**编译申优文章**

15061129 解小锐

这次编译大作业的完成过程中，还是遇到了很多困难的，但是这些困难只要花时间，总会克服或者找到一个折中的办法。

首先，我想说一说这一次编译大作业的整体理解，C0文法的编译系统其实可以分为3个层次2个映射，3个层次分别是：高级语言，中间代码，mips汇编代码，2个映射分别就是高级语言到中间代码的映射，中间代码到mips代码的映射。在这个过程中，难点在于：

①高级语言到中间代码的映射，即语法，语义分析以及符号表的建立，中间代码的生成，而其中我觉得最复杂且最重要的可能是语法分析，因为语法分析是语义分析的**驱动**，我们的语义分析是语法制导的，所以如果语法都不熟悉，那么语义分析会一塌糊涂

②中间代码层次的优化，这个层次的优化主要是指dag图的优化

③中间代码到目标代码的映射，这个映射最最关键的就是寄存器的分配了。

下面我把我在课程设计中遇到的各个困难点做一一详述。

**问题1. 语法分析中，部分非终结符的右部FIRST集冲突**

在给定的C0文法中，有很多非终结符的多个右部的FIRST集合是冲突的，例如：

<因子>::= <标识符>|<标识符>’[’<表达式>’]’|’(’<表达式>’)’|<整数>|<字符>|<有返回值函数调用语句>

可以分别求出每个右部的FIRST单词集合：

FIRST(<标识符>) = {标识符}

FIRST(<标识符>’[’<表达式>’]’) = {<标识符>}

FIRST(’(’<表达式>’)’) = { ( }

FIRST(<整数>) = {整数}

FIRST(<字符>) = { <字符> }

FIRST(<有返回值函数调用语句>) = {标识符}

可以看到其中 <标识符> ， <标识符>’[’<表达式>’]’，<有返回值函数调用语句> 的FIRST集合是冲突的。

按照课本上的知识，我们要想不回溯的使用递归下降分析法，就必须对语法进行改写，即反复提取左部，因此我将<因子>改写为了：

＜因子＞ ::=<标识符>(<空>（这里有两种情况，根据语义分析结果判断）|‘[’＜表达式＞‘]’|‘(’＜值参数表＞‘)’ )|‘(’＜表达式＞‘)’｜<常量>

可以看到，在提取后，简单的变量标识符和无参的函数调用还是会冲突，而且它们的FOLOW集是相同的，所以此时只能借助语义分析来帮助判断。不过此时已经可以使用递归下降子程序来分析了。

对于那些使用BNF表示法所写的文法，在分析时一定要小心，如果觉得理解困难，我觉得可以把方括号 [t ] 转化为<空>|t 或许会更好理解，把大括号 {t}转化为<空> | t | tt | t...。

**问题2 .基本块的划分**

基本块的划分算法在书上有介绍：

1. 语句序列的第一句作为入口语句
2. 跳转语句的下一句作为入口语句
3. 跳转语句要跳转的第一句作为入口语句

但是在实际实现时遇到了如下问题：

①整个语句序列是指什么？

②跳转语句包括哪些？

③基本块的前驱和后继怎么求？

为了解答上述问题，我们需要搞明白一个事情：划分基本块是为了做什么？

**是为了之后全局数据流分析和基本块内的dag图优化。**

数据流分析的“全局”是指整个程序吗？显然不是，对于C0文法来说，一个函数块是分析的一个整体单元，因为C0文法是不允许函数内部的嵌套定义的。所以说基本块的划分也是要以函数为整个块划分。这样就解答了第一个问题：**第一句是指函数的入口语句**。

那么我们如何建立基本块之间的联系呢？就是通过跳转语句，但是需要注意的是，有两种跳转需要特殊对待：

①函数返回语句

②函数调用语句

因为我们的基本块是以函数为整个单元划分的，函数返回语句是结束这个函数，函数调用语句是跳到另一个函数，都与当前函数的其他基本块无关，所以我们在遇到上面两种跳转时，不建立相应基本块的前驱后继关系。

**问题3. dag图算法的修改和扩展**

在一开始使用dag图时，我遇到了如下问题：

①课本上只讲到了简单运算类型的dag图优化，并没有讲数组取值存值，输入输出等其他四元式语句的dag图创建方式，因此，我在一开始的基本块划分必须要在这些地方断开，因为它们不能参与dag图优化，这样做会导致基本块太小，优化不能做的很充分

