

 返回值

循环优化

- 1. while -> do-while

 删除无用循环

 循环展开

局部数组提升 & 标记常量数组

全局转局部

目标代码生成

数据结构

- RegisterType
- MIPSCode
- Register

算法流程

难点分析

 函数调用

 栈指针偏移

 phi 函数的实现

目标代码优化

 寄存器分配

 乘除优化

 窥孔优化

 无用跳转指令删除

 无用 move 指令删除

结语

参考文献

词法分析

数据结构

Token

建立 Token 类, 存储每个 Token 所在的行号 (lineNum), 包含的内容 (content) 和 Token 的类别 (TokenType)。

```
enum TokenType {
    IDENFR, INTCON, STRCON, MAINTK, CONSTTK, INTTK, BREAKTK,
    CONTINUETK, IFTK, ELSETK, NOT, AND, OR, WHILETK, VOIDTK,
    GETINTTK, PRINTFTK, RETURNTK, PLUS, MINU, MULT, DIV, MOD,
    LSS, LEQ, GRE, GEQ, EQL, NEQ, ASSIGN, ERROR,
    SEMICN, COMMA, LPARENT, RPARENT, LBRACK, RBRACK, LBRACE, RBRACE
};

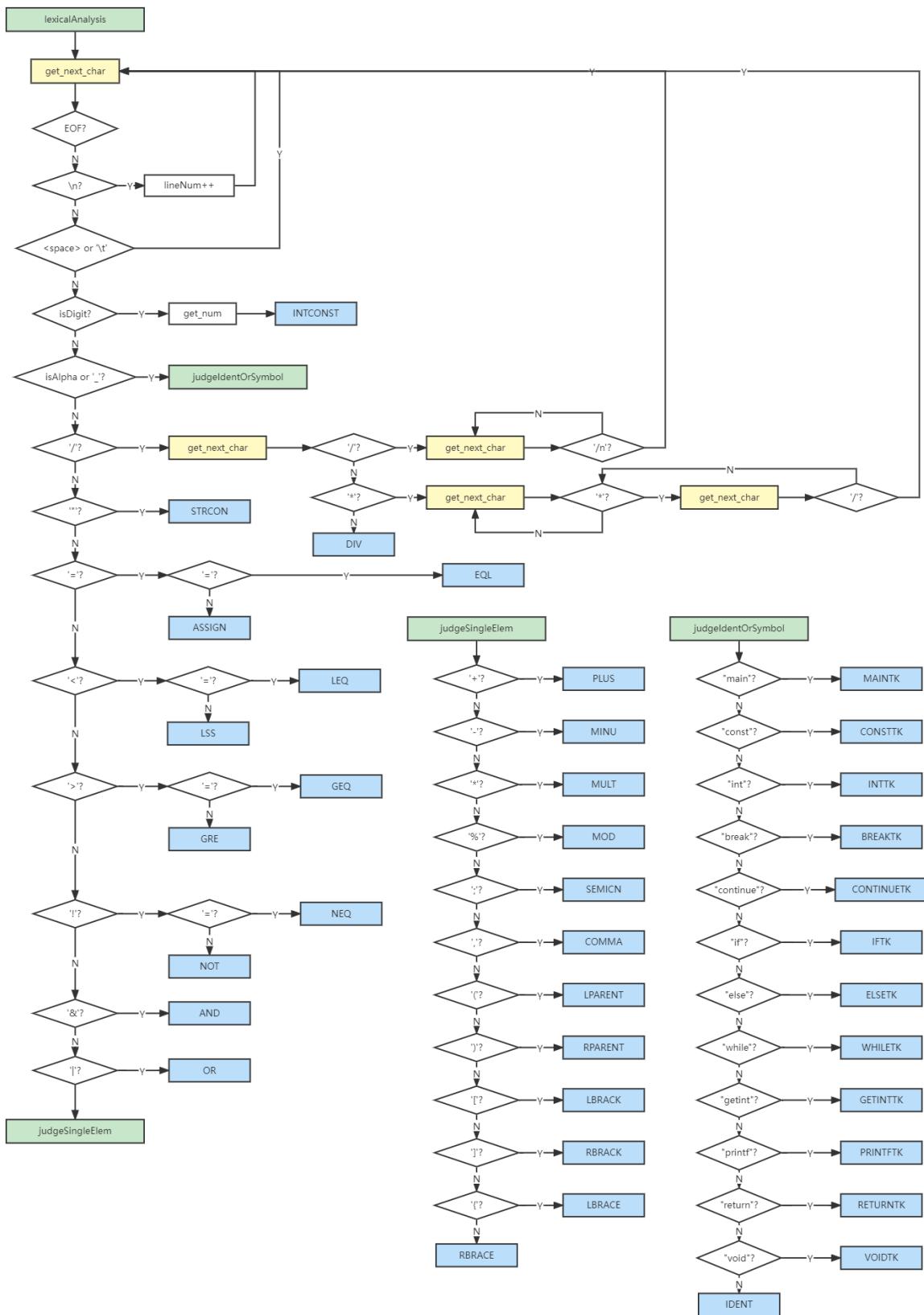
class Token {
public:
    int lineNumber = 0; // 行号
    string content; // 内容
    string classId; // 类别
    TokenType tokenType = ERROR;

    Token() = default; // 默认构造

    Token(int l, string c, string id, TokenType type) :
        lineNumber(l), classId(std::move(id)), content(std::move(c)),
        tokenType(type) {}
};
```

算法流程

以字符为单位遍历输入程序，根据下图所示的流程图，将输入字符流逐一转化为相应 Token



难点分析

词法分析的正确实现需要细致，这体现在以下几个方面：

- 不漏读，不复读字符：
 - 在判断当前字符所对应的 Token 类型时，若不符合当前类型，应继续分析可能的下一种 Token 类型，而非将指针向后移动，从而跳过当前字符；也不能在当前字符已符合某种 Token 类型时继续判断当前字符类型，造成字符重复使用。
- Token 前缀相同时的判断

- 如 `<=` 与 `<`, `>=` 与 `>`, `!=` 与 `!`, `==` 与 `=` 等前缀相同的情况，需要根据后一个字符判断 Token 类型

语法分析

数据结构

ASTNode

建立 AST 结点类 `ASTNode`，所有语法成分将继承自此节点。

```
class ASTNode {
public:
    virtual ~ASTNode() = default;

    ASTNode() = default; // 默认构造

    virtual void print() {} // 输出形式
};
```

对文法中的每个终结符和非终结符建类，所有类均继承自 `ASTNode`，类内含有该类可推导出的语法成分的指针。

例：

```
class Exp : public ASTNode { // 表达式
public:
    AddExp *addExp;

    Exp() : ASTNode() {}

    explicit Exp(AddExp *ae) : addExp(ae) {}

    void print() override {
        cout << "<Exp>" << endl;
    }
};
```

算法流程

1. 消除文法中的左递归

e.g.

原文法： `AddExp := MulExp | AddExp ('+' | '-') MulExp`

消除左递归后的文法： `AddExp := MulExp | MulExp ('+' | '-') AddExp`

2. 采用 LL(n) 的方式进行语法分析，避免回溯

3. 构建 AST (Abstract Syntax Tree, 抽象语法树)

采用递归的方式对每个非终结符构建 `parse` 函数。

例：

```

Exp* parseExp() { // 解析 Exp 类
    auto *exp = new Exp(parseAddExp()); // 调用 addExp 调用程序 (返回 *
    addExp)
    ASTTree.push_back(exp); // 将 expression 结点加入树中
    return exp;
}

```

难点分析

- 消除左递归：
 - 由于原文法中存在左递归的情况，因此为了避免在语法分析中出现循环的情况，需要消除原文法中的左递归。但由于输出为根据原文法生成的语法树的后缀形式，因此在消除左递归后应适当变换，使输出满足原文法要求。
- 指针的使用
 - 在设计时考虑在每个类中存储对象本身而非对象的指针，但在实现过程中，由于各个类之间存在复杂的调用关系（包括环路），因此可能需要提前使用未定义的类。因此于其在类中用对象的形式存储属性，用指针更加节约空间且灵活，除此之外，还能避免拷贝构造时发生浅拷贝等问题。
- 利用堆空间而非栈空间存储对象
 - 在设计时，设想直接在 parse 函数中创建各个类的对象，但由于复杂的程序往往涉及到多层次深入的调用关系，而函数内部声明的对象也有其生存周期，因此发生了程序在面对长输入程序时出现信息丢失和爆栈的情况。
 - 解决方案是用 `new` 声明对象，将各个类的对象存放在堆空间中，加长对象的生存周期，避免爆栈和信息缺失的问题。

