编译技术实验参考（2020）

代码生成 代码优化

时间：2020 年 9 月

版本：0.8

Go!Go!Go!

目　　录

第 1 章 生成中间代码

经历了重重分析，我们终于要将源程序进行第一步转化了，那就是“生成中间代码”。其实，生成中间代码也是伴随着语法分析和语义分析进行的，我们可以这么理解，我们的编译器的工作可以大致分为两部分，第一部分是通过语法分析和语义分析，将源程序翻译为我们自己定义的中间代码（在课设中推荐用四元式的形式）；第二部分是通过分析中间代码，将其翻译为最终的目标代码。

也就是说，目标代码的生成，是完全基于中间代码的。所以，中间代码的正确性以及高效性，将对目标代码的正确性以及效率造成很大影响。

1.1 一种中间代码的生成方式——四元式

此处我们介绍一种中间代码的形式——四元式。书上是这么介绍四元式的：四元式是 N-源表示的一种，四元式的每条指令有 4 个域：

< 操作符 >, < 操作数 1>, < 操作数 2>, < 结果 >

其中，< 操作数 1> 和 < 操作数 2> 分别表示第一个操作数和第二个操作数，< 结果 > 表示计算的结果，该结果通常是一个临时变量，编译程序可以为该变量分配一个寄存器或一个主存地址。

但私以为这样并不是十分好理解，遇到某些语句也并不能严格按照 < 操作数 1> 和 < 操作数 2> 来翻译，因此笔者认为把四元式结构改成：< 操作符 >，<label1>, <label2>, < 结果 > 更好理解一些。就像是前文提到的符号表一样，可能不同人所设计的数据结构并不相同，有的记录了 4 个信息，有的记录了个 5 信息，但只要能够保证可以通过符号表查找到自己所需要的全部信息即可。四元式也是如此，只要通过两个 label，能够准确翻译源程序，生成正确的中间代码即可，至于这两个 label 是否都用到了，不必太过死板。比如说下面三条语句：

```
(1) a = b;
(2) a = b + c;
(3) a = num[3];
```

对于第一条语句，不难理解生成的四元式应当是：

=, b, , a

对于第二条语句，首先我们要计算 b + c 的值，然后将计算结果赋值给 a，则生成的四元式应当是：

+, b, c, t0

=, t1, , a

对于第三条语句，首先我们要将相对于 num 这个数组的首地址偏移量为 3 的内存中的值取出，然后将数值赋值给 a，则生成的四元式应当是：

[], num, 3, t1

=, t1, , a

可以看出，第一条语句生成的四元式中，我们只使用了三个信息，有一个 label 并没有使用；第二条语句中，“+, b, c, t0”是一个标准的“< 操作符 >, < 操作数 1>, < 操作数 2>, < 结果 >”格式；而第三条语句中，取 num[3] 的值所对应的四元式“[], num, 3, t1”则既使用了两个 label，又不能说 num 和 3 都是操作数。

当然，四元式具体如何设计，或者说是否要使用四元式，这都取决于编译器的作者。只要生成的中间代码可以准确无误地表达源程序的逻辑，并且便于优化，就是好的中间代码。

对于课程设计要求的 C0 文法来说，下面简单分析一下在生成中间代码阶段的各类语句（仅仅是笔者的浅见，完全可以自行设计优化）：

- 条件语句：文法中的条件语句包括 if-then 和 if-then-else 两种结构。对于这两种结构的设计与书中 10.6 章节类似，都是先计算出 if 条件是否为真，若条件为真，则执行 if 后的语句（需要注意的是，若存在 else 分支，则 else 分支的语句在 if 后的语句执行完毕后就不必执行了，可以通过在 if 语句最后加入一个跳转至 else 分支的语句之后的中间代码来实现）；若条件为假，则执行 else 分支的语句（存在 else 分支）或退出该条件语句；
- 循环语句：文法中的循环语句包括 while 和 for 两种结构。与条件语句类似，这两种结构也是通过在不同条件下实现不同的跳转来实现的，就不过多阐述了；
- 函数调用语句：对于函数调用语句，在生成中间代码时可以通过将先将参数压入栈中，在调用相应的函数来实现；
- 赋值语句：即先计算赋值号右边表达式的值，在将其赋给赋值号左侧的变量；

