编译课设申优文档

17373508 刘泽华

漫长的编译实验终于结束，所谓优化，原来就是不断地把正确程序改错，然后再去改 bug 的过程。不过，既然选择了最高难度，便只顾风雨兼程。那些嘴上说着不优化最后却真香的大佬，通过排行榜才能发现原来身边的同学都是魔鬼(×)。

那么废话不多说，我们进入正题，和大家分享下我在完成编译课设的过程中遇到的问题和一些解决方案。（如果你已经迫不及待可以直接翻到最后来一起探讨优化方案）

补充于 2019.12.15：

很尴尬的在这里写下这样一段，虽然可能有些残忍，但仍要指出来。如果你决心冲到前 20%，那就几乎要把所有的优化都做了，不要小看你身边同学的野心。本人就是这样一个例子，本以为可以收手了，谁成想大家在最后都想着了魔一样……（或者本来就是有的大佬在藏分隐藏实力）迫不得已，我连申优文档都写好了，被挤出前 20%也太惨了吧。所以在 ddl 前的最后一天又绞尽脑汁补充了许多优化，成绩又有了些提升也算是让我心里好受点了……

整体架构

c0 编译器采用我们熟悉的 C 语言实现，当然也支持 C++。由于最初对 C++的使用没有足够的自信，因为毕竟还是对 C 了解更多，所以许多地方仍是通过 C 的语法来实现。当然，这里不得不说一句，C++的数据结构真的好使。比如 map，vector，stack……

但我们都知道 C 语言最大的风险就在于**指针问题**，而这次我也毫无例外地遇到了这个问题。我们都知道，vector 是变长数组，那么当它现有的存储空间满足不了数组长度时，它会在内存中重新分配一块新空间，而数组的存储要求是一块连续的空间，变长数组也是数组，这就意味着重新分配的空间可能和之前的空间没有任何重叠，即如果你用指针指向 vector 数组元素的地址的话，那么由于

它每次增加都在变化，所以你不知道什么时候你指向 vector 元素的指针就变成野指针了。

根据课程作业安排，我们的编译器实现分成了以下几个阶段：文法解读，词法分析，语法分析，错误处理，代码生成，代码优化。前面几个单元的部分较为简单，重点在熟悉语法，同时确定大体的程序框架，不要最后连自己都看不下自己写的程序了。

最后提一句编译器，之前没有用过 VS，不过使用后体验挺不错的。

词法分析

词法分析较为简单，完成的任务就是将输入的程序拆分成一个个单词，并分析出单词的类型，那么直接都用数组保存起来就可以了。这部分较为简单，不再赘述。

有个小 tip 是在后面代码生成阶段，由于课程要求对要输出的**字符串原样输出**，不转义。这句话的意思其实是在告诉我们要我们提前分析出要转义的成分然后转义，在和同学交流后发现，将转义的工作放在词法分析实现非常简单，即在分析单词时，当遇到反斜杠字符时，就在反斜杠字符后面紧接着加一个反斜杠字符即可。

语法分析

语法分析阶段通过识别第一阶段的各个词法成分，识别出各类语法成分。

识别 FIRST 集有相交的语法成分

要求我们实现的文法中，许多文法的 FIRST 集合之间有相交，按照我们理论课学习的知识，一种方法是改写文法，这是实现不回溯的递归下降子程序法的前提条件，另一种是预读和回溯分析。由于有些文法改写较复杂，因此除了少部分简单的直接改写文法外，我的语法分析中还是采用了预读和回溯的方法。具体来说就是在进入一个新的语法成分分析前，设置一个指向当前状态的指针，然后递归下降分析，如果分析不下去就回溯，还原指针位置，然后重新分析下一种可能的语法成分。如果能直接通过预读来判断语法成分类型，那就直接往下读一个

