《编译技术》课程实验

申优文档

张羿凡
19373758

北京航空航天大学
计算机学院

2021 年 12 月 22 日

目 录

1 概述

本课程实验为编写一个将 SysY 语言翻译为 MIPS 汇编语言的编译器，编写所用到的高级语言为 C++。SysY 语言是 C 语言的一个子集，支持的功能包括但不限于：

- 常量变量定义与使用
- 一维二维数组的定义与使用
- 条件与循环
- 函数调用
- 整数输入
- 字符串和整数的输出
- 短路求值

编译过程可以分为五个阶段：

- 词法分析
- 语法分析
- 语义分析和中间代码生成
- 代码优化
- 目标代码生成

此外，还有符号表管理和错误处理两部分穿插在五个阶段之中，接下来的架构设计章节将逐个介绍每一部分。代码优化部分将中间代码转化为了 SSA (Static Single-Assignment，静态单一赋值) 形式，并进行了相关优化，将额外开设一个章节进行介绍。

2 架构设计

2.1 词法分析

词法分析过程本质是根据文法所用到的终结符构建一个 DFA，识别出所有的关键字及标识符。由于实现的文法关键词较少，文法相对简单，关键词长度不超过 10，实现过程中并不需要真正构建出一个 DFA，只需模拟 DFA 进行贪心匹配即可。具体来说，每个关键字 / 标识符之间可以用如下的方式来进行分隔：

- 空格、制表符、换行符。
- 字母数字下划线和其它字符相邻。

对于每个分隔出来的部分去判断其是属于哪一类，如果以数字开头则是 `IntConst`，否则如果和某个关键字相同则是对应关键字，否则是标识符。

因为文法中支持单行注释、多行注释和字符串输出，要进行三个特判：

- 单行注释，从 `//` 开始，到换行符结束。
- 多行注释，从 `/*` 开始，到 `*/` 结束。
- `FormatString`，从 `"` 开始，到 `"` 结束。

使用三个变量标记当前是否处于上述三者内，若是则跳过即可。

此外，还有两个细节需要注意，不过无伤大雅：

- 由于很多数据是在 Windows 上生成的，需要特判 `\r` 字符。
- 像 `=` 和 `==` 这样的关键字，不能只读一个就判断，需要读入第二个字符后再进行判断。

2.2 语法分析

产生 SysY 语言的文法是 CFG (Context Free Grammar，上下文无关文法)，后续语义分析中通过符号表以实现上下文的联系。

首先进行词法分析，并将每个关键字及标识符作为终结符，使用一个结构体进行存储：

```
struct Terminal{
    string S, s;
    int line;
};
```

分别存储该终结符的原字符串、类别码以及行号。

接下来使用递归下降进行推导，文法中不存在能推出 ε 的非终结符，因此考虑回溯问题时可以不用考虑非终结符的 FOLLOW 集合，仅需避免 FIRST 集合相交并删除左递归。

对于 FIRST 集合相交，我通过预读多个终结符来解决，具体来说，构建如下函数：

```
inline bool check(initializer_list<string> a){
    int i = 0;
    for(auto x : a){
        if(pos + i == v.size()) return false;
        if(v[pos + i].S != x) return false;
```

```
            i++;
        }
        return true;
    }
```

initializer_list 可以实现传入不定个参数，例如 {"a", "bb", "ccc"}，该函数返回接下来往后读的终结符是否和传入的参数相同，这样就可以解决 FIRST 集合相交的问题，递归下降时知道选择哪一个推导。

举例来说，CompUnit 这一语法成分中，FIRST 集合相交的问题，只读取一个终结符不知道该推导为变量声明还是函数声明，用以下代码即可完成识别：

```
    void CompUnit(){
        while(1){
            if(check("CONSTTK")) ConstDecl();
            else if(check({"INTTK", "IDENFR", "ASSIGN"}) ||
                    check({"INTTK", "IDENFR", "LBRACK"}) ||
                    check({"INTTK", "IDENFR", "SEMICN"}) ||
                    check({"INTTK", "IDENFR", "COMMA"})) VarDecl();
            else break;
        }
        while(1){
            if(check("VOIDTK") || check({"INTTK", "IDENFR"})) FuncDef();
            else break;
        }
        MainFuncDef();
        output("CompUnit");
    }
```

对于左递归，仅出现在 _____ Exp 相关语法成分，使用扩展的 Backus 范式将其进行修改即可解决，例如：

$$\texttt{MulExp} \rightarrow \texttt{UnaryExp} \mid \texttt{MulExp ('*' | '/' | '\%') UnaryExp}$$