而对于读语句、写语句、返回语句、语句列等语义是简单明了的，也可以直接生成相应的中间代码，就不做分析了。

下面给出一个函数调用语句可能存在问题的示例，可自行思考一下：

```
int sum3(int a, int b, int c){
    return(a + b + c);
}
void main(){
    printf(sum3(sum3(1, 2, 3), sum3(4, 5, 6), sum3(7, 8, 9)));
}
```

下面给出一个程序段以及生成的中间代码，但只是个人设计一定存在许多不足，在此只作为例子方便同学们更好的理解。

例如有如下程序段：

```
const int const1 = 1, const2 = -100;
const char const3 = '_';
int change1;
char change3;
int gets1(int var1,int var2){
    change1 = var1 + var2;
    return (change1);
}
void main(){
    printf("Hello World");
    printf(gets1(10, 20));
}
```

则生成的中间代码为：

```
const int const1 = 1
const int const2 = -100
const char const3 = '_'
var int change1
var char change3

int gets1()
para int var1
para int var2
+        var1     var2     t0
=        t0                change1
```

```
=         change1           t1
Return    t1
Return    -1

void main()
Printf    string0
Push      10
Push      20
Call      gets1
RetValue  t0
Printf    t0
```

希望大家都可以设计出自己用着最顺手的中间代码，这样会有利于后期生成目标代码以及做对编译器的优化！

1.2 评测时中间代码输出约定

原则上按照中缀表达式格式输出中间代码，即，形如 x = y op z，其中 x 为结果，y 为左操作数，z 为右操作数，op 为操作符。以下根据基本语法现象举例说明。

1. 函数声明源码形如：

```
int foo( int a, int b, int c, int d)
```

中间代码：

```
int foo()
para int a
para int b
para int c
para int d
```

2. 函数调用源码形如：

```
i = tar(x,y)
```

中间代码：

```
push x
```

```
push y
call tar
i = RET
```

3. 函数返回源码形如：

```
return (x)
```

中间代码：

```
ret x
```

4. 变量声明源码形如：

```
int i, j;
```

中间代码（符号表信息输出，程序中可不生成真正的中间代码）：

```
var int i
var int j
```

5. 常数声明源码形如：

```
const int c = 10
```

中间代码（符号表信息输出，程序中可不生成真正的中间代码）：

```
const int c = 10
```

6. 表达式源码形如：

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

中间代码（可优化）：

```
t1 = b + c
t2 = a * t1
x = t2
```

7. 条件判断源码形如：

```
x == y
```

中间代码：

```
x == y
```

8. 条件或无条件跳转中间代码：

```
GOTO LABEL1 //无条件跳转到LABEL1
BNZ LABEL1 //满足条件跳转到LABEL1
BZ LABEL1 //不满足条件跳转到LABEL1
```

9. 带标号语句源码形如：

```
Label_1: x = a + b
```

中间代码：

```
Label_1 :
x = a + b
```

10. 数组赋值或取值源码形如：

```
a[i] = b * c[j]
```

中间代码：

```
t1 = c[j]
t2 = b * t1
a[i] = t2
```

11. 其他本文档未涉及到的语法现象，或者程序员自行定义的四元式操作，原则上均按照“x = y op z”形式的中缀表达式进行表达。

请提交语法分析程序的源代码,要求将编译产生的中间代码输出到命名为"output.txt"的结果文件中。对于中间代码的具体设计不做统一要求，但应符合四元式的要求。

第 2 章 生成目标代码

代码生成是编译的最后阶段，它通常以编译器此前生成的中间代码、符号表及其他相关信息作为输入，输出与源程序语义等价的目标程序代码。在之前的阶段中生成了中间代码后，便可以进行生成目标代码的工作了。可以说生成目标代码最简单的实现方式就是，对每一条四元式中间代码，可以设计它的代码骨架，它将给出为这种语句生成的目标代码的轮廓。

如上一章中提到的例子，依据：

```
(1) a = b;
(2) a = b + c;
(3) a = num[3];
```

生成的中间代码：

（1）

```
=, b, , a
```

（2）

```
+, b, c, t0
=, t1, , a
```

（3）

```
[], num, 3, t1
=, t1, , a
```

就可以一一对应的翻译为：

（1）

```
mov ax, b
mov a, ax
```

（2）

```
mov ax, b
mov dx, c
```

```
add ax, dx /* 将ax和dx寄存器的值相加并写入ax寄存器中 */
mov a, ax
```