符号表生成

数据结构

Symbol

建立符号类 `Symbol` 记录符号的名字，地址，维度，声明行号和使用行号等信息。

```

enum VarType {
    DEFAULT, INTCONST, INTVAR
};

class Symbol {
public:
    string name;
    int address = 0;
    int dimension = 0;
    int row = 0;
    int column = 0;
    VarType varType = DEFAULT; // const?
    int declareLine = 0; // 声明行号
    list<int> usedLine; // 使用行号

    Symbol() = default;

    Symbol(const Token& token, VarType vt, int dim) : name(token.content),
    varType(vt), dimension(dim),
    declareLine(token.lineNum) {}
}

```

```

void use(int lineNumber) {
    usedLine.push_back(lineNumber);
}
};

```

SymbolTable

建立符号表类，用 map 的方式存储符号集 Symbol，其中，键为 Symbol 的名称，值为 Symbol 的指针

```

class SymbolTable {
public:
    map<string, Symbol *> symbols; // symbol 符号集
    SymbolTable *father = nullptr // 指向父符号表的指针
    list<SymbolTable *> sons; // 指向子符号表的指针集

    SymbolTable() = default; // 默认构造

    int insert(Symbol *sym) { // 向符号表中插入 Symbol* sym
        /* return 1 on success, return 0 on failure */
        string sym_name = sym->name;
        map<string, Symbol *>::iterator iter;
        iter = symbols.find(sym_name);
        if (iter == symbols.end()) { // not find
            symbols[sym_name] = sym;
            return 1;
        } else return 0;
    }

    Symbol *find(const string &sym_name) { // 在符号表及其父符号表中查找名为 sym_name 的 Symbol
        map<string, Symbol *>::iterator iter;
        iter = symbols.find(sym_name);
        if (iter == symbols.end()) { // not find
            if (father == nullptr) {
                auto *symbol = new Symbol();
                return symbol;
            } else {
                return father->find(sym_name);
            }
        } else {
            return iter->second;
        }
    }

    SymbolTable *newSon() { // 建立子符号表并返回其指针
        auto *symbolTable = new SymbolTable();
        symbolTable->father = this;
        this->sons.push_back(symbolTable);
        return symbolTable;
    }

    SymbolTable *newBrother() { // 建立兄弟符号表返回其指针
        auto *symbolTable = new SymbolTable();
        symbolTable->father = this->father;
        if (father != nullptr) {
            father->sons.push_back(symbolTable);
        }
    }
};

```

```

    }
    return symbolTable;
}

SymbolTable *back() const { // 返回父符号表的指针
    return father;
}
};

```

算法流程

建立根符号表，并将当前符号表的指针指向根符号表。

```

SymbolTable *symbolTable = new SymbolTable();
SymbolTable *currentSymbolTable = symbolTable;

```

每当进入一个新的 Block 或新的函数时，建立新的子符号表，并将当前符号表的指针 `currentSymbolTable` 指向新的子符号表。

```

block* parseBlock() {
    currentSymbolTable = currentSymbolTable->newSon();
    // ...
    currentSymbolTable = currentSymbolTable->back();
    return b;
}

```

当遇到声明 (`Dec1`) 语句和函数形参声明时，向符号表中填入 `Symbol` 并检查是否存在重名问题

在 `stmt` 中使用符号时，由当前符号表逐层向上检索所用符号名和符号类型，检查是否存在未定义和改变 `const` 值等问题。

难点分析

符号表实现中的困难主要来自于对变量生存周期和作用域的理解。

在函数中，形参和函数体应共享一个符号表，而不能因为函数体为 `Block` 而创建只属于函数体的符号表。因此在进入 `parseBlock` 时，应首先判断是否在函数体内，若是，则不创建新符号表，否则创建新的子符号表，并将当前符号表的指针指向该符号表。

函数表生成

数据结构

Function

建立 `Function` 类，存储函数名，形参和返回值等信息

```

enum ReturnType {
    VOID, INT
};

class Function {
public:
    string name; // 函数名
    ReturnType returnType = VOID; // 函数返回值类型
}

```

```

list<Symbol *> parameters;

Function() = default;

Function(string n, ReturnType rt, list<Symbol *> p) : name(std::move(n)),
returnType(rt), parameters(std::move(p)) {}

int paramsMatch(list<Symbol *> params) { // 判断函数参数个数，类别是否吻合
    /* return: 0 -- success, -1 -- number doesn't fit, -2 -- type doesn't
     * fit */
    if (parameters.size() != params.size()) return -1;
    else {
        auto require_iter = parameters.begin();
        auto get_iter = params.begin();
        while (require_iter != parameters.end()) {
            if (!(*require_iter)->varType == (*get_iter)->varType &&
                (*require_iter)->dimension == (*get_iter)->dimension)
                return -2;
            ++require_iter;
            ++get_iter;
        }
    }
    return 0;
}

```

FunctionTable

建立函数表 `FunctionTable`，用于存储全部声明函数的函数名，返回值等信息。

```

class FunctionTable {
public:
    map<string, Function *> functions; // 用 map 存储全部声明函数

    FunctionTable() = default; // 默认构造

    int insert(Function *func) {
        /* return 1 on success, return 0 on failure */
        string func_name = func->name;
        map<string, Function *>::iterator iter;
        iter = functions.find(func_name);
        if (iter == functions.end()) { // not find
            functions[func_name] = func;
            return 1;
        } else return 0;
    }

    Function *find(const string &func_name) { // 在函数表中寻找名为 func_name 的函数
        map<string, Function *>::iterator iter;
        iter = functions.find(func_name);
        if (iter == functions.end()) { // not find
            auto *function = new Function();
            return function;
        } else {
            return iter->second;
        }
    }
}

```

```
};
```

算法流程

建立全局函数表 functionTable，每声明一个新的函数时向其中加入 Function 并判断函数是否存在重名等问题。

```
FunctionTable *functionTable = new FunctionTable();
```

每次用到函数时在函数表中根据函数名查找函数，判断是否存在未声明的问题。若函数已经声明，则进一步判断参数个数与类型是否与声明中相吻合。

难点分析

函数表中不仅需要记录函数名和返回值等信息，还需记录函数参数个数和类型等信息。

错误处理

数据结构

Error

建立 Error 类，存储错误行号和错误类型

```
class Error {
public:
    int lineNumber = 0; // 错误行号
    string errorType; // 错误类型

    Error() = default; // 默认构造

    Error(int ln, string et) : lineNumber(ln), errorType(std::move(et)) {}

    friend std::ostream &operator<<(std::ostream &out, Error &t) {
        out << t.lineNumber << " " << t.errorType;
        return out;
    }
};
```

算法流程

建立全局错误队列，存储所有程序中出现的错误，由于错误处理的输出需要有序，因此在输出前统一对错误进行排序，再逐一输出。

```
vector<Error*> errors;
```

错误处理办法：

- 分析每个错误所有可能出现的位置
- 在语法分析建立 AST 的同时对生成符号表、函数表并对错误进行判断。
- 每判断出一个错误将错误的行号和类型码存储在 Error 类中，并将其加入 errors