可以修改为：

$$\texttt{MulExp} \rightarrow \texttt{UnaryExp \{('*' | '/' | '\%') UnaryExp\}}$$

实践过程中，这两个应对策略都存在一个问题：

- 对于前者，有一个推导需要预读的终结符数量不定：

  $$\texttt{Stmt} \rightarrow \texttt{LVal '=' Exp ';' | [Exp] ';'}$$

  如果下一个终结符是 Ident，则预读情况较为复杂，处理方式为先调用 LVal 的递归下降子程序，看下一个字符是否是 =，若是则按第一个推导接着来，若不是则调用 Exp 的递归下降子程序并通过特判最终进入到 LVal，即可解决这一问题。

然而这个解决方法仅仅完成了语法分析的任务，在后续语义分析过程中会造成语法信息的丢失，因此实际采取了另一种更为暴力的方法：向后遍历直到分号出现，如果遇到等号则是第一个推导，否则认为是第二个推导，对于正确的代码，复杂度是均摊 $O(n)$ 的，即每个字符至多被遍历常数次。

- 对于后者，因为 OJ 的判题方法为每完成一个语法成分就要输出一个对应名称，以上面的 `MulExp` 为例，无论有多少个 `UnaryExp`，文法修改后相当于仅有一个 `MulExp`，但实际对应很多个 `MulExp`，因此加上对应的输出即可，例如：

```
void MulExp(){
    UnaryExp(), output("MulExp");
    while(check("MULT") || check("DIV") || check("MOD")){
        get(), UnaryExp(), output("MulExp");
    }
}
```

2.3 错误处理

错误处理实际上应该穿插在五个阶段中，例如词法分析过程中如果无法识别单词、语法分析中语法成分缺失、语义分析中使用未定义的变量等，但本课程设计中错误处理部分与其它部分较为独立，作为一次单独的作业出现，而之后的代码生成与竞速作业均保证代码是正确的。

此外，测试样例中所有可能的错误都已经给出，仅在递归下降过程中建立简单的符号表就可以判断，而不需要进行更具体的语义分析。

首先基于语法分析程序，构建符号表，使用三个结构体，分别对应作用域、常/变量、函数：

```
struct Scope{
    bool loop; //是否在循环体内
    bool ret; //是否有返回值
    set<string> var, fun; //对应的常/变量和函数
};

struct Var{
    bool con; //是否是常量
    int dim; //维数
};

struct Fun{
    bool ret; //是否有返回值
    vector<int> v; //参数对应维数
};
```

接着使用三个数据结构分别存储目前栈中的作用域，及每个名称对应常/变量和函数对应的结构体的栈，同一个名称可能对应多个，栈顶即为当前起作用的：

```
vector<Scope> w;
map<string, vector<Var> > vmp;
map<string, Fun> fmp;
```

通过两个函数完成进入作用域和离开作用域：

```
    inline void scopeIn(bool loop, bool ret){
        w.push_back({loop, ret});
    }

    inline void scopeOut(){
        for(auto s : w.back().var){
            vmp[s].pop_back();
            if(vmp[s].empty()) vmp.erase(s);
        }
        w.pop_back();
}
```

这样名称重定义、未定义的错误都可以处理。

对于剩下的错误：

- 非法符号，直接遍历每个 FormatString 即可。
- 函数参数个数不匹配，调用时对参数个数计数，之后查符号表。
- 函数参数类型不匹配，调用时将对应维数放入 vector 和符号表中对应的 vector 比较。
- 无返回值的函数存在不匹配的 return 语句，查当前作用域是否是有返回值的函数即可。
- 有返回值的函数缺少 return 语句，特判函数最后一句内容，并查当前作用域是否是有返回值的函数。
- 不能改变常量的值，查符号表即可。
- 缺少分号 / 右小括号 / 右中括号，递归下降过程中直接处理即可。
- printf 中格式字符与表达式个数不匹配，遍历 FormatString 得到数量后和后面参数数量对比。
- 在非循环块中使用 break 和 continue 语句，查当前作用域是否是循环体内部即可。

此外，函数参数类型不匹配，这个错误还存在参数中调用无返回值的情况，例如 g() 是一个无返回值函数，对于函数 void f(int x)，f(g()) 就会出错。