（3）

```
mov bx, offset num /* 获取num在内存中的偏移量 */
mov ax, 3
shl ax, 1 /* 将ax寄存器中的值左移一位 */
add bx, ax
mov ax, [bx] /* 读取内存中偏移量为bx的字 */
mov a, ax
```

2.1 目标代码生成阶段需要完成的一些功能

2.1.1 存储管理

若要生成目标代码，则需要将中间代码中的变量、数组等都映射为它们在内存中的位置。这一部分功能的实现，需要从符号表中找到相应的变量和数组在内存中的相对地址。可以针对四元式中的变量、数组，分别计算出其在内存中的相对地址。而对于函数调用等，可以通过 jal 指令来实现（需要注意对返回地址的保存）。

语言中过程的语义决定了执行期间名字如何与存储空间相联系。过程的一次执行所需要的信息存放在一个被称为活动记录的存储块中，过程中的局部变量的存储空间也在活动记录中。

一个过程的活动记录包含存放下列信息的域：参数、结果、机器状态信息、局部数据、临时变量等等。不是所有的语言，也不是所有的编译器都使用所有这些域。像 Pascal 和 C 这样的语言的习惯做法是，在过程被调用时把它的活动记录压入运行栈，在控制返回调用者时把这个活动记录从栈弹出。

我们知道有两种标准的存储分配策略，即静态分配和栈式分配。对于静态分配，内存中的活动记录在编译时便已确定。对于栈式分配，过程的每次执行都将在栈顶压入一个新的活动记录，活动结束时将该记录弹出。

对于简单的类 C 文法来说，运行时活动记录的分配和释放是作为函数调用和返回序列的一部分，也即主要包括：（1）call（调用），（2）return（返回）。同时还需要思考函数被调用时如何传递参数。

下面给出一个运行时存储管理值得思考的样例：

```
int sum3(int a, int b, int c){
    return(a + b + c);
}
void main(){
    int a, b, c, d, e, f;
    a = 1;
    b = 2;
    c = 3;
    d = 4;
    e = 5;
    f = 6;
    printf(sum3(sum3(a, b, c), sum3(7, 8, 9), sum3(d, e, f)));
}
```

在 Pascal 和 C 这样的语言中，当调用出现时，一个活动的执行被中断，有关机器的信息，如程序计数器和机器寄存器的值就保存在栈中。当从调用中返回时，在恢复了有关寄存器的值和把程序计数器的值置为紧接该调用的下一个点后，该活动能继续执行。具体来说，对于生成 MIPS 指令作为目标代码的编译器来说，可以选择使用 fp 寄存器来指向堆栈里的过程帧（一个子函数）的第一个字，子函数可以用其做一个偏移访问栈帧里的局部变量（这只是一种方案而不是要求，完全可以自行设计优化）。而在函数调用及返回时，可以通过更改 fp 寄存器的值来实现活动记录的入栈和出栈。

2.1.2 指令选择

如果不考虑目标程序的效率问题，那么指令的选择是直截了当的。对每一条四元式中间代码，可以设计它的代码骨架，它将给出为这种语句生成的目标代码的轮廓。例如，若 x、y、z 都在数据段中直接指定其在内存中的地址，则可以将每个形如 $z \rightarrow x + y$ 的四元式翻译为如下代码序列：

```
mov ax, x /* 将x装入ax寄存器中 */
mov dx, y /* 将y装入dx寄存器中 */
add ax, dx /* 将ax、dx相加 */
mov z, ax /* 将结果存入z中 */
```

然而，这种逐条语句的代码生成方法常常产生质量较低的代码。例如对于指令序列

$$a \rightarrow b + c$$

$$d \rightarrow a + e$$

将被翻译为：

```
mov ax, b
mov dx, c
add ax, dx
mov a, ax
mov ax, a
mov dx, e
add ax, dx
mov d, ax
```