错误类别码	错误类型	判断方法
a	非法符号	当 Token 为 STRCONST 时，对格式字符串的合法性进行判断。
b	名字重定义	建立符号表与函数表，每定义新的符号 / 函数时向表中添加，当前符号 / 函数名在当前作用域内已定义，则产生名字重定义问题。
c	未定义的名字	在使用变量和函数时在符号表和函数表中查询，若无法查询到，则证明出现未定义的名字。
d	函数参数个数不匹配	在函数声明时在 Function 类中存储函数的形参信息（类别、数量、顺序），在调用函数时记录传递参数的信息，并于 Function 中的参数个数相对比，若不同，则抛出函数参数个数不匹配的错误
e	函数参数类型不匹配	函数声明时在 Function 类中存储函数的形参信息（类别、数量、顺序），在调用函数时记录传递参数的信息，并于 Function 中的参数类型（类型、维度）相对比，若不同，则抛出函数参数个数不匹配的错误
f	无返回值的函数存在不匹配的 <code>return</code>	在函数声明时，若函数的 returnType 为 VOIDTK，则获取函数块内最后一条语句，若其为 return 语句且不为 "return;"，则报错
g	有返回值的函数缺少 <code>return</code> 语句	在函数声明时，若函数的 returnType 为 INTTK，则获取函数块内最后一条语句，若其不为 return 语句，则报错
h	不能改变常量的值	在变量声明并加入符号表时存储其是否为 const，在对变量赋值时，若变量存在且为 INTCONST，则报错
i	缺少分号	在 parse 函数遇到 <code>auto *semicn = new Semicn(*iter++);</code> 时将其替换为先检查当前 iter 的 Token 类是否为 SEMICN，若是，则重复之前的操作，若不是，则自动补全分号并抛出 i 型错误
j	缺少右小括号	在 parse 函数遇到 <code>auto *rparent = new Rparent(*iter++);</code> 时将其替换为先检查当前 iter 的 Token 类是否为 RPARENT，若是，则重复之前的操作，若不是，则自动补全右小括号并抛出 j 型错误
k	缺少右中括号	在 parse 函数遇到 <code>auto *rbrack = new Rbrack(*iter++);</code> 时将其替换为先检查当前 iter 的 Token 类是否为 RBRACK，若是，则重复之前的操作，若不是，则自动补全右中括号并抛出 k 型错误
l	<code>printf</code> 中格式字符串与表达式个数不匹配	在语法分析的过程中，在遇到 Stmt -> printf '(' FormatString {, Exp} ')' 时记录 Exp 的数量和 FormatString 中 '%d' 的数量，将两者对比，若不同，则抛出 l 型错误
m	在非循环块中使用 <code>break</code> 和 <code>continue</code> 语句	设立 recursive 变量，用于记录所在的循环层数，当循环层数为 0 (不处于任何循环当中时)，若出现 <code>break</code> 和 <code>continue</code> 则抛出 m 类型错误。

难点分析

错误处理考察实现细节，需要对每种错误可能出现的情况进行详细的讨论，并适当修改语法分析代码。

另外，对于“缺失”类型的错误，需要注意不让程序进入死循环。

例：在语法分析中，读取函数形参 `parseFuncFParams` 在读入右小括号时退出

```
while (...) {
    if ((*iter).tokenType == LBRACK) {
        break;
    }
}
```

但需要考虑右小括号缺失的情况，此时应添加条件判断，即：若读入 `{`（代表进入函数体），也视作函数形参输入完成，且抛出右小括号缺失类型错误。

中间代码生成

中间代码采用 SSA（静态单一赋值）的形式，采用类似 LLVMIR 的语言，便于后续优化。

在静态单一赋值的代码中，每个变量仅被赋值一次。生成 SSA 形式的中间代码极大的简化了优化的实现和优化程序的效率，但缺点在于构建 SSA 形式的代码难度较大。

中间代码语法简述

由于中间代码并不需要真正被 LLVM 编译器编译通过，因此在实现上以方便书写且满足 SSA 规范为准，在具体语法上可能与标准的 LLVM 有所差异。

类型

中间代码中需要对变量的类型进行标识，为了使得优化程序更好的对于各个不同的变量类型执行相应的优化操作，因此选择保留尽量多的信息。如：源程序中的 `int` 在中间代码中可以对应全局变量和局部变量。

中间代码中所使用的变量 / 常量类型如下：

- 临时变量 `TEMP`

在定义后使用一次即消亡的变量，不对应程序中的变量，用于在计算时保存临时结果。

- 全局变量 `INTGLOBAL`

对应程序中全局声明的 `int` 类型。

- 全局常量 `INTGLOBALCONST`

对应程序中全局声明的 `const int` 类型。

- 局部变量 `INTLOCAL`

对应程序中局部声明的 `int` 类型。

- 局部常量 `INTLOCALCONST`

对应程序中局部声明的 `const int` 类型。

- 指针变量 `POINTER`

对应程序中声明的局部数组类型。

- 数字 `NUMBER`

对应程序中的立即数。

- 字符串 `FORMATSTRING`

对应程序中 `printf` 中的字符串。

- 函数名称 `FUNCTIONNAME`

对应程序中的函数名。

- label `LABEL`

用于表示基本块的开始和便于基本块之间跳转语句的正常实现。

为了能够更加直观的在输出到文件中的中间代码中辨识各个变量类型，对每种类型均定义了不同的输出方式，如下：

变量类型	输出方式	举例
<code>TEMP</code>	<code>i32 \${id}</code>	<code>i32 \$1</code>
<code>INTGLOBAL</code> or <code>INTGLOBALCONST</code>	<code>i32 @{id}</code> or <code>[{size} * i32]* @{id}</code>	<code>i32 @0, [3 * i32] @4</code>
<code>INTLOCAL</code> or <code>INTLOCALCONST</code>	<code>i32 %{id}</code>	<code>i32 %5</code>
<code>POINTER</code>	<code>i32* %{id}</code> or <code>[size * i32]* %{id}</code>	<code>i32* %1 or [2 * i32]* %5</code>
<code>LABEL</code>	<code>Label{id}</code> or <code>main</code>	<code>Label1</code> or <code>main</code>
<code>FUNCTIONNAME</code>	<code>@{name}</code>	<code>@func</code>
<code>NUMBER</code>	<code>i32 {number}</code>	<code>i32 512</code>

语句

中间代码的语句主要分为一下几种：

- 声明语句

- 算数语句：实现基本的加减乘除等操作，变量类型可以为 `TEMP` 或 `NUMBER`

- 双目运算
 - 算数运算 (`ADDR`, `SUBIR`, `MULIR`, `DIVIR`, `MODIR`)
 - 逻辑运算 (`ANDIR`, `ORIR`, `EQIR`, `NEQIR`, `GREIR`, `GEOIR`, `LSSIR`, `LEQIR`)
 - 单目运算
 - 算数运算 (`ADDSINGLEIR`, `SUBSINGLEIR`)
 - 逻辑运算 (`NOTIR`)

- 跳转语句

- 无条件跳转 (`BR`)
 - 有条件跳转 (`BRTRUE`, `BRFALSE`)
 - 跳转并移动栈指针 (由于存在 `continue` 和 `break` 语句，在跳转时可能出现跳转前后栈指针不一致的情况，因此这种指令需要在跳转后将栈指针移动至正确的位置) `BRCHECK`

- 函数相关语句

- 函数定义 (`FUNCDCLIR`)
 - 函数调用 (`CALL`)
 - 函数返回 (`RET`)

- 存取语句 (`LOAD`, `STORE`, `GETPOINTER`)，其中，`GETPOINTER` 可获得数组相应位置的地址
 - `mem2reg` 相关语句 (`MEM2REGASSIGN`, `MEM2REGMOVE`)
 - 其它语句（用于标识程序中 `Block` 的开始和结尾，便于栈指针移动至正确位置）
 - `LBRACEIR`, `RBRACEIR`, `LSHARPIR`, `RSHARPIR`

为了能更清楚的在输出后的中间代码中辨识语句类型，对每种语句定义了相应的输出操作。

mem2reg 与 phi

静态单一赋值要求同一个变量只能被赋值一次，但遇到下图所示的情况，应当如何确定应该输出哪一个 i 的值呢？

一个最简单的实现办法就是利用 LOAD 和 STORE 语句，每次直接从内存中读写，而不需要考虑具体应当取哪一种赋值（如下图中左边代码所示），但这样做显然效率不高，大量的存取指令会占用大量的程序运行时间。