这一点递归下降中直接判断函数参数的属性，如果出现这种情况直接判定为出错即可。

2.4 语义分析和中间代码生成

语义分析的最终目标是生成中间代码，通过代码优化后再将其转化为目标代码也就是 MIPS 代码。

我采取自顶向下的语法制导翻译完成语义分析，在语法分析递归下降的过程中同时完成符号表的填充以及中间代码的生成，而不是显式地建出语法树，这样的好处是代码量小，容易编写，将代码优化部分统一放在后续来做，避免模块间耦合度过高。

2.4.1 中间代码格式

中间代码	含义
add a b c	$a \leftarrow b + c$
sub a b c	$a \leftarrow b - c$
mul a b c	$a \leftarrow b \times c$
div a b c	$a \leftarrow b \div c$
mod a b c	$a \leftarrow b \bmod c$
neg a b	$a \leftarrow b$
not a b	$a \leftarrow [b = 0]$
lss a b c	$a \leftarrow [b < c]$
gre a b c	$a \leftarrow [b > c]$
leq a b c	$a \leftarrow [b \leq c]$
geq a b c	$a \leftarrow [b \geq c]$
eql a b c	$a \leftarrow [b = c]$
neq a b c	$a \leftarrow [b \neq c]$
beqz a b	如果 $a = 0$ 则跳转到标签 b
bnez a b	如果 $a \neq 0$ 则跳转到标签 b
jump a	跳转到标签 a
assign a b	$a \leftarrow b$
store a b c	将数组 a 第 b 个元素赋值为 c
load a b c	将 a 赋值为数组 b 第 c 个元素的值
label a	标签 a
para a b	声明变量 a 是第 b 个参数
push a b	将 a 作为第 b 个参数进行传参
call a	调用函数 a
return a	函数返回，将 a 作为返回值，若为空则无返回值
get a	将 a 赋值为刚调用完参数的返回值
addr a b	将 a 赋值为 b 的地址，b 是一个数组或数组地址参数
prints a	将 a 作为字符串输出
printd a	将 b 作为有符号十进制整数输出
getint a	将 a 赋值为读入的整数

对一些符号进行解释：

- $a \leftarrow b$ 是指将 b 的值赋 a。
- 所有运算均和 C 语言中 32 位有符号整数的运算相同，特别是除法和取模。
- 方括号 [] 称为艾佛森括号，若其中的表达式为真则值为 1，否则为 0，形式化地说：

$$[P] = \begin{cases} 0, & P = \text{false} \\ 1, & P = \text{true} \end{cases}$$

2.4.2 变量作用域

符号表和作用域等处理方式依旧延续错误处理的设计，为解决同名变量在不同作用域中的问题，一种方式是加入两种中间代码分别表示进入作用域和离开作用域，然而这种方法的可优化性和可拓展性极低，取而代之的是为每个作用域的每个变量重新命名，这样两个不同的变量都会有不同的名字，生成中间代码后就可以彻底抛弃作用域这个概念。我通过在符号表中加入每个变量的新名字来实现这一方法，在变量名后面拼接 `_id` 得到新的名字，其中 `id` 是每个变量的唯一标识，通过自增的计数器保证任意两者均不同，对应结构体如下：

```
struct Var{
    string name;
    vector<string> v;
};
```

通过栈式符号表，翻译过程中可以得到当前作用域对应的变量，从而生成正确的中间代码。

2.4.3 变量与数组声明

可以看到中间代码中不包含变量和数组的声明，这是因为它们不需要出现在中间代码中，只需使用符号表记录它们，最终在目标代码生成时为他们分配空间即可。

2.4.4 常量预处理

对于所有的常量表达式，我们可以在递归下降的过程中就把他们算出来，只需在 Exp 相关的递归下降函数中设置返回值，返回对应的变量名或是常数，从而进行合并。

这一步看似是代码优化该做的事情，但是数组的大小可能是常量表达式，若不预处理在访问数组时将无法进行下标的计算。

2.5 代码优化

这一部分作为重中之重，将在后续的章节详细说明。

2.6 目标代码生成

暂不考虑寄存器的分配，寄存器分配的部分将在优化部分详细说明。

绝大部分中间代码可以直接翻译成对应的 MIPS 代码，需要特别说明的是运行栈设计。我采用的方式是以函数为单位，预先处理出每个函数内部定义了多少变量和数组，并计算出对应的空间，同时为每个变量和数组分配其相对于栈指针的偏移量，进入函数时将栈指针 (`$sp` 寄存器) 减少对应的值，离开函数时再把这个值加回来。