我们可以看出其中的第 5 条指令是冗余的。若以后 a 不会再被用到，那么第 4 条指令也是冗余的。

所选择的指令同时还影响了对同一条中间代码的执行速度和长度。指令集丰富的目标机器可能提供几种办法实现某一操作，由于不同的实现方法的开销可能不大一样，因此中间代码的简单翻译会产生正确但效率可能较低的目标代码。如对于四元式 $a \rightarrow b*2$，翻译成：

```
mov ax, b
shl ax, 1 /* 将ax寄存器中的值左移一位 */
mov a, ax
```

显然比：

```
mov ax, b
mov dx, 2
mul dx /* (dx:ax) := ax * dx */
mov a, ax
```

具有更高的效率。

2.1.3 寄存器分配

在不考虑优化的情况下，完全可以将所有变量都存储在内存中，寄存器仅暂存变量的值，操作结束后将结果写入内存中的相应位置。但是操作数在寄存器中的数据通常要

比操作在内存中的数据具有更高的效率。充分利用寄存器以生成好的代码就显得尤为重要了。寄存器的使用可以分为两个子问题：

1. 在寄存器分配期间，在程序的某一点选择要驻留在寄存器中的变量集；
2. 在随后的寄存器指派阶段，调出变量将要驻留的具体寄存器。

而选择最优的寄存器指派方案是非常困难的，只能选择一个相对较好的方案。

2.2 MARS 仿真器使用说明

如下是对选择生成 MIPS 代码的同学，编译课程设计 MARS 仿真器使用说明。

1. MARS 使用 4.5 版本（https://courses.missouristate.edu/KenVollmar/MARS/）；
2. 编译器生成代码可以选择基础指令及伪指令，不能选择宏指令；
3. 关闭延迟槽；
4. 内存配置为默认选项；
5. 寄存器按照表2.1说明使用。

表 2.1: MARS 寄存器使用说明

REGISTER	NAME	USAGE
$0	$zero	常量 0
$1	$at	保留给汇编器
$2-$3	$v0-$v1	函数调用返回值
$4-$7	$a0-$a3	函数调用参数
$8-$15	$t0-$t7	临时寄存器
$16-$23	$s0-$s7	全局寄存器
$24、$25	$t8、$t9	临时寄存器
$28	$gp	全局指针 (Global Pointer)
$29	$sp	堆栈指针 (Stack Pointer)
$30	$fp	帧指针 (Frame Pointer)
$31	$ra	返回地址 (return address)

2.3 评测时目标代码输出约定

选择生成 MIPS 汇编的同学，应将 MIPS 代码输出至 mips.txt 文件中，并确保 MIPS 代码能在 MARS 上成功运行。

选择生成 PCODE 代码的同学，应直接在编译完成后立即执行输入 testfile.txt 中的测试代码。也即对形如：

```
void main(){
    printf("Hello World!");
}
```

的测试代码，编译器在“pcoderesult.txt”文件输出"Hello World!" 即可。

第 3 章 代码优化

代码优化模块通常在代码生成模块之前被调用，在不改变程序原有语义的前提下，为代码生成模块提供优化后的中间代码作为后者的输入。代码优化阶段位于代码生成阶段之前，但即使没有代码优化模块，一个具有代码生成模块的编译器仍是可工作的，反之则不然。

代码优化阶段的目的包括改进中间代码，以生成执行速度较快的机器代码。在前几个阶段后已经可以生成一个能够运行出正确结果的 MIPS 代码了，而代码优化部分各个人选择的方案都不尽相同，且笔者水平有限，不会过多进行讲解，希望自行设计。

在我们这门课程中，代码优化阶段要求完成的优化包括基本块内部的公共子表达式删除（DAG 图）、全局寄存器分配（引用计数或着色算法）、数据流分析（通过活跃变量分析，或利用定义-使用链建网等方法建立冲突图）等，而对于其他优化，例如复制传播、循环强度削弱等，可以自行选作，成功完成都有加分。

3.1 一些常用的优化方案

课本中介绍的代码优化方法包括数据流分析、常量传播、公共子表达式删除、死代码删除、循环优化、代码内联等。下面按照实现顺序介绍一些可用的优化方案，以及笔者当初实现时的一些想法（仅供参考）。