单词。比如对有返回值函数定义和变量定义，在标识符后面有无小括号是区分它们的重要标志。

<有返回值函数定义>::=<声明头部>'('<参数表>')'……

<变量定义>::=<类型标识符><标识符>……

表达式类型

表达式类型有 int 型和 char 型两种。课程组规定的 char 类型只有以下三种情况。

（1）表达式由<标识符>或＜标识符＞'['＜表达式＞']'构成，且<标识符>的类型为 CHAR，即 CHAR 类型的常量和变量、CHAR 类型的数组元素。

（2）表达式仅由一个<字符>构成，即字符字面量。

（3）表达式仅由一个有返回值的函数调用构成，且该被调用的函数返回值为 CHAR 型

由于表达式由项相加而成，项由因子相乘而成，因子中为 char 类型的是单个字符因子、字符型标识符或字符型数组元素，或者返回值类型为 char 的函数调用返回值，而且表达式只能由这些单个因子组成，即在 char 型表达式的情况下只有：<表达式>=<项>=<因子>（三选一）。

符号表管理

符号表管理这一部分十分关键，符号表的建立对语法分析、错误处理和代码生成都起着非常重要的作用。在语法分析阶段建立我首先建立起一张栈式符号表，后面由于需要又单独建立一张函数符号表。

管理函数

在不涉及中间代码生成基本块的概念前，我们所有的操作都是建立在以函数为单位基础上的，因此将各个成分与每个函数关联起来方便查找与管理，建立函数表的思路就确定了。

那么函数表中要包括些什么内容呢？在我的函数表中，首先有这个函数定义的各个参数、变量和常量信息，甚至可以有在中间代码生成阶段的中间变量信息，也包括它的返回值类型等信息。（反正原则就是你需要什么往里加就好了，内存够不够的问题不需要你考虑）

```
typedef struct node_fun
{
    FunType type;
    std::vector<symNode> paraName;
    std::vector<symNode> consName;
    std::vector<symNode> varName;
    std::vector<symNode> tmpName;
    int size;
    bool callFunction;     // leaf function
    bool inlineFunction;   // inline
    int getPosition(std::string name);
    SymKind getKind(std::string name);
    SymType getType(std::string name);
    symNode getNode(std::string name);
} funNode;
```

错误处理

错误处理这一部分我的观点是不要想太复杂，有些问题可能复杂起来真的不好分析。不确定的地方在答疑区提问，找到一个明确的回复可以减轻自己工作量。错误处理阶段我认为还是关于函数的错误处理和其他相比有些复杂。

函数相关的错误处理

1. 函数参数个数不匹配
2. 函数参数类型不匹配

这两个错误是和函数的传参相关的。

我刚开始的做法是在分析<值参数表>语法成分时，一项一项匹配参数类型，同时看能能不能匹配到相应的参数个数。后面分析时发现，可以换种更简单的做法。上面我们已经提到，在一个函数中定义的所有参数都已经保存在函数的结构体中，那么我们在分析值参数表的语法成分时，只需要将每个值参数先都保存起来（比如用 vector），然后再相应地去比对参数信息和保存的值参数信息，包括类型和个数。

3. 无返回值的函数存在不匹配的 return 语句

4. 有返回值的函数缺少 return 语句或存在不匹配的 return 语句

这两种错误是和函数返回相关的。首先课程组在这里有个简化，即

> *不需要检查每个分支是否有返回语句，一个有返回值的函数，只要有一条返回了值的返回语句就算对。测试程序不会出现某分支没有返回值的情况。*

由于我们报错时同时要输出对应的行数，所以对于不匹配的 return 语句要在检测到时就马上发现并报错。对于有返回值的函数缺少返回语句，要在整个函数都分析完才能知道是不是有返回语句，并在函数末尾报错。

我们首先来看函数的定义：

> *<有返回值函数定义> ::= <声明头部>'('<参数表>')'{'<复合语句>'}'*
>
> *<无返回值函数定义> ::= VOID <标识符>'('<参数表>')"{'<复合语句>'}'*

那么我们要判断一个函数内部是否有 return 语句，一定是通过<复合语句>然后逐层深入下去。

关于面向评测机的编程（~~不推荐~~），我相信大家一定都各有各的技巧，我说一点我是怎样发现我错误处理的问题的。它在错误处理阶段的测试还是挺弱的，并没有覆盖到所有情况，所以我在之后的作业中有意去测试一些很容易发现的问题，果不其然，在代码生成阶段我发现了函数返回语句出错的问题。