全局变量统一存放在 `.data` 区，将 `$gp` 寄存器设为其初地址以加速访问。当然每个变量相对其的偏移量也需要预先计算。

访问栈内元素时，要分三种情况：

- 如果访问的是全局变量，通过对应偏移量和 `$gp` 实现访存。
- 如果访问的是局部变量，通过对应偏移量和 `$sp` 实现访存。
- 如果访问的是参数中的数组地址对应的数组，通过对应偏移量和该值实现访存。

当然，获取地址时也要考虑以上三种情况。

输入、输出均使用 `syscall` 指令，所有输出的字符串通过 `.asciiz` 预先存放在 `.data` 区。

3 代码优化

除函数内联外，所有优化均为函数内优化，均局限于单个函数内。

3.1 函数内联

调用函数时，需要将还留在寄存器中的值写回内存，同时还要进行传参，因此多次调用会造成极大的开销。

函数内联即为将对应函数的中间代码复制过来，取代原本的函数调用，直接复制会造成一些问题，需要进行以下改动：

- 将传参改为数个中间代码 `assign` 来完成。
- 变量名字可能会和前一个函数重，不过不需要更改，因为函数之间是独立的，而且后续 SSA 还会再进行一次重命名。
- 所有标签都要统一偏移一个值，不然就跳转到之前的位置了。
- 定义的数组要额外分配好空间。

因为文法规定只会调用前面已经定义的函数或者自己，而不会有循环调用，因此上述过程可以直接在递归下降的部分完成，只需单独存储每个函数的中间代码即可。当然，调用自己是递归函数，内联效果不大，因此仍然保留函数调用。

3.2 划分基本块

基本块是指这样的极大连续中间代码段：一旦从代码段的开头开始执行，那么段内所有代码一定会被顺序执行。按照以下规则确定基本块的起点，即可将中间代码划分为数个基本块：

- 第一条中间代码。
- 跳转语句后的下一条代码。
- 可以被跳转到的代码，即中间代码中的 label。
- return 语句的下一条代码。

3.3 建立控制流图

控制流图 (Control Flow Graph, CFG) 是指这样的有向图：每个基本块代表一个点，如果基本块 i 可以跳转或顺序执行到基本块 j，那么有向边 $i \to j$ 存在。

划分基本块的过程中就可以同时建立控制流图，使用 `vector` 实现的邻接链表形式存储。

可以很容易删除从起点不能到达的基本块，下面默认所有基本块均可从起点到达。

3.4 转化为 SSA 形式

3.4.1 SSA 的定义

SSA (Static Single-Assignment，静态单一赋值) 形式，是指中间代码中每个被赋值的变量仅会被赋值一次。注意到全局变量可能跨函数调用，因此将全局变量当作大小为 1 的数组来处理。

SSA 形式下中间代码的很多优化将可以更高效地完成。

一种朴素的想法是直接进行重命名，这种做法在基本块内显然是成立的，但是跨越基本块时则会发生错误。例如，基本块 i 和基本块 j 中对同一个变量 x 进行了定义，分别重命名为 x_i 和 x_j，而 $i \to k$ 和 $j \to k$ 均存在于控制流图中，基本块 j 中用到了变量 x，那么程序在运行过程中将无法知道 x 应该是 x_i 还是 x_j，这时就需要在程序中插入 ϕ 函数以解决这个问题。

3.4.2 ϕ 函数的定义

ϕ 函数是一种特殊的中间代码，它形如：

$$x_k \leftarrow \phi(x_1, x_2, \ldots, x_n)$$

ϕ 函数被放置于基本块的首端，假设基本块为 b，那么右侧每一个 ϕ 中的 x_i 都对应于一个指向 b 的基本块 a_i，ϕ 函数的含义即为如果程序从基本块 a_i 移动到基本块 b，那么就令 $x_j \leftarrow x_i$。

3.4.3 ϕ 函数的翻译

中间代码的 ϕ 函数可以通过数据结构保存下来，但是 MIPS 中并没有这样的指令。因此，我们需要在翻译成 MIPS 指令时，假设从基本块 a 到 b，对于 b 中每一个 $x_b \leftarrow \phi(\ldots, x_a, \ldots)$，令 $x_b \leftarrow x_a$。

注意到每个基本块至多有两条出边 (当末尾是条件跳转时)，当基本块 b 结束时：

- 若 b 没有出边，则不需要翻译 ϕ。
- 若 b 仅有一条出边，则直接在末尾进行 ϕ 函数的翻译。
- 若 b 有两条出边，其中一条一定是到紧接着它的基本块，可以直接翻译，另一条新创一个标签跳转到末尾进行翻译，再回来跳转到对应的标签。