3.1.1 常量传播和死代码删除

按照实现顺序来说，笔者首先进行的是常量合并和传播。在语义分析阶段，分析表达式时就将能计算出来的常量都计算出来，可以减少目标代码中的重复计算。同时，可以判断一些条件跳转指令是否永真或永假，可以对无法进入的分支进行死代码删除。如以下代码：

```
if(2019 > 9012){
    printf("Hello World!");
}
```

可以看出其中的 printf 语句是不可达的，可以删去。

3.1.2 基本块和流图

在之前一定程度上减少了程序中可能存在的分支之后就可以进行基本块的划分了。书中对于基本块的定义如下：

1. 基本块中的代码是连续的代码序列；
2. 程序的执行（控制流）只能从基本块的第一条语句进入；
3. 程序的执行只能从基本块的最后一条语句离开。

在了解了基本块的定义后就要思考如何划分出各个基本块了。书中按照

1. 整个语句序列的第一条语句属于入口语句；
2. 任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句；
3. 紧跟在跳转语句之后的第一条语句属于入口语句

来作为划分基本块的方法。课本中还强调正常的函数调用并不意味着“程序执行的离开”，因为调用完成后程序仍会继续执行后继的指令。但是，如果结合实际应用考虑，在建 DAG 图的时候，需要把整个基本块内的代码全部放入 DAG 图中。为了实现的方便，故我在这里将函数调用也作为划分基本块的标志了。

在划分基本块后还需要进行建立流图的工作，按照书中的做法进行就可以了。

3.1.3 活跃变量分析

在流图建立之后就可以进行活跃变量分析了（笔者认为也可以在消除公共子表达式结束之后再进行，但在这里进行可以先减少一定的工作量）。关于活跃变量分析，按照书中给出的做法完成就可以了。

如果变量的值以后还要被引用，则称它在程序该点是活跃的，否则称它在该点是无用的。其相关概念是无用代码，即其计算的值不会被引用的语句。如以下示例：

```
void main(){
    int i, a;
    for(i = 0; i < 10; i = i + 1){
        a = 10;
    }
    a = 6;
    printf(a);
}
```

我们可以看出在 for 循环语句中对 a 的赋值就是无用代码。对于程序中的无用代码，我们可以从代码中删除。

3.1.4 消除公共子表达式

在前一步骤完成后，就可以进行建立 DAG 图和从 DAG 图重新导出中间代码，实现公共子表达式删除了。

有向无环图（DAG）是实现基本块变换的一种非常有用的数据结构。DAG 图说明了基本块中每一个语句计算的值是如何被本块中的后继语句引用的。

基本块的 DAG 是一种其节点带有如下标号的无环有向图：

1. 叶节点由唯一的标识符所标记，即变量名或常量。根据作用到名字上的操作符可以确定需要的是名字的左值还是右值。叶节点代表名字的初始值，为了避免与指示名字当前值的标号相混淆，我们给这些叶节点加上下标 0。
2. 内部节点用操作符符号标记。
3. 节点还可能为标号附以一系列标识符，目的是用内部节点表示已经计算的值，而且标记节点的标识符也具有该值。

书中对于基本块的介绍是比较简单的，一些较为复杂的情形未给出具体实现建议，包括 IO 指令、数组存取指令、函数调用等。关于这些情况的讨论，我们在这里可以给出一些实现的建议。