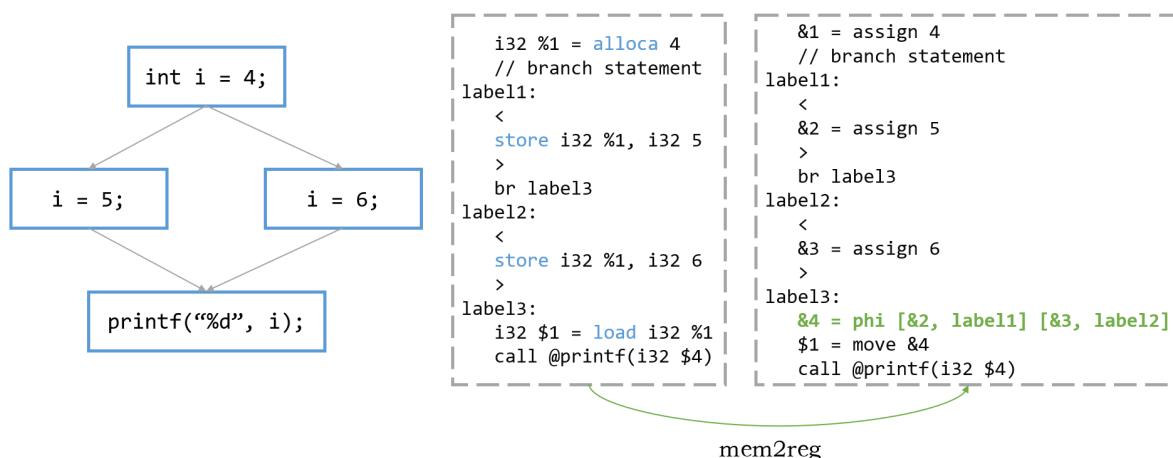
为了能够提高程序的运行效率，需要消解 LOAD 与 STORE 语句，将其变为 SSA 形式的赋值语句，而实现从 LOAD, STORE 到 SSA 的转化的步骤就被称作 mem2reg。为了使程序能够在具有分支的情况下依然保持正确性，引入 phi 指令。

phi 指令的语法是

```
<result> = phi <ty> [<val0>, <label0>], [<val1>, <label1>] ...
```

phi 指令被安排在每个基本块的最开头，且执行没有先后顺序。phi 指令中的 label 对应着基本块的前驱，而 val 则是 result 可能的取值。

举例来说，下图中右面的代码就是插入了 phi 的代码。其中，语句 `&4 = phi [&2, label1] [&3, label2]` 代表若从 label1 跳转至该基本块，则 `&4` 取 `&2` 的值，否则若从 label2 跳转至该基本块，`&4` 取 `&3` 的值。



数据结构

value

在 SSA 形式的中间代码中，包括变量、常量、label 等一切皆是 `value`，因此建立 `value` 类以及其对应的 `VarType` 枚举类。

```
enum VarType {  
    DEFAULT, INTCONST, INTVAR, VOIDVAR,  
    INTGLOBAL, INTGLOBALCONST, INTLOCAL, INTLOCALCONST,  
    LABEL, POINTER, NUMBER, FUNCTIONNAME, FORMATSTRING};
```

```

    TEMP, STRING, PMET
};

class value {
public:
    int id; // 独一无二的 id, 若为 NUMBER 则 id 为其对应的值
    VarType varType = DEFAULT; // value 类型
    int symbolTableId = -1; // 对应的符号表
    string name; // 对应的函数名或变量名
    int size = 1; // 变量大小
    int declLine; // 定义变量的行号
    vector<int> useLine; // 使用变量的行号
};

```

LLVMIR

建立中间代码类 `LLVMIR` 以及其对应的枚举类 `IRType`，用于表示各类中间代码，它的构造函数接受不同数量的 `Value` 和 `IRType`，生成各种形式的 IR。

```

enum IRType {
    IRTYPE, // DEFAULT
    ADDIR, SUBIR, MULIR, DIVIR, MODIR, // 加减乘除
    ANDIR, ORIR, EQIR, NEQIR, // 与或非、等于、不等于
    NOTIR, ADDSINGLEIR, SUBSINGLEIR, // 单目运算
    GREIR, GEQIR, LSSIR, LEQIR, // 比较运算
    BR, RET, CALL, BRTURE, BRFALSE, BRCHECK, // 跳转运算
    FUNCDECLIR, GLOBALDECLIR, LOCALDECLIR, CONSTDECLIR, // 定义（函数、全局、局部、常量）
    LOAD, STORE, ALLOCA, LABELIR, ASSIGNIR, TEMPALLOCA, // 存取运算
    LBRACEIR, RBRACEIR, LSHARPIR, RSHARPIR, // 大括号 / 中括号（用于标记 block 的始末，便于 stack pointer 移动到正确的位置）
    GETPOINTERIR, // 数组取地址
    MEM2REGASSIGN, MEM2REGMOVE, TEMPPHI, PHI // mem2reg 有关指令
};

class LLVMIR {
public:
    IRType irType = IRTYPE; // IR 类型
    int lineNumber; // IR 行号
    int blockLineNumber; // 在 BasicBlock 中的行号（在变量冲突分析时使用）
    // 以下属性均为 LLVMIR 构造时需要用到的参数
    value *dest = nullptr;
    vector<Value *> params;
    Variable *variable;
    vector<Variable *> varParams;
    vector<int> labels;
    int valueId;
};

```

BasicBlock

`BasicBlock` 的含义是“基本块”，用来表示一段连续执行的程序序列。基本块中的代码只能顺序执行，只能从开头进入，从结尾离开。将程序划分为基本块的优点在于便于分析程序的控制流，从而使得优化能够更加方便的完成。

由基本块的特性，定义对应的类及其属性如下：

```

class BasicBlock {
public:
    int id; // 每个 BasicBlock 独有的 id, 对应其开头的 label 值
    int dfnNum = -1; // 深度优先搜索遍历顺序, 在构建控制流图时使用
    set<BasicBlock *> precursor; // 前驱基本块
    set<BasicBlock *> successor; // 后继基本块
    set<BasicBlock *> bucket; // 深度优先搜索树中的孩子节点, 在构建控制流图时使用
    set<BasicBlock *> strictDominator; // 严格支配基本块
    set<BasicBlock *> dominanceFrontier; // 支配边界
    BasicBlock *sdom = this; // 直接支配基本块
    BasicBlock *idom = nullptr; // 半支配基本块
    BasicBlock *dfsFather = nullptr; // 深度优先搜索树中的父亲节点
    set<int> defValue; // 当前基本块中定义的变量
    set<int> useValue; // 当前基本块中使用的变量
    set<int> phis; // 当前基本块开头插入的 phi 指令
    map<int, int> def; // 当前基本块中定义的变量及其定义行号
    map<int, int> use; // 当前基本块中使用的变量及其使用行号
    set<int> defBeforeUse; // 当前基本块中定义先于使用的变量
    set<int> useBeforeDef; // 当前基本快中使用先于定义的变量
    set<int> in; // 进入基本快的数据流
    set<int> out; // 从基本快中出来的数据流
};


```

Edge

为了更方便的得到控制流图, 还需要了解基本块之间的跳转关系, 对应图论中两个节点之间的边, 因此定义边类 Edge。

```

class Edge {
public:
    BasicBlock *source = nullptr; // 边的起始点
    BasicBlock *dest = nullptr; // 边的终点
};


```

算法流程

0. 判断代码是否存在错误

若代码有错误, 则直接退出程序。在需要编译的代码无语法错误时, 进入中间代码生成程序。

1. 含有 ALLOCA, LOAD, STORE 的中间代码

遍历语法树, 对其中的每一个节点建立 CodeGen 函数, 并返回一个 value 值。

例如, 对于 AddExp, 其在文法中的定义为 AddExp → MulExp | AddExp ('+' | '-') MulExp, 由定义可知, AddExp 需要将 MulExp 计算出的结果进行加法或减法操作并返回计算结果。故其生成中间代码的函数为: (只展示主题结构)

```

value* AddExp::CodeGen() {
    auto *result = mulExps[0]->CodeGen();
    size_t length = addSubASTs.size();
    for (size_t i = 0; i < length; ++i) {
        auto *temp_val = mulExps[i + 1]->CodeGen(); // 调用每个 MulExp 的 codeGen
        // 函数, 得到 MulExp 的运算结果。
        // ...
        midCode.push_back(new LLVMIR((opType == PLUS ? ADDIR : SUBIR),
11vmLineNumber++, value, params)); // 根据运算符生成对应的中间代码 (加法 / 减法)
        // ...
    }
    return result;
}