3.4.4 ϕ 函数的插入

我们可以在每一个基本块的入口插入每个变量的 ϕ 函数，但显然绝大部分是毫无意义的，这样反而会让程序变得巨慢无比。

为了只在必要的位置插入 ϕ 函数，我们需要求出控制流图的支配树，下面给出一些定义：

- a 支配 b 当且仅当所有从起点出发到 b 的路径都经过 a。
- 支配树是指这样一棵有根树，若 a 支配 b 则 a 是 b 的祖先，可以证明这一条件能唯一确定一棵有根树。

支配树可以在 $O(n \log n)$ 的时间内使用 Lengauer-Tarjan 算法求解，包括后续很多步骤都有更优的时间复杂度算法，但鉴于代码对应的基本块和变量数量不会超过 10^4 数量级，且编译器更加注重于目标程序的高效运行而非编译器的高效运行，我选择使用 $O(n^2)$ 甚至更高复杂度的算法进行后续内容的求解，这有效减少了代码编写的难度。

对于任意两点间的支配问题可以枚举支配点 i，从起点开始 dfs，不允许经过点 i，那么不能到的点 (除了 i) 均被 i 所支配。

接着我们可以得到每个点被哪些点支配，从而知道每个点在支配树上的祖先数量，这即为它在支配树上的深度，从小到大遍历深度 i，保证所有深度 $\leq i-2$ 的点都已经被标记，那么对于所有深度为 i 的点它们的祖先一定只有一个点没有被删除，即为它的父亲，这样就可以找到所有点的父亲，从而建出支配树。

我们称 a 严格支配 b 当且仅当 a 支配 b 且 $a \neq b$。定义点 x 的支配边界：

$$\mathrm{DF}(x) = \{y \mid x \text{ 支配 } z \text{ 但 } x \text{ 不严格支配 } y \text{ 且控制流图中存在边 } z \to y\}$$

我们可以遍历控制流图的每一条边从而得到每个点的支配边界。

可以证明，如果变量 x 在基本块 b 处有定义，那么所有 DF(b) 中的点都需要插入关于变量 x 的 ϕ 函数，其它点则不需要插入 ϕ 函数。注意到，此时 DF(b) 中的基本块也有关于变量 x 的定义，因此需要继续遍历其支配边界进行 ϕ 函数的插入，直到没有变化才停止迭代。

3.4.5 变量的重命名

插入完 ϕ 函数后，我们需要对所有变量进行重命名，同时注意到我们只是在一些基本块插入了关于一些变量的 ϕ 函数，并没有确定 $\phi(\ldots)$ 应该填写什么。

支配树所带来的节点支配性很好地帮我们解决了这个问题，在支配树上 dfs，每遇到一个变量的定义就为其重命名，同样使用在后面新增唯一标识符的方法。

注意对于全局变量和数组，不能对其进行重命名，前者可能在其它函数处被更改，后者当访问下标是变量时将无法确定访问的是哪个值。

对于每个变量来说，它的最新定义就是从它到根节点路径上最先遇到的该变量定义，这可以在 dfs 过程中通过入栈弹栈来维护每个变量的最新定义名称。

对于所有被使用而非被赋值的变量，将其更改为其最新定义所对应的名称。

每到达一个节点 i，就遍历其在控制流图上所有指向的节点，接着遍历对应节点基本块的 ϕ 函数，对于每一个 $x \leftarrow \phi(\ldots)$，将 x 的最新定义名称放入其中，并标记其来源也就是基本块 i，以便后续进行翻译。

至此，我们将中间代码转换为了 SSA 形式，此时的中间代码大概率是冗长且塞满 ϕ 函数的，接下来的优化就轮到 SSA 大显身手了。

3.5 死代码删除

这是 SSA 极其重要和关键的优化，它可以帮你删掉绝大多数无意义的代码。注意，我们只能删除所有 SSA 重命名过的变量，不能把全局变量或数组删除。

第一个算法称为正向删除，考虑维护每个变量被用到的次数，例如 $a \leftarrow b + c$ 中 b 和 c 各被用到了一次，而 $x_4 \leftarrow \phi(x_1, x_2, x_3)$ 中 x_1, x_2, x_3 各被用到了一次。不断将被用到 0 次的变量插入一个先进先出队列，不断从队列首端弹出变量，将该变量的定义删除，并减少其定义过程中用到变量的次数，一旦有新的变量使用次数为 0，就将其插入队列。