3.1.4.1 IO 指令

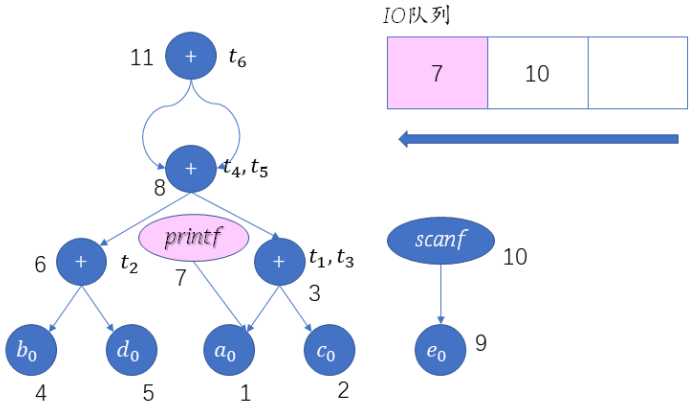

图 3.1: IO 节点队列

可以将输入输出指令加入 DAG 图中，但是，由于从 DAG 图导出代码的启发式算

法中，中间结点的选取是随机的，将 IO 指令直接放入 DAG 图中后，并不能保证导出的 IO 指令是有序的，从而造成执行结果的不同，为此可以使用 FIFO 策略对启发式算法进行改进。改进的启发式算法的主要思路依然是尽量选取左子结点，但是不同之处在于，如图3.1所示，维护了一个 IO 结点队列，建图时 IO 指令结点号入队，导出中间代码时 IO 指令结点号出队。当选取的中间结点是 IO 指令时，需要满足其结点号与 IO 队列队首元素相同，方能被选中。

3.1.4.2 数组存取指令

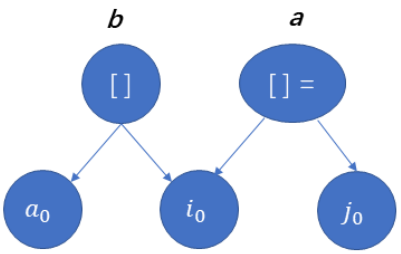

图 3.2: 数组操作的 DAG 图

关于数组存取的问题，书中 14.2 节其实已经提到了，可以保守地认为对数组的赋值改变了数组中的各项，而不对其进行优化。但仍需要考虑正确性的问题，例如对如下指令序列

```
b = a[i]
a[i] = j
```

将生成如图3.2所示的 DAG 图。由于两个中间结点都没有父结点，按照启发式算法，导出的代码顺序是任意的。这里为了保证正确性，可以与 IO 指令类似，设计一个数组存取指令队列，存取数组的指令也按照 FIFO 的原则导出，严格有序。

3.1.4.3 函数调用的问题

由于之前已将函数调用作为划分基本块的标志，每个基本块内最多仅会有一个函数调用。完全可以按照传入参数的顺序生成一个函数调用的 DAG 图，在 DAG 图的其余部分都导出为中间代码之后，再将函数调用导出为中间代码，以保证正确性。

3.1.5 寄存器分配和指派

我们在生成目标代码一章中提到，操作数在寄存器中的数据通常要比操作在内存中的数据具有更高的效率。充分利用寄存器以生成好的代码就显得尤为重要了。寄存器的使用可以分为两个子问题：

1. 在寄存器分配期间，在程序的某一点选择要驻留在寄存器中的变量集。
2. 在随后的寄存器指派阶段，调出变量将要驻留的具体寄存器。

书中介绍了引用计数和图着色算法两种常用的全局寄存器分配方法。

在这里笔者选用的是图着色算法实现的全局寄存器分配。笔者简单的利用了之前活跃变量分析阶段的结果建立冲突图，并利用书中介绍的图着色算法。但笔者一开始忽略了分配的寄存器和活跃范围之间的关系。如对以下代码

```
void main(){
    int i, j, a, b;
    for(i = 0; i < 10; i = i + 1){
        // 1 begin
        for(j = 0; j < i; j = j+1){
            a = i * j;
        }
        printf(a);
        // 1 end
    }
    for(i = 0; i < 10; i = i + 1){
        // 2 begin
        for(j = 0; j < i; j = j+1){
            b = i + j;
        }
        printf(b);
        // 2 end
    }
    for(i = 0; i < 10; i = i + 1){
        if(i / 3 = 2){
            // 3 begin
            a = i * j;
            b = i + j;
```

```
            printf(a + b);
            // 3 end
        }
    }
}
```

我们可以看出，变量 a、b 都是跨越基本块活跃的变量，且它们并不冲突。假设我们将 a 和 b 指派了同一全局寄存器，若认为 a 和 b 在第三块（不代表基本块的划分）中仍指派了该全局寄存器，则会导致 BUG 的出现。这里应该认为对变量 a、b 指派该寄存器的范围分别是第一块和第二块内。

3.1.6 窥孔优化

逐条语句进行的代码生成策略经常产生含有大量冗余指令和次优结构的目标代码。通过对这些目标代码进行“优化”转换可以提高这种目标代码的质量。有许多简单的转换可以显著地改善目标程序的运行时间和空间需求，所以知道哪种转换在实际中是非常重要的。