我的现象是这样的：

1. 进入错误处理时，直接 exit(-1)，交上去运行时错误。
2. 进入错误处理时，直接 return，AC。
3. 进入错误处理时，让它正常执行，AC。

上面的现象很明显告诉我们它在运行某个正确程序时，进入到了错误处理里面，即把某个正确的语法误判为错误，接下来我们逐个定位是哪个错误处理的有问题。

我的方法是这样的：采用二分法(/滑稽) 注释掉错误处理部分，逐个定位。

那么我的问题出在哪里了呢？对于有返回语句的函数定义我没有识别到它的返回语句。

我们递归下降分析语法成分时，判断是否有返回语句的流程是这样的：

<复合语句> -> <语句列> -> <语句> -> 各种语句

我在最初分析是否有返回语句的时候是通过在“各种语句”这个阶段有没有 return 语句得到的，但忽略了在各种语句中包括<条件语句>、<循环语句>、<语句列>这三种语句类型，在这三种语句类型中又可能包含 return 语句，因此对这三种类型的语句，我们不能直接说它不是返回语句，而是要递归深入下去判断这种语句里面是否包含 return 语句。

代码生成及优化

啰啰嗦嗦说了许多，终于到最后的大 BOSS 环节了。

中间代码生成

毫不客气地说，我在代码生成一阶段的思路是完全错误的，所以我在代码生成二阶段对自己代码进行了重构。(惨兮兮)

由于代码生成一阶段考察的点较少，因此我的处理办法似乎是直接跳过了生成中间代码阶段，面向的目标代码编程。简单来说是这样的，如果你和我一样那只能祝你好运了，想必我们都能发现代码生成阶段最难处理的一个是表达式，一

个是函数。由于代码生成一阶段没有考察函数，而表达式等于项相加，项等于因子相乘，所以在我遇到因子时就压栈，在分析项的语法成分时如果遇到乘法运算符就先将两个因子压栈后，取出栈顶的两个因子做乘法运算然后再压栈；在分析表达式时，如果有加法运算符，则先将两个项的结果压栈后，取出栈顶两个元素做加法运算后再压栈。看到这里，相信聪明的你已经发现了（来自操作系统的嘲讽），是不是像极了后缀表达式。没错！可是，我们的中间代码中的中间变量体现在哪里了呢？

其实现在看来，体会到上面的思想已经懂得怎么生成中间代码了，只不过我一直卡在自己的思维误区里，想明白后才懂得怎么做。

生成中间变量

生成中间变量是要在语义分析时就设置一个（字符串）栈，用来保存所有的操作数（当然也包括中间变量）。这里面最关键的一点在于中间变量的标号，我的策略是这样的：

1. 初始化中间变量的序号为 0。
2. 两个数值类型做运算，直接将结果压到栈顶。
3. 有中间变量参加运算，将新的运算结果赋给这个中间变量（相当于更新这个中间变量的值）
4. 没有中间变量参加运算，根据序号生成一个新的中间变量。
5. 每次语句执行完毕，将中间变量的值重置为 0。

接下来是丧心病狂的代码优化部分：

划分基本块：

划分基本块的算法关键在于找到入口语句，在我们的 c0 文法中，确定入口语句有以下几种情况：

1. main 函数的第一条语句为入口语句。
2. 条件或无条件跳转到的第一条语句为入口语句。
3. 条件/无条件跳转的下一条语句为入口语句。
4. call 语句视为无条件跳转语句。

5. return 语句视为无条件跳转语句。(return 能跳到的第一条语句和后面紧跟的一条语句都是入口语句，但由于 return 回的位置一定是 call 语句的后一条语句，而这条语句在分析 call 语句时已经加入，因此这里只加入 return 语句的后一条语句即可)
6. exit 语句视为无条件跳转语句。