这个算法看起来可以删掉所有无用的代码，其实则不然，考虑下面这两条中间代码：

$$x \leftarrow \phi(y, \ldots)$$
$$y \leftarrow x \times 3$$

如果 x 和 y 没有在其它地方用到，那么显然它们都应该被删去，但按照上述算法它们都不会被删，因为它们的依赖关系成环，一旦中间代码中大量充斥着上述依赖环，程序的效率将大打折扣。

这时就需要使用第二个算法，称为逆向删除。考虑那些一定不能被删去的语句，及它们被用到的变量，包括：

中间代码	用到的变量
beqz a b	a
bnez a b	a
store a b c	b 和 c
para a b	a
push a b	a
return a	a

printd a	a

将这些变量放入一个先进先出队列，不断从队列首弹出变量，将该变量定义中用到的变量标记，如果该变量第一次被标记则将其放入队列，，直到队列为空。

所有被标记的变量均为有意义变量，其它变量的定义语句都可以被删掉。

这个算法可以删掉所有没用意义的变量，且时空复杂度非常高效。

需要特别注意的是，上述表格中如果一个值是常数而非变量，则不能进行操作。此外，getint 这个中间代码不要删，不然输入可能会错位，但是也不要将其标记为有意义变量，那样保留很多无意义变量，仅仅在最后删除的时候不要删它即可。

3.6 复制传播和常数传播

复制传播是指将形如 $x \leftarrow y$ 的中间代码，将之后的 x 全部用 y 代替。

SSA 形式下的复制传播可以做的更加彻底，可以将代码中全部的 x 全部用 y 代替，因为 SSA 形式保证了至多存在一个 x。

此外，还有两条特殊的合并：

- 对于 $x \leftarrow a$，如果 a 是常数，可以直接用 a 去代替所有 x。
- 对于 $x \leftarrow \phi(x_1, x_2, \ldots, x_n)$，如果全部的 x_i 都是同一个变量或同一个数，设为 y，可以将其改为 $x \leftarrow y$，从而删掉一条 ϕ 语句。特别地，$x \leftarrow \phi(y)$ 的情况也包括在这里。

实现上，使用并查集可以高效实现，初始每个变量各自属于一个集合，对于上述三种情况不断合并集合。特别地如果一个集合有常数，需要特别标记。合并完成后，将每个变量更改为自己所在集合 (并查集数组) 根节点的名字。

3.7 常数合并

对于中间代码中所列的基本运算，如果运算对象均为常数，可以直接算出来将其改为赋值语句。

例如对于中间代码 $x \leftarrow 1 + 2$ 来说，直接改为 $x \leftarrow 3$ 即可，之后通过常数传播就可以消掉这个变量。

对于一元运算符，如果运算对象是常数，那么直接就可以进行更改。

此外，还有一些特判，也许会有大的奇效：

- 对于加减乘运算，如果其中一个运算对象是 0，那么可以进行简化。
- 对于乘法运算，如果其中一个运算对象是 1，那么可以进行简化。
- 对于除法运算，如果其中一个被除数是 0 或除数是 1，可以进行简化。

3.8 公共子表达式合并

相较于复杂的 DAG 删除公共子表达式合并算法，SSA 可以使用更高效的算法完成。

因为每个变量只被定义一次，那么两个赋值语句相同当且仅当它们完全相同。这里说相同是指从编译器角度能判别的相同，当然可能一开始两个相等的赋值语句并不完全相同，但当做了足够多的复制传播和常数传播后，它们一定会相同。

因此对于块内的公共子表达式删除，直接从前往后扫记录每个将每个表达式放入哈希表或平衡树 (map) 中，判断之前有没有相同的即可完成消除，若没有则插入。

对于全局公共子表达式合并，需要用到支配属性，支配树又一次发挥了强大的作用。在支配树上 dfs，它祖先对应基本块中的变量都被定义过，因此如果有一个相等就可以进行替换。同样使用入哈希表或平衡树，判断有没有相同的，若有则替换，若没有则插入，继续 dfs 子树，回溯时再删除即可。

3.9 基本块合并

对于基本块 a 和 b，如果控制流图中有边 $a \to b$，同时 b 的入度恰为 1，那么可以将这两个基本块合并为同一个基本块。

具体来说，a 和 b 之间必然没有 ϕ 函数，因此只需将 a 中的 ϕ 函数迁移到 b 中，将所有指向 a 的边改为指向 b，同时如果 a 的结尾有跳转语句或 b 开头有标签，都将其删去，接着将 a 中的中间代码放到 b 中。最后删掉基本块 a 即可。