```

将遍历语法树时产生的所有中间代码都保存在全局的 `midcode` 中, 便于之后进行多遍优化。

需要注意的是, 由于中间代码需要保持静态单一赋值的特性, 因此所有对变量的定义均转化为 `ALLOCA` 语句, 意为直接为变量分配一段内存; 所有对变量的取值操作则转化为 `LOAD` 语句, 意为从内存中读相应值; 而所有对变量的赋值操作则转化为 `STORE`, 意为向内存中存入相应值。

特别的, 对于数组, 用 `GETPOINTER` 指令获取数组的首地址, 再用 `LOAD` 和 `STORE` 指令对数组内容进行存取。

2. mem2reg

`mem2reg` 需要将 `load / store` 形式的中间代码通过插入 `phi` 的方式转化为 `phi` 形式的 `SSA`。

需要注意的是, 该步骤只针对局部 `int` 类型变量进行处理, 而不考虑数组、全局变量等情况。

a. 划分基本块

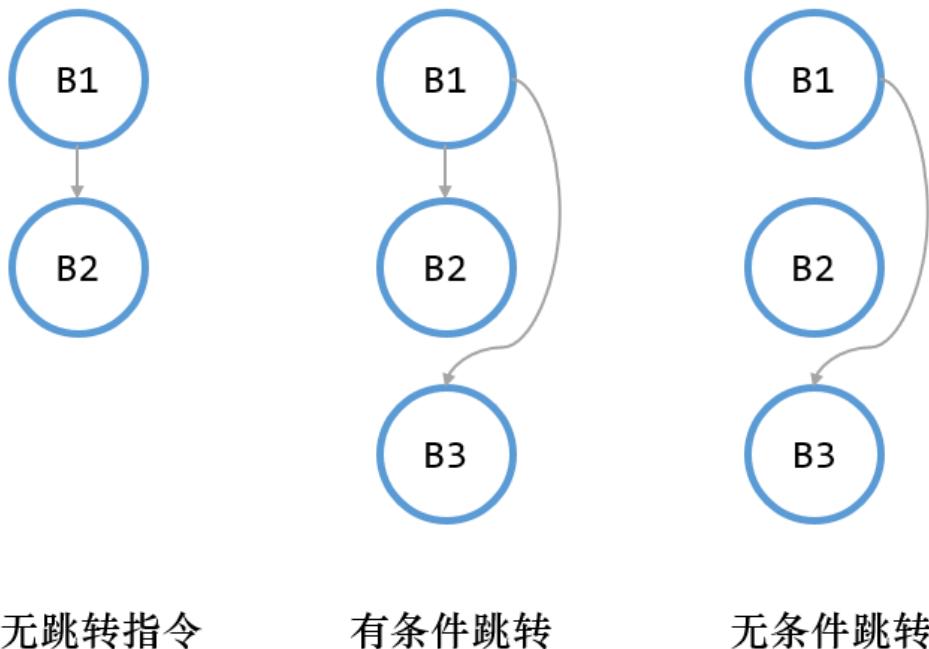
基本块的程式有以下特点:

- 单一入口点, 其他程式中, 没有任何一个分支指令的目标在这段程式基本块之内 (基本块的第一行除外)。
- 单一结束点, 这段程式一定要执行完最后一行才会执行其他基本块的程式。

因为上述特点, 基本块中的程式, 只要执行了第一行, 后面的程式码就会依序执行, 每一行程式都会执行一次。

由基本块的特性, 在划分基本块时只需关心跳转指令, 并根据不同的跳转指令生成基本块的前驱、后继关系, 其具体算法如下:

- 由于在生成中间代码时, 需要跳转到的地址前均有一个对应的 `label`, 因此先针对每一个 `label` 新建其对应的基本块类, 并初始化其前驱、后继基本块为空集。
- 遍历所有中间代码, 并记录当前基本块 `currentBasicBlock` 和上一个遍历完的基本块 `prevBasicBlock`
 - 当遇到一个新的 `label` 时
 - 若 `prevBasicBlock` 为空, 则说明当前基本块为程序第一个基本块, 没有对应的前驱。
 - 否则, 寻找上一个基本块中最后一条指令, 并分类讨论



b. 建立深度优先搜索树

根据基本块的定义，我们很容易联想到图论中的知识，我们可以把每个基本块当作一个节点，而基本块之间的跳转关系则为基本块之间的边。这样生成的图就是程序的控制流图 (CFG, Control Flow Graph)。可以发现，由于循环和跳转的存在，有些基本块无法被到达，而有些可以形成环路。但这样复杂的跳转关系不利于生成 phi 节点，因此我们需要在控制流图的基础上生成支配树。

从第一个基本块开始，依照深度优先的顺序遍历所有基本块，每遇到一个新的基本块时则将其的 dfn 更新为深度优先搜索树的遍历顺序。

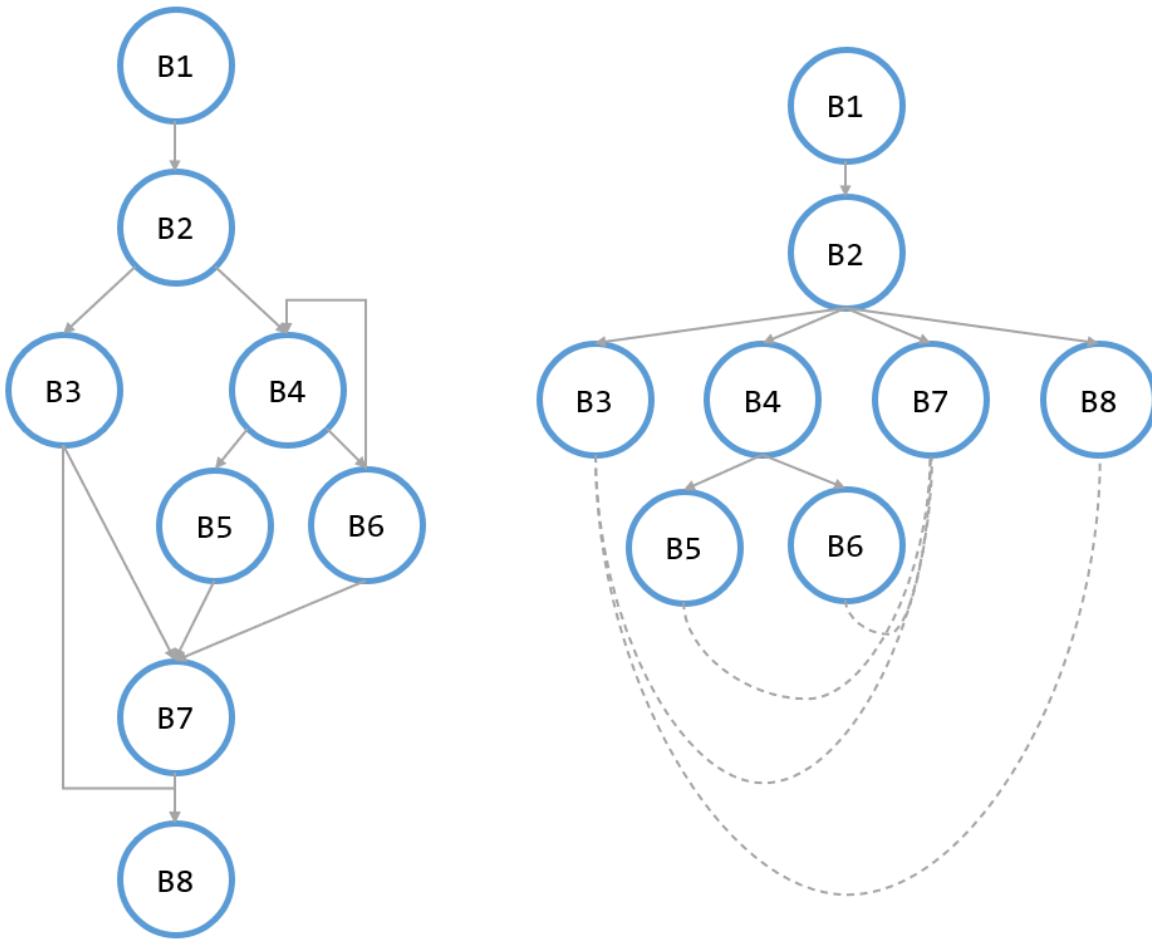
c. 无用基本块删除

在深度优先遍历后，未被遍历到（dfn 未更新）的基本块是在程序中不可到达的，因此可以直接删除。

d. 计算半必经点

前置概念 ([参考链接1](#), [参考链接2](#))