在所有的入口语句确定后，我们根据入口语句的先后位置进行简单的排序和去重。

之后下一个基本块的入口语句作为上一个基本块的结束语句，最后一个基本块的结束语句为所有语句的最后一条语句。

至此，基本块划分完成。

建立流图

建立流图即在于找到基本块能跳转到的基本块，算法如下：

1. **如果一个基本块的最后一条语句为 exit 语句。**则将其指向最后一个基本块。
2. **如果一个基本块的最后一条语句为 return 语句。**则将其指向若干个基本块，满足该基本块的上一个基本块满足最后一条语句是 call 语句（或者说基本块的上一条语句是 call 语句）且调用的函数是该 return 语句所在的函数。(这里遇到一个 call 语句时不能退出，因为可能有多处调用该函数，那么 return 回的位置就是指向这多个基本块)
3. **如果一个基本块的最后一条语句是 goto 语句或 call 语句。**则将其指向一个基本块，且该基本块的第一条语句是函数定义语句或者标签语句，且该标签为无条件跳转语句跳转到的标签。
4. **如果一个基本块的最后一条语句是有条件跳转语句。**则将其指向两个基本块，一个是它所在的下一个基本块，一个是它跳转到的基本块。
5. **如果上面四个条件都不满足，则将其指向它所在的下一个基本块。**

至此，流图建立完毕。

计算每个基本块的 use 和 def 集合和活跃变量分析

这两步都按照书上步骤来即可，无坑点。

构建冲突图

构建冲突图时一定要考虑好算法，如果采用错误的算法构建冲突图，后面一连串都会跟着错，所以建议在这一步时就一定要验证正确性。这个地方如果只单纯照搬书上构建冲突图的方法是有问题的，这点我也是在读学长文档时发现的。

书上认为，两个活跃范围重合的变量冲突，即两个变量所在的基本块集合（这里所在基本块指在 in 集合里）重叠，则冲突。但在下面的例子中：

```
1.  function(int a, int b){
2.      int x, y;
3.      x = a;
4.      y = b;
5.      …
6.  }
```

根据定义 in=use∪(out-def)，x 会出现在 def 集合里，所以不会出现在这个基本块的 in 集合中，但是可能出现在其他的基本块里，所以依然有可能分配 s 寄存器。而 a 和 b 出现在 use 集合里，所以一定会出现在这个基本块的 in 集合中分配 s 寄存器，如果仅仅要求 in 集合中的变量冲突，a 和 b 是不会和 x、y 冲突的。这样的话，b 可能和 x 分配了同一个 s 寄存器，显然这样有问题。这个例子说明，至少在活跃变量分析中，书上对于冲突的描述是有问题的。

因此，改正构造冲突图的算法后，我们添加将 in 集合和 def 集合的元素也冲突，def 和 def 集合的元素也冲突。

至此，构造冲突图完成，然后分配全局寄存器的工作按照图着色算法即可。在这里，我们参与全局寄存器分配的变量只包括函数内定义的局部变量。不包括全局变量的原因书上已经提到：

> *处于线程安全的考虑，全局变量和静态变量一般不参与全局寄存器的分配，这是因为当一个全局变量或静态变量的值保存于寄存器中时，如果发生线程切换，当前的寄存器状态将作为线程现场被保存，切入线程将恢复其此前保存的寄存器状态，这就导致了其他线程无法得到该寄存器在此前线程中的值，程序运行可能发生不可预知的错误。*

简单来说，就是分配的全局寄存器在我们调用函数时要保存现场，而全局寄存器无需也不能保存现场，它在调用函数中怎样被修改就修改了。当然，我认为在这里加个条件判断来判断它是否需要保存也是可以实现的，不过有点和全局寄存器的思想相悖，故没有实现。实际上，一个好的 C 程序也应该尽可能使用全局变量。

在分配全局寄存器的过程中，为了避免函数内定义的局部变量与全局变量发生冲突，我对函数内定义的每个变量都进行了重命名操作，方便确定一个运算数是局部变量还是全局变量，更好的进行数据流分析。