此外，如果条件跳转语句中的参数为常数，这条语句可以被优化为一个 jump 语句或者直接被删掉，此时要对控制流图中的边进行修改，以便进行下一次基本块合并。

3.10 寄存器分配

3.10.1 写回策略

一般来说，一个变量会占据一个寄存器。当有某些原因该寄存器要被分给别的变量时，需要将该寄存器的值写回内存。然而，绝大部分的写回是没有意义的，一个变量会被写回当且仅当：

- 它是在当前基本块定义的，否则因为只会被赋值一次，它一定被写回正确值了。
- 接下来有可能用的到它，这可以通过从当前中间代码开始 bfs，遍历所有可能到达的中间代码，设当前变量为 x，如果存在一条路径在遇到 x 被使用之前没有遇到 x 被定义，则有可能用的到它。这有些类似于活跃定义分析，不过比其更加精准。

这个策略几乎将访存指令缩减到了 $\frac{1}{5}$！

3.10.2 临时寄存器分配

需要明确的是，临时寄存器分配对于每个基本块是独立的，因为程序在进入一个基本块后，无法得知它是从哪个基本块来的，这和 ϕ 函数翻译要在出发基本块是同一个道理。

我采取的方法类似于页面置换中的 LRU 算法，维护一个寄存器池，每当一个变量需要寄存器时：

- 如果还有空的寄存器则分配给它，并标记该寄存器权值为最大。
- 否则找到权值最小的寄存器，将该寄存器分配给它，并标记该寄存器权值为最大。注意到此时也许需要将原本占有这个寄存器的值写到内存中，需要根据上述的写回策略进行判断。

值得注意的是，如果一个要被定义的变量被分配寄存器，那么不需要从内存中加载值，否则需要。

3.10.3 跨基本块的寄存器调度

每当一个基本块结束，将所有寄存器的值根据写回策略放回内存并不是最明智的，我选择令每个块初始将 ϕ 函数所定义的变量优先放到初始几个寄存器中，如果寄存器不够则还是放到内存 (有点类似于 MIPS 传参规范) 。

这样在翻译 ϕ 的时候，大部分情况只需将对应的值放到前几个寄存器中。翻译时，从小到大遍历对应的前几个寄存器，将原本的值根据写回策略决定是否写回，接着将对应的值放进去。

最后，根据写回策略，选择是否把除了前几个寄存器外的值放回内存。

3.10.4 全局寄存器分配

全局变量想要优化需要跨函数分析，我并没有做这个优化，因此过于精细的寄存器分配意义不大。

我采取了选择 M 个寄存器永久分配给一些全局变量，使用等量的 `move` 指令换取访存指令。具体分配给哪些全局变量取决于简单的引用计数，甚至可以采取随机化以应对构造数据。

3.11 运算强度削弱

3.11.1 乘法优化

$x \times 2$ 可以优化为 $x + x$。

$x \times 3$ 可以优化为 $x + x + x$，因为乘法代价是 3。

对于乘以 2 的次幂，可以使用位运算来代替。

上述只需要判变量乘以常数的情况，对于变量乘以变量的情况，再使用分支跳转就得不偿失了。

3.11.2 除法优化

因为 $x \bmod y = x - x \left\lfloor \dfrac{x}{y} \right\rfloor$，所以只需考虑如何优化除法，取模可以再做一次乘法和减法得到。

除法优化是基于一个引理：

- 设 m, d, l 是非负整数且 $d \neq 0$，满足 $2^{N+l} \leq md \leq 2^{N+l} + 2^l$，则对于所有 $0 \leq n < 2^N$ 的所有整数 n，有 $\left\lfloor \dfrac{n}{d} \right\rfloor = \left\lfloor \dfrac{nm}{2^{N+l}} \right\rfloor$。

如果我们能找到一个 l 和对应的 m，就可以将除法优化为一次乘法和一次右移运算，这利用了乘法可以得到高 32 位结果这一特点。

对于一个给定的 l，m 的取值范围是 $\left[\left\lceil \dfrac{2^{N+l}}{d} \right\rceil, \left\lfloor \dfrac{2^{N+l} + 2^l}{d} \right\rfloor \right]$，取 $l = \lceil \log_2 d \rceil$，不断减小 l 直到上述区间中只有一个整数，这样可以得到最小的 m。

如果此时 $m < 2^{32}$，使用无符号乘法指令 `multu` 就可以完成上述的任务，取 $N = 32$，将 nm 的高 32 位右移 l 位就可以得到答案。