- **定义**：对变量进行初始化、赋值等改变变量的值的行为。
- **使用**：在语句/指令中将变量的值作为参数的行为。
- **控制流图** (Control Flow Graph, CFG)：一个程序中所有基本块执行的可能流向图，图中的每个节点代表一个基本块，有向边代表基本块间的跳转关系。
- **支配** (dominate)：对于 CFG 中的节点 n_1 和 n_2 , n_1 支配 n_2 当且仅当所有从入口节点到 n_2 的路径中都包含 n_1 ，即 n_1 是从入口节点到 n_2 的必经节点。需要特别注意的是，每个基本块都支配自身。
- **严格支配** (strictly dominate)： n_1 严格支配 n_2 当且仅当 n_1 支配 n_2 且 $n_1 \neq n_2$ 。
- **直接支配者** (immediate dominator, idom)：节点 n 的直接支配者严格支配 n ，且不严格支配任何严格支配 n 的节点的节点）。入口节点以外的节点都有直接支配者。节点之间的直接支配关系可以形成一棵**支配树** (dominator tree)。
- **必经点**：若 n_1 严格支配 n_2 ，则 n_1 为 n_2 的必经点
- **半必经点**：对于一个节点 n_2 ，若存在某个点 n_1 能够通过一系列点 p_i (不包含 n_1 和 n_2) 到达且 $\forall p_i$ 都有 $dfn[p_i] > dfn[n_2]$ ，则称 n_1 是 n_2 的半必经点，用 $sdom$ 表示。



算法流程

依据深度优先搜索的相反顺序遍历所有基本块，并通过下面的算法找出每个基本块的半必经点。

- 对于基本块 Y，对于它的每一个前驱基本块 X，若 $\text{dfn}[X] < \text{dfn}[Y]$ ，则 X 是 Y 的一个半必经点。
- 否则，对于 X 在深度优先搜索树中的祖先 Z（包括 X 自身），如果满足 $\text{dfn}[Z] > \text{dfn}[Y]$ ，那么 Z 的半必经点也是 Y 的半必经点。

e. 计算直接支配点

一个点的半必经点可能是也可能不是一个点的支配点。

对于一个点 X，考虑深度优先搜索树上从 X 的半必经点 $X.\text{sdom}$ 到 X 路径上的所有点 p_0, p_1, \dots, p_k ，对于所有的 $p_i : 0 \leq i \leq k$ ，记其中 $\text{dfn}[p_i.\text{sdom}]$ 最小的一个 p_i 为 Z。

- 若 $Z.\text{sdom} = X.\text{sdom}$ ，则 $X.\text{idom} = X.\text{sdom}$
- 否则， $X.\text{idom} = Z.\text{idom}$

由此，已经计算出所有基本块的直接支配者，也就是在支配树中的父节点。

f. 计算支配边界

在完成上一步后，我们已经成功将控制流图转化为支配树，下一步需要计算出每个基本块的支配边界 (Dominance Frontier)，以便之后插入 phi 节点。

计算支配边界的算法如下：

Algorithm 3.2: Algorithm for computing the dominance frontier of each CFG node.

```
1 for  $(a, b) \in$  CFG edges do
2    $x \leftarrow a$ 
3   while  $x$  does not strictly dominate  $b$  do
4      $\text{DF}(x) \leftarrow \text{DF}(x) \cup b$ 
5      $x \leftarrow \text{immediate dominator}(x)$ 
```

g. 插入 phi 节点

构建出支配树，求出各基本块的支配边界之后，我们终于可以向正规的 SSA 形式迈进了，这种实现从 STORE, LOAD 形式的代码向 SSA 转换的过程被称为 mem2reg，具体的实现方法则是插入 phi 节点。

插入 phi 节点需要计算出每个变量的定义、使用情况，其具体算法如下：

Algorithm 3.1: Standard algorithm for inserting ϕ -functions

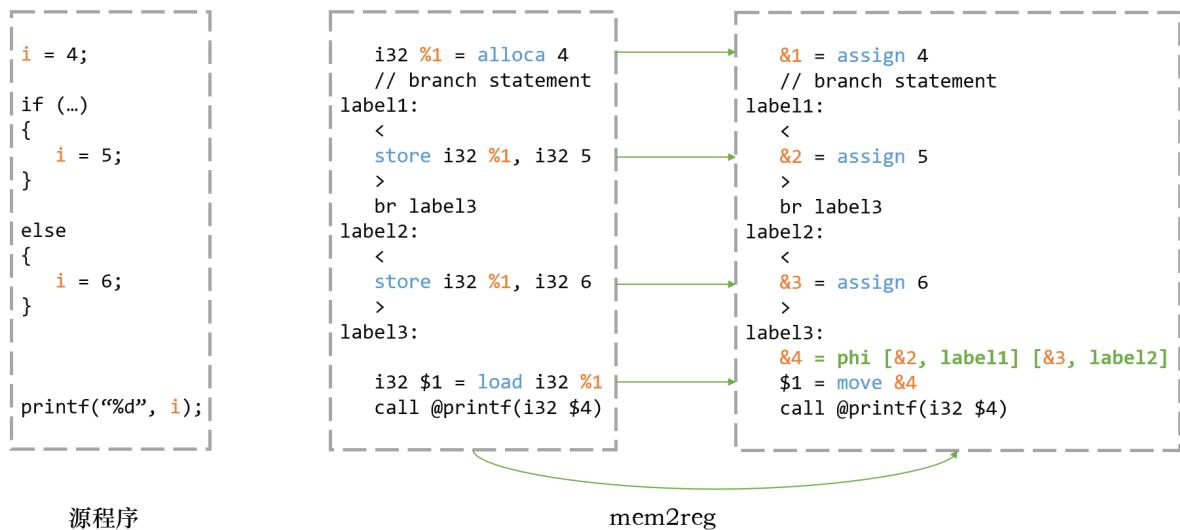
```
1 for  $v$ : variable names in original program do
2    $F \leftarrow \{\}$                                  $\triangleright$  set of basic blocks where  $\phi$  is added
3    $W \leftarrow \{\}$                                  $\triangleright$  set of basic blocks that contain definitions of  $v$ 
4   for  $d \in \text{Defs}(v)$  do
5     let  $B$  be the basic block containing  $d$ 
6      $W \leftarrow W \cup \{B\}$ 
7   while  $W \neq \{\}$  do
8     remove a basic block  $X$  from  $W$ 
9     for  $Y$ : basic block  $\in \text{DF}(X)$  do
10       if  $Y \notin F$  then
11         add  $v \leftarrow \phi(\dots)$  at entry of  $Y$ 
12          $F \leftarrow F \cup \{Y\}$ 
13         if  $Y \notin \text{Defs}(v)$  then
14            $W \leftarrow W \cup \{Y\}$ 
```

h. 变量重命名

在插入 phi 节点后，我们还需要对变量进行重命名，从而使其满足 SSA 的要求。

具体的来说，对于 STORE 指令，我们新产生一个变量，并将其转化为对该变量的赋值语句 (assign)；对于 LOAD 指令，我们将其转化为 move 指令；而对于 phi 指令，我们也新产生一个变量，并将 phi 指令中的 val 值相应的填充为变量号。