窥孔优化是一种简单有效的局部优化方法，它通过检查目标程序的短序列（称为窥孔），并尽可能用更小更短的指令序列代替这些指令以提高目标程序的性能。尽管我们将窥孔优化作为改善目标代码质量的技术，它也可以直接用在中间代码生成之后以提高中间代码的质量。

下面给出一些窥孔优化的实例。

3.1.6.1 冗余加载与保存

如以下指令序列：

```
mov ax, b
mov dx, c
add ax, dx
mov a, ax
mov ax, a
mov dx, e
add ax, dx
mov d, ax
```

可以看出其中第 5 条指令是冗余的。

3.1.6.2 强度削弱

用在目标机器上开销小的指令代替开销大的指令，如以下指令序列：

```
mov ax, b
shl ax, 1 /* 将ax寄存器中的值左移一位 */
mov a, ax
```

显然比：

```
mov ax, b
mov dx, 2
mul dx /* (dx:ax) := ax * dx */
mov a, ax
```

具有更高的效率。

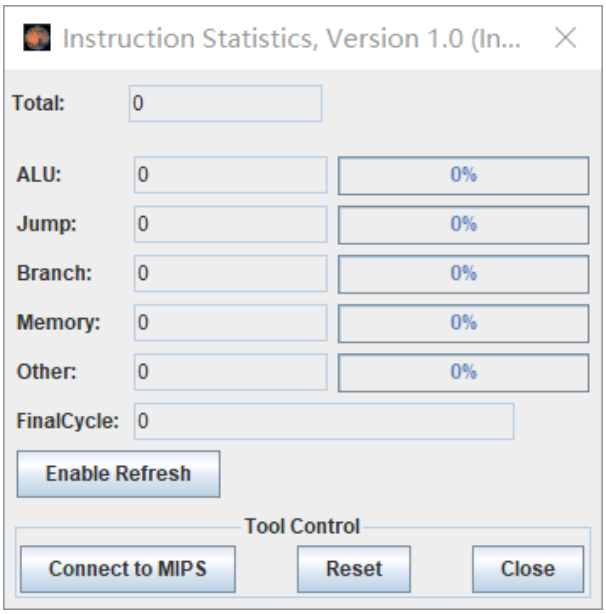

图 3.3: Instruction Statistics 对话框

3.2 编译课程设计竞速排名要求

首先需要满足目标代码生成阶段对 MARS 仿真器的使用要求，在此基础上进行竞速排名。竞速排名根据程序运行后的统计信息，加权计算后排名。在 Tools 菜单选择 Instruction Statistics 后弹出如图3.3的对话框。其中 Total = ALU + Jump + Branch + Memory + Other，将其按照 ALU 权重 1，Jump/Branch 权重 2，Memory 权重 2，others 权重 1 的比例重新计算后，得到 FinalCycle = ALU * 1 + Jump * 2 + Branch * 2 + Memory * 2 + Other * 1 进行排名。在运行正确的前提下，FinalCycle 越小排名越靠前。违反上述要求的，取消竞速排名资格。

注意 这里提供的 FinalCycle 的计算公式仅为举例，不代表最终的计算方法。请以平台最终公布的计算方法为准。

3.3 结束语

至此，编译器课程设计就算告一段落了，本实验参考也接近结束了。

最后一点点碎碎念，一学期过去，也许你觉得编译课设让你很痛苦；也许你觉得这与你未来从事的方向相差甚远，并无用处，但编译原理一定会帮助你对计算机有了更深刻的理解和认识。

2020 年，编译课程改革已经进入第二个年头了。今年首次加入了同学们提交测试文件再基于同学们的提交完成生成测试程序库的工作、将生成 PCODE 和 MIPS 的编译器在代码生成阶段使用同一套评测流程进行考察都是课程新的尝试。

本参考的作者也仅是将自己在完成小编译器过程中的经验和犯过的错误进行归纳总结，结合自己对编译浅薄的认识，完成了本参考的撰写。在参考中，由于笔者水平有限，难免会让同学们有浅尝辄止的感觉，也难免会有一些纰漏，还请大家谅解。

最后的最后，编译课程组非常欢迎学弟学妹们加入到我们中来，一起为课程改革添砖加瓦。