如果此时 $m \geq 2^{32}$，可以通过一系列分类讨论和乘法分配律规避掉值域过大的问题，鉴于复杂度和时间原因，我并未进行这种情况的优化。

需要注意负数除法需要特判，但是本质并没有改变。

3.11.3 取模优化

尽管可以用上述方法优化取模，不过有一个更加常用的优化，如果 $|x| < y$，则 $x \bmod y = x$，因此可以用一条比较指令和一条跳转指令的代价使得有一定几率优化掉取模。

事实上，这个优化非常显著，因为一般大量取模是用在模意义下的运算，那么很容易就出现小于模数的情况。算法竞赛中也经常使用这个技巧优化加法取模来卡常。

3.12 窥孔优化

3.12.1 循环翻译优化

文法中仅存在 while 循环，一种可行的翻译结构如下：

```
label1:
if !<Cond> goto label2
<Stmt>
```

```
    goto label1
    label2:
```

那么 n 次循环就需要 $O(n)$ 次跳转和 $O(n)$ 次分支。

可以将其为如下形式：

```
    if !<Cond> goto label2
    label1:
    <Stmt>
    if <Cond> goto label1
    label2:
```

n 次循环就需要 $O(1)$ 次跳转和 $O(n)$ 次分支。

最重要的是，第二种形式将循环条件单独提了出来，通过上述的全局公共子表达式合并就可以实现循环不变量外提的效果！

3.12.2 指令选择优化

MARS 支持一些扩展指令，例如常用的 `li` 和 `move`，但有一些指令 MARS 的翻译比较愚蠢，此时手动翻译可以用更少的指令将其取代。

例如 `sltu` 这个指令用来判断一个数是不是 0，因为无符号比较下 0 是最小的。因此可以用 `xor` 和 `sltu` 两条指令取代扩展指令 `sne`，用 `xor` 和 `sltu` 和 `xori` 三条指令取代扩展指令 `seq`。

此外有一些 MIPS 原生指令是非常强大的，比如 `mul` 可以直接将低 32 位结果放到一个寄存器里，不需要再去 `mflo` 了。

4 总结

从头到尾经历了一遍编译器的设计和编写，深刻体会到了编译工作者们的艰辛！

作业总体可以分为文法解读、词法分析、语法分析、错误分析、生成中间代码和目标代码、代码优化六个部分。前面四个部分可能只花了几个小时，生成中间代码和目标优化花了一天多，最后的代码优化只留了一周多去做，被迫成为 ddl 战神极限冲刺，可见难度和复杂性的陡然增大。

每个部分几乎课程组都通过整理第一部分我们提交的样例提供了数十个测试样例库，以方便我们去 debug，毕竟编译器的测试样例不太好自动化生成。这一点体验比较好，绝大部分 bug 都能被样例库覆盖。

评测机可以无限制提交，不过需要等上一次测完才能交，最后代码优化部分评测过慢这可能是由于 MARS 的低性能造成的，或许专门写一个用于评测的高性能程序 (MIPS 模拟器) 可以有更好的体验。

此外，还有几点希望课程组可以改进：

- 错误处理部分最好可以形式化定义地给出可能出现的错误，目前的题面描述还是过于模糊，很多不通过讨论区和特判根本想不出来。而且以目前题面的定义还有很多难以判断的错误，然而因为测试样例没有对应的情况所以不用考虑，这未免过于面向样例编程了。
- 代码优化部分最好可以增加一些样例，现在只有六个样例，显然不能考察每种优化算法。此外，有时候加一个优化算法就能让一个测试点指令数少很多，很明显这个测试点是构造过的，然而工程上不会有人这么写代码，比拼谁能猜中样例的构造方法应该不是课程中想考察的目标。因此增多尽可能多的正常代码作为竞速样例是非常有必要的。
- 最好可以删除竞速，而是设置 baseline，即只要优化效果超过 baseline 就可以获得满分，现在内卷化过于严重，设置竞速无异于鼓励内卷。事实上学习各种各样的优化算法是有趣的一件事，但当最终目标变成取得更高排名时，一切都变得索然无味了。如果课程组想要提高区分度，完全可以将 baseline 设高一点，或者像 OS 那样添加其它挑战性任务，例如添加文法对指针、堆等的支持，而不是只有竞速一条路。

最后，当我写完这篇一万字的申优文档，深感相关知识体系的庞大，以及编译相关理论的学无止境。感谢课程组对编译实验的建设，希望编译实验能越来越好！

参考文献

[1] Andrew W.Apple: *Modern Compiler Implementation in Java, 2nd*,

[2] SuperSodaSea: `https://zhuanlan.zhihu.com/p/151038723`