上面的工作，我们完成了全局寄存器的分配。

关于流图的构建，我补充以下几点我的做法和观点：

1. 计算 use 和 def 集合时，包含中间变量。

2. 计算 in 和 out 集合时，由于第一步中包含中间变量，因此 in 和 out 集合包含中间变量。

3. 构建冲突图和建立流图时不含中间变量。

之所以采用上面的方法是和我的架构相关的，在分配临时寄存器时需要判断中间变量是否跨块，而中间变量是不能用全局寄存器保存的。（后面提到）

临时寄存器池

如果说我比别人看得更远些,那是因为我站在了巨人的肩上。

同样，寄存器池的分配算法参考了学长文档中的 LRU 算法。

LRU 是我们在学习操作系统页面置换中提到的一种算法，它的全称是 LEAST RECENTLY USED，即最近最少使用。

分配策略：

首先保留$t8, $t9 寄存器作为数据的缓冲。怎样理解缓冲呢？就是你在优化之前怎么实现的目标代码生成，应该是只用到了两个寄存器就够了吧。那么，我们在优化后，仍然要保留两个寄存器作为这个作用。

剩余的$t0 - $t7 共 8 个寄存器用于分配给临时变量，分配给临时变量的策略如下：（LRU）

1. 如果临时寄存池中已经有该变量，即已经为该变量分配过临时寄存器，那么将这个变量及对应的寄存器移至队首，并返回该寄存器。否则进入 2。

2. 如果临时寄存池中有剩余寄存器，那么建立该寄存器与变量的对应关系，将该寄存器标识为被该变量利用，并将其移至队首，返回该寄存器。否则进入 3。

3. 取出队尾的临时寄存器，首先将其现在对应的变量保存到它相应的内存位置，然后建立新的对应关系，该寄存器与新申请的变量，将其移到队首，并返回该寄存器。

这里涉及到一个很严重的问题，**中间变量是否可能跨块？**

先说结论，中间变量可能跨块！

大部分中间变量是不跨块的，除了两种情况：

1. 将函数调用的返回值赋给一个中间变量，那么这个中间变量跨块。

e.g.

```
1.  int add(int x, int y) {
2.      return (x + y);
3.  }
4.  void main() {
5.      add(add(1, 2), add(1, 3));
6.  }
```

在上面的例子中，add(add(1, 2), add(1, 3))对应的中间代码如下：

```
1.  push     1
2.  push     2
3.  call     function___add
4.  #TEMP_VAR_0*     =   ret
5.  push     1
6.  push     3
7.  call     function___add
8.  #TEMP_VAR_1*     =   ret
9.  push     #TEMP_VAR_0*
10. push     #TEMP_VAR_1*
11. call     function___add
```

显然，#TEMP_VAR_0*这个中间变量在下一个基本块中被用到。

2. 第二种情况在下面的循环优化中提到。

后面我在测试自己程序的时候发现中间变量跨块的情况不止这两种，我哭了。简单举例：

```
int add(int x, int y) {
    return (x + y);
}
void main() {
    int x;
    int y;
    x = 2;
    y = 1;
    add((x * y), add(1, 2));
}
```

上面的测试程序生成的中间代码为：

```
function___main___x      =   2
function___main___y      =   1
#TEMP_VAR_0      =   function___main___x     *   function___main___y
push     1
push     2
call     function___add
#TEMP_VAR_1*     =   ret
push     #TEMP_VAR_0
push     #TEMP_VAR_1*
call     function___add
```

显然，中间变量#TEMP_VAR_0 跨块了。

那么，我们在分配临时寄存器时策略就要有些变化，这时就用到我们在上面提到的 in 和 out 集合中都包含了中间变量了。

方法一：

在分配临时寄存器时，判断该中间变量是否跨块，若跨块则不分配。

注：判断是否跨块的可以通过看 in 和 out 集合中是否包含该中间变量。

方法二：

分配临时寄存器时，所有中间变量都可以分配，不过在进入新的基本块清空寄存器时把跨块的中间变量保存到内存中。

经笔者实验，上面两种方法在公开的小测试集上性能差不多，之后在大数据集上测试后可以择优选择。

循环优化

当我惊艳于为什么有的大佬 j 型跳转指令那么少时，已经有大佬在讨论区开源了，于是我又站在巨人的肩膀上实现了循环优化。

在给我们的 c0 语法中，循环语句有三种结构：

1. while() {}
2. do{} while()
3. for() {}

那么这三种形式的循环其实对应的跳转指令是不一样的。

对 while(){}型循环：

```
START_LABEL:
BZ END_LABEL
...
j START_LABEL
END_LABEL:
```

一条 j 指令，一条 b 指令；

对 do{} while()型循环：

```
START_LABEL:
...
BNZ START_LABEL
END_LABEL:
```

只需一条 b 指令。

对 for() {}型循环：

```
...
START_LABEL:
bz END_LABEL
... // 循环体内部
... // for(;;)
j START_LABEL
END_LABEL:
```