一个具体的例子如下：



通过上述一系列的操作，我们终于得到了含有 phi 语句，且符合 SSA 规范的程序了。

难点分析

生成 SSA 形式的中间代码是整个编译器中较为复杂的一步，它设计到大量图论相关的知识，且在实现时也需要阅读很多的教材、文献。

笔者在最初生成中间代码时由于懒惰和过分自信，并没有阅读文献就自行编写 mem2reg 相关代码。编写出的编译器针对小型样例可以通过，但在编译大型程序时出现了时间复杂度过高的情况，因此不得不阅读论文并重构，在这一步走了不少弯路。事实证明，阅读文献是很重要的，文献中的方法不但经过了正确性的检验，算法的时间复杂度也较低，虽然阅读文献需要耗费时间和经历，但却能大大提高写码时的效率，也让 debug 变得简单。

中间代码优化

下面简单介绍编译器中在中间代码中进行的优化，需要注意的是，在实现时对每一种优化均设置了开关，这样便于查看优化效果，也便于程序调试。

生成 SSA 形式的中间代码极大的简化了中间代码的优化难度。以下是一些笔者采用的代码优化：

死代码删除

一般代码删除

生成 SSA 形式的代码后，死代码删除就很好实现了，只需要找到定义但未被使用的变量，将其相关语句删除即可。

跳转代码删除

对于 `BRTRUE` 与 `BRFALSE` 语句，当判断跳转的变量取固定值时，则只有一种跳转的可能，因此可删去无法跳转到的分支。

常量折叠

由于每个变量只被定义一次，因此若在定义时为常数，则变量始终为常数。特别的，当 phi 语句可取的值均为常数且相同时，phi 语句生成的变量也为常数。

函数内联

1. 判断函数是否内联

当函数不存在递归调用时，则可以内联。由于 SysY 语法中不包含函数声明语句，因此不需要生成函数调用关系图即可以判断函数是否可以内联。

具体的来说，在语法分析的过程中，若在函数定义中调用了自己，则存在递归调用；否则则为可以内联的函数。

2. 函数内联实现

返回语句

函数内联实现时，需要特别注意返回语句的处理。由于返回语句执行后将不再执行之后的语句，因此需要对每个内联的函数末尾生成一个 label，将返回语句转化为向相应 label 的跳转语句。

返回值

由于函数可能包含多条返回语句，且可能有多种可能的返回值，因此将返回值作为一个变量处理。

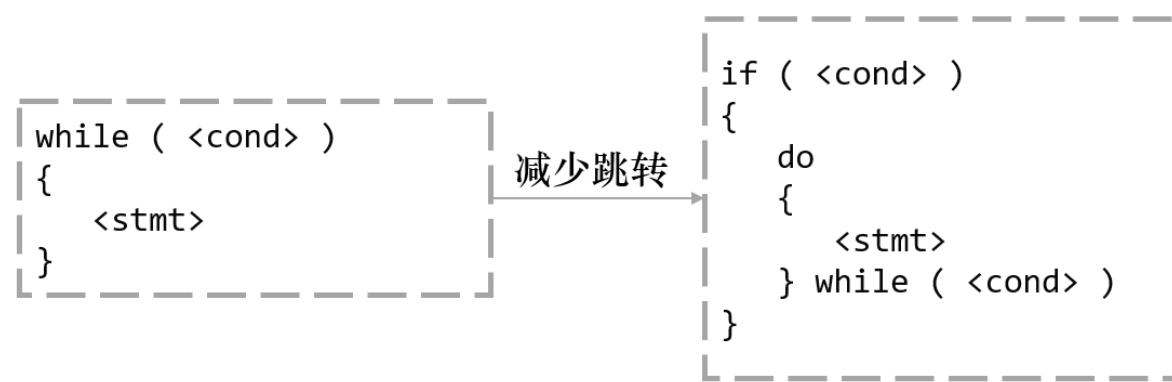
具体的来说，若函数的返回类型为 int，则在开始函数内联之前先声明一个变量，用于存储返回值。在遇到内联函数的返回语句时向该变量中填入相应的值。

循环优化

循环是程序中很重要的组成部分，且虽然语句量不一定多，但占据了程序运行中的大量时间，因此针对循环的优化可以有效提高程序的运行效率。

1. while -> do-while

将 `while` 语句改写为 `do-while` 形式，可以减少跳转次数。



删除无用循环

当循环内部的变量在循环外不再使用时，可以将整个循环删除。判断循环内变量是否使用需要配合活跃变量分析。

循环展开

将小循环体展开，配合跳转代码删除，可以有效减少循环所需要的判断。

局部数组提升 & 标记常量数组

由于在全局中可直接用 `.word` 进行初始化且不需要占用运行时间。因此，若局部数组的值未被改变或原本就定义为常量数组，则可以将其提升为全局数组并在全局中初始化。

全局转局部

由于对全局变量的读写均需要直接进行内存的读写，占用程序运行时间，因此对于函数中不会直接引用到的全局变量，可将其转化为局部变量并对其分配寄存器，减少存取指令，提高运行效率。

目标代码生成

下面介绍由中间代码生成目标代码的过程。

数据结构

.RegisterType

为了能够更方便的表示各个寄存器，新建 `.RegisterType` 枚举类，如下：

```
enum RegisterType {
    REGZERO = 0, REGAT = 1,
    REGV0 = 2, REGV1 = 3,
    REGA0 = 4, REGA1 = 5, REGA2 = 6, REGA3 = 7,
    REGT0 = 8, REGT1 = 9, REGT2 = 10, REGT3 = 11, REGT4 = 12, REGT5 = 13, REGT6
= 14, REGT7 = 15, REGT8 = 24, REGT9 = 25,
    REGS0 = 16, REGS1 = 17, REGS2 = 18, REGS3 = 19, REGS4 = 20, REGS5 = 21,
    REGS6 = 22, REGS7 = 23,
    REGK0 = 26, REGK1 = 27,
    REGGP = 28, REGSP = 29, REGFP = 30,
    REGRA = 31,
    REGDEFAULT = 32
};
```

MIPSCode

构建 `MIPSCode` 类，用来表示 mips 代码。此外，为了更好的区别不同种类的 mips 代码，新建枚举类 `MIPSType`，如下：

```
enum MIPSType {
    MIPSADD, MIPSADDI, MIPSDIV, MIPSMULT, MIPSSUB, MIPSSUBI, MIPSMUL, MIPSMULU,
    MIPSAND, MIPSANDI, MIPSOR, MIPSORI, MIPSNOT,
    MIPSBEQ, MIPSBEQZ, MIPSBEQZ, MIPSBLEZ, MIPSBLTZ, MIPSBNNE,
    MIPSSLT, MIPSSLE, MIPSSGT, MIPSSGE, MIPSSSEQ, MIPSSNE,
    MIPSJ, MIPSJAL, MIPSJALR, MIPSJR,
    MIPSMFH, MIPSMFL,
    MIPSSLL, MIPSSLLV, MIPSSLTI, MIPSSRA, MIPSSRAV, MIPSSRL, MIPSSRLV,
    MIPSSW, MIPSLW, MIPSLWLABEL, MIPSSWLABEL,
    MIPSSYSCALL, MIPSLABEL, MIPSDATA, MIPSTEXT, MIPSLI, MIPSLA,
    MIPSSPACE, MIPSWORD, MIPSAASCIIIZ, MIPSMOVE
};

class MIPSCode {
public:
    MIPSType op; // 操作符
    RegisterType rs = REGDEFAULT; // rs 寄存器
```

```

RegisterType rt = REGDEFAULT; // rt 寄存器
RegisterType rd = REGDEFAULT; // rd 寄存器
int imm = 0; // 立即数
string label;
};

```

Register

表 2.1: 通用寄存器的习惯命名和用法

寄存器编号	助记符	用法
0	zero	永远返回 0
1	at	(assembly temporary 汇编暂存) 保留给汇编器使用
2-3	v0, v1	子程序返回值
4-7	a0-a3	(arguments) 子程序调用的前几个参数
8-15	t0-t7	(temporaries) 临时变量, 子程序使用时无需保存
24-25	t8-t9	
16-23	s0-s7	子程序寄存器变量; 子程序写入时必须保存其值并在返回前恢复原值, 从而调用函数看到这些寄存器的值没有变化。
26,27	k0,k1	保留给中断或自陷处理程序使用; 其值可能在你眼皮底下改变
28	gp	(global pointer) 全局指针; 一些运行系统维护这个指针以便于存取 static 和 extern 变量。
29	sp	(stack pointer) 堆栈指针
30	s8/fp	第九个寄存器变量; 需要的子程序可以用来做帧指针(frame pointer)
31	ra	子程序的返回地址