一条 b 指令，一条 j 指令。

通过分析跳转指令的指令数，相信你已经可以发现，三种循环体结构中，do{}while()型循环所需的跳转指令数最少，因此，我们可以通过将其他两种循环结构转为 do{}while()型循环结构来减少 j 指令数。

但，这样做是不是有问题呢？果不其然，是有问题的。这里就引申出我们上面所说的中间变量跨块问题。

首先来看我们对临时寄存器的分配策略，由于每次进入基本块前我们都清空临时寄存器池然后重新分配，这样做的前提是中间变量不跨块，即一个中间变量的生命周期仅仅存在于自己的基本块中。

这里首先引用讨论区中大佬对此循环的分析：

缺点：

1. *通常使用前端实现优化，不属于严格的中间代码优化*

2. *产生额外的一份循环条件计算的中间码，占用内存空间，且变更了中间码的出现次序。*

这里我们重点理解第二条，以 for 循环变为 do{}while()型循环为例，我们在循环体开始前的条件判断不能少，因为 for 循环不能保证一定能进入循环体内，而 do{}while() 型循环至少会走循环体一次，因此必须保留循环体开始前的“一次性条件判断器”，同时，我们改成 do{}while() 型循环后，在循环体结束后要加一个条件判断，这样就从原来的一个条件判断多了一个条件判断。那么我们知道，条件跳转语句是重要的入口语句标志，所以势必会对基本块的划分产生影响。

说了这么多，体现在 for 转 do…while 的问题在于中间变量跨块了。见下例：

```
1.  void main() {
2.      int i, a;
3.      i = 0;
4.      while (i < a + 3)
5.      {
6.          i = i + 3;
7.      }
8.  }
```

上段代码生成的中间代码如下：

```
function___main___i      =   0
#TEMP_VAR_0      =   function___main___a     +   3
BGE      function___main___i     #TEMP_VAR_0     WHILE_END_LABEL__0
WHILE_START_LABEL__0:
#TEMP_VAR_1      =   function___main___i     +   3
function___main___i      =   #TEMP_VAR_1
BLT      function___main___i     #TEMP_VAR_0     WHILE_START_LABEL__0
WHILE_END_LABEL__0:
```

可见，#TEMP_VAR_0 跨块了。

之所以会产生这样的问题，便在于上面我们理解的第二条，由于我们多生成了一次条件判断，而且这次条件判断只是对第一次条件判断的一个复制版本，因而可能会造成中间变量出现在条件判断中导致此中间变量在两个不同的基本块中的条件判断中都出现。

这时候回过头来说我们的寄存器分配策略，由于我们的临时寄存器的分配前提建立在中间变量不跨块的前提下，因此每次进入基本块都会清空临时寄存器池重新分配寄存器，导致两个相同的中间变量在不同的基本块中分到的寄存器不同。

（解决方法在上面已经提过）

a 寄存器分配

a 寄存器用来存储参数，由于$a0 寄存器在输出时会被改变，因此我们只是用$a1-$a3 三个寄存器保存函数最多三个参数，其他参数通过压栈的方式放入内存中保存。

函数内联

此处我做的是十分简单的函数内联，只减少了 j 型跳转指令，即不用 jal 到一个函数，再 jr 跳回函数调用的位置。

函数内联满足的条件：

(1) 内联函数中只有一个基本块

(2) 内联函数不能调用其他函数

在上面的条件中，其实(2)条件大部分情况都可以包含在(1)条件中，因为如果内联函数内调用了其他函数，那么它一定不止一个基本块（除了函数只有最后一次函数调用且 call 语句在函数最后一句）。

满足上面的条件后，其实我们就可以发现这个函数中没有什么特殊的成分了，即你让他直接接着往下执行和跳过去执行没有任何区别，因此我们就直接把函数跳转到后要执行的那段代码“接”到这个 call 语句的下面就可以了。