为了能够更好的调度寄存器和存储寄存器的使用情况, 构建 `Register` 类, 用于存储各个寄存器是否被使用, 已经分配空寄存器。

```

class Register {
public:
    map<RegisterType, bool> registers; // 各个寄存器的标号及其使用情况

    Register() { // initialize register
        for (int i = 0; i < 32; ++i) {
            registers[(RegisterType) i] = true; // 初始化寄存器, 将所有使用位标志为空闲
        }
    }

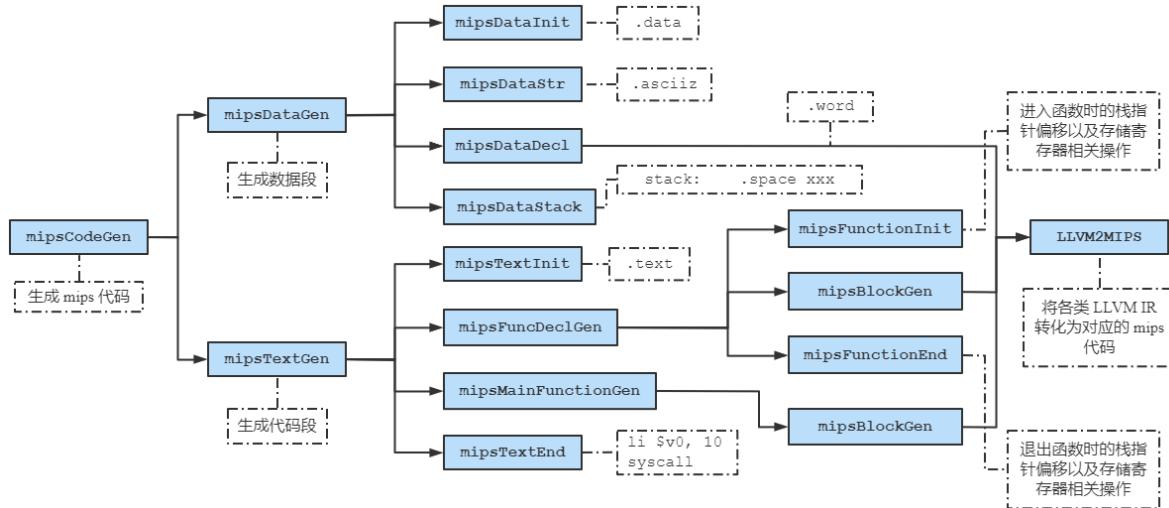
    void freeRegister(int id) {
        registers[(RegisterType) id] = true; // 将指定 id 的寄存器标记为空闲
    }

    int getFreeTRegister() {
        // 获得空闲的 t 寄存器, 若全部 t 寄存器均被使用, 则返回 -1
    }
};

```

算法流程

生成中间代码的主要算法流程用下图表示。



其中，需要特别注意的是在函数调用过程中的寄存器偏移和在退出基本块时的寄存器偏移。

此外，也需要在生成代码时记录下各个变量的地址，以便后续调用。

难点分析

函数调用

函数调用时需要对 s 寄存器和必要的 t 寄存器进行保存，为了程序实现的正确性以及规范性，需要区分调用者和被调用者，并分别保存相应的变量。

更具体的来说

- 调用者需要保存必要的临时变量
- 被调用者需要保存所有 s 寄存器，函数的返回地址，传入参数，初始栈指针与帧指针

栈指针偏移

目标代码生成时最重要也最容易出问题的部分就是栈指针偏移。

对于全局变量，可以直接用 la (load address) 从全局中获取变量地址并进行存取操作，而对于局部变量，则需要通过栈指针 sp 和帧指针 fp 来确定其相对地址。

在实现时，在进入函数时将原帧指针地址保存并将帧指针移动至此时栈指针所在的位置，并在整个函数体内部不再改变，直至函数体执行完毕后再将帧指针恢复。这样做的好处是，在栈指针地址不断变化时可以通过相对于帧指针的偏移来获取局部变量的地址。

phi 函数的实现

phi 函数两种可能的实现方式，一是在遇到 phi 函数时用 move 指令将所有可能的取值移动到同一个寄存器上，而另一种相对简单的方法是在分配寄存器时即将属于同一个 phi 函数的变量分配同一个寄存器。笔者采用的是后者。因此在全局分配寄存器前，需要先根据 phi 函数对变量进行归类，再将同一类的多个变量看作一个变量，进行活跃、冲突分析和寄存器分配操作。

目标代码优化

寄存器分配

采用图着色寄存器分配法，先通过活跃变量分析得到各个变量之间的冲突图，再在图中不断移去变量，最终完成寄存器分配。

乘除优化

乘除法占据的指令周期数极大，尤其是除法指令，一条就可以占据 100 个时钟周期，因此乘除指令的优化效果也十分显著。

主要针对一个操作数为常数的乘除指令进行优化。

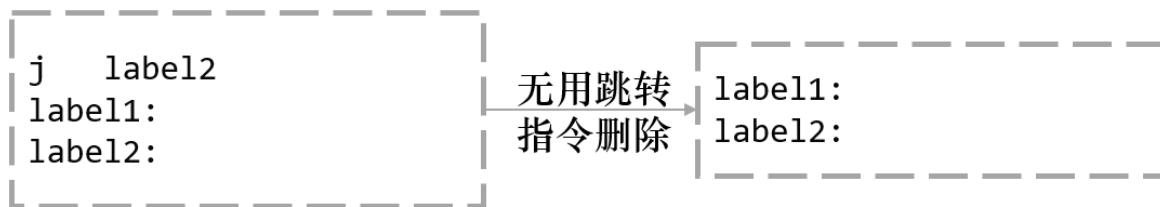
乘法的优化方式主要为：将乘法转化为左移和加法操作，需要注意正负号的问题。

除法的优化方式参照论文 Division by Invariant Integers using Multiplication，可以将除法指令转化为乘法指令。

余数也可以转化为乘除指令，再进行相应优化 ($a \% b = a - a / b$)

窥孔优化

无用跳转指令删除



无用 move 指令删除



结语

一学期的编译实验终于要告一段落了。

在学期开始之初，我无论如何也想象不到自己可以用并（没）不（有）熟（用）练（过）的 c++ 写出上万行的程序，并且取得还不错的性能。因此，编译对我的代码能力和设计能力都有着极大的提升。看到编译器正确运行并且输出目标代码，还是很有成就感的。

在编写大型系统时，一个清晰的设计尤其重要。因此，在编写代码前我都会给自己列出一个步骤清单，先把架构设计想清楚，从整体结构再到每步实现时的注意事项，然后严格按照之前书写的步骤进行编译器的书写。这样做虽然一开始进度可能会慢于周围的同学，但一个清晰的思路会让后面的工作轻松不少。

除此之外，代码风格和命名也十分重要。在编译器的规模逐渐增加时，一些具体的函数实现方法可能已经变得模糊，如果能有一个清晰的命名和辅助的注释，则可以让自己迅速理解函数的作用，并决定是否调用。

总的来说，虽然中间有过很痛苦的时候，但能自己写出一个基本的编译器还是很有趣的经历 :)

参考文献

- [1] 《编译技术》, 张莉, 史晓华等
- [2] *Modern Compiler Implementation in C*, Andrew W. Appel (2004)
- [3] *Static Single Assignment Book*, 2018, Lots of authors
- [4] *LLVM IR TUTORIAL*, LLVM Developers Conference, Vince Bridgers, Felipe de Azevedo Piovezan
- [5] *LLVM Cookbook*, Mayur Pandey, Suyog Sarda
- [6] A Linear Time Alogrithm for Placing ϕ -Nodes, Vugranam C. Sreedhar, Guang R. Gap, School of Computer Science, McGill University
- [7] Division by Invariant Integers using Multiplication., Granlund, Torbjorn & Montgomery, Peter. (2004).