看到学长/姐的文档中有提到对函数不能定义局部变量和对函数的参数个数有限制的条件，那样应该是对函数做了更精确的内联。由于我们已经进行过活跃变量分析，所以即使函数内定义了其他变量也不会有任何影响。

死代码删除

在程序中经常会遇到许多代码无法执行，或者执行后对程序运行结果不产生任何影响，比如下面的 swap 例子:

```
1.  void swap(int a, int b) {
2.      int temp;
3.      temp = a;
4.      a = b;
5.      b = temp;
6.  }
```

这个代码执行完后其实不会产生任何影响，包括许多其他情况，如果死代码出现在循环体内，那么优化的效果将相当明显。如在我们的竞速测试样例中就存在这样的情况。

算法:（参考学长文档）

1. 遍历每个基本块，在每个基本块内做 2-4 操作。

2. 初始化 tmpout 集合为基本块的 out 集合。

3. 从后往前遍历基本块内的语句，到达某条指令时如果 out 中不包含当条指令定义的变量且当条不是 scanf 语句(这是一个坑点，虽然输入是无用的，但是不能删掉) 则删除这条语句；否则，更新 out 集合。

4. 重复 2-3 直至不能再删除语句为止。

保存现场

在函数调用时，我们需要保存的现场包括被利用的 s 寄存器和 a 寄存器，我们判断的依据如下：只保存需要保存的寄存器，如果没有之前没有被利用或者之后没有再被利用则无需保存。（可以通过活跃变量分析得到）

ra 寄存器的保存：每进入到一个函数中的时候判断是否需要保存 ra 寄存器，如果当前函数是叶子函数，即它函数内部不会再调用其他函数，则无需保存 ra 寄存器。

一些其他的小优化

常数传播

在语法分析阶段我们已经提到过，如果参与运算的运算数是一个常值，则直接使用常值参与运算，即在目标代码中不会出现常量标识符和可以由我们完成的计算(如$t0 = 1 + 2)。

无用跳转指令的删除

通过上面的常数传播，如果条件恒为假，那么我们可以直接删除这一段代码（甚至都不用跳过这段代码），更加减少跳转指令。

如果跳转指令跳转的位置就紧接在这条指令的下面则无需跳转，可直接删除。

指令的选择

div $t1, $t2, $t3 会被拆成 4 条指令，可改为 div $t2, $t3;mflo $t1；

subi $t1, $t2, 100 会被拆成 2 条指令，可改为 addi $t1, $t2, -100

这种优化方法可以放到 Mars 中去编译你的 MIPS 指令，检查是不是有代价过大的伪代码指令可以通过基础指令替换掉。

目标代码层面的优化

遍历我们要输出的 MIPS 指令，如果紧邻的两条指令出现类似 sw $t1, 0($sp); lw $t1 0($sp)的情况（或者两条指令交换顺序），则可删除后面一条指令。本种优化方法面向目标代码，可以在自己生成 MIPS 指令后人眼观察一下，有些无用的指令经过特判就直接可以删除掉了。

不过这个优化方法有风险，考虑情况一定要全面！

写在最后

北航 6 系的学生如果到大三还不会写脚本有点说不过去了（经历过 OO 的痛的人都懂）。Mars 本质就是个 jar 包，可以在命令行通过 java 指令运行，同时可以将输出重定向到一个文件中，再使用指令对比（win 中是 fc 指令，bash 中是 diff 指令），这是最简单的一个脚本，当然你可以有更加复杂的验证程序，使用脚本一定能提高你的效率。充分的验证是成功的前提，想必经历过大二专业课的同学们都会深有体会。

在最后优化部分仍有许多可以完成的优化，我在这几天没日没夜的优化过程中虽然有苦痛，但更多体会到了优化的乐趣。可做任何事都是一种取舍，在面临接二连三考试的情况下不可能将全部精力都投入到一门课设上面，所以希望学弟学妹也能权衡好自己时间，不要因噎废食，得小失大。

编译课程带给我酸甜苦辣，只有经历才能懂得！感谢老师助教和身边同学们的帮助，也希望课程越来越好。

文档中有不对之处，欢迎批评指正，希望能给后来者一些参考。