②从dag图恢复中间代码的启发式算法无法保证正确的代码执行顺序

③书上所讲的dag图算法在某些情况下会出现某个dag图的中间节点没有节点表中相应的变量与之对应，这样在dag图恢复时无法恢复

所以一开始我认为dag图很鸡肋，但是随着与同学的不断讨论，以及自己的不断摸索改正，现在我的dag图不敢说逻辑上完全正确，但是在已有的测试文件下表现都很良好，下面我对上面三个问题作出详细的解决方案的说明。

**（1）dag图导出中间代码的顺序确定**

对于问题①，除了不知道把这些四元式“加入”dag图的创建的方法，最大问题还是**无法保证这些四元式在经过创建dag图然后导出这一过程后，执行顺序有没有被改变，**也就是问题②。

问题②很好解决，书上的dag图重新导出中间代码的算法是一个启发式的算法，这个算法倾向于找**最左子节点**，为什么要这么找呢？书上说是为了保证在子节点被计算好后，可以尽快的利用这个子节点的寄存器进行父节点的计算，即书上的启发式导出中间代码的背景条件是**寄存器资源稀缺**，但是我觉得这个条件对于我们来说不是一个限制条件，因为我们的目标体系结构是mips架构，寄存器资源相对来说是不稀缺的，所以**不必要按照书上的启发式顺序导出中间节点**。

所以我采取的方式就是：**就按照中间节点的创建顺序导出中间代码**，这样可以完全保证代码的正确执行顺序。

1. **对于printf，scanf（即输入输出），push（函数参数压栈），数组元素的取值和赋值的支持**

对于printf和push，这两者其实是一类操作符，都是对变量的**使用**，而不是重新赋值，它们还有一个特点就是：printf和push都是**有一句就有一句的效果**，即多个printf和多个push并不能合并为同一句。

根据这两个特点，一开始我是想让这两个操作数作为dag图中间节点的特殊属性，毕竟并没有对变量进行重新定义，在重新导出dag图时，如果检测到节点有这个特殊属性，那么就做相应的操作。但是这样做的一个弊端就是：**无法保证正确的顺序**，因为dag图重新导出中间代码，我们是按照中间节点的创建顺序导出的，所以**要想保证正确的顺序，就必须把printf和push作为中间节点创建**，因此我把printf和push操作都作为中间节点创建，子节点就是要被输出或压栈的操作数。

但是还有一个问题就是，printf和push因为之前说的那个“有一句就有一句的效果”这个性质，不能在遇到多个相同的printf，就把它们合并到一个节点上，所以我在这里实现的方式是遇到printf或者push语句，就创建新的中间节点，这样不但能保证每一句printf或者push都会被恢复，而且还能保证顺序的正确性

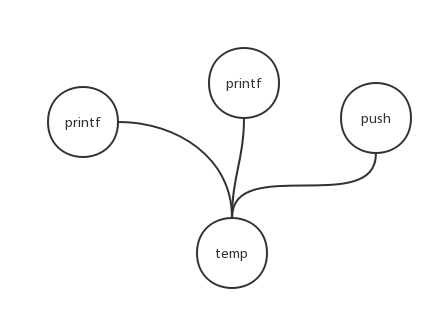
按照上面的说法，在遇到

printf temp;

printf temp;

push temp;

这样的四元式时，可能的dag图是这样的



对于scanf，其实就是变量的重新定义，所以按照一种新的运算方式和加减乘除的处理是一样的。

对于给数组元素赋值这个操作，和普通的运算不同的是，两个给数组元素操作可以合并为一个不仅要满足赋值操作数，下表操作数一样，还要满足被赋值的数组名字相同。所以在寻找公共中间节点时，给数组元素赋值操作需要与普通运算操作区分开来。

只要注意上面几点，就可以把输入输出，数组元素的取值和赋值，函数参数压栈等操作加入到dag图优化中，这样做也扩大了每个基本块的范围，让dag图的优化更加充分。

1. **dag图中间节点恢复为中间代码**

在书上的dag图恢复算法中，中间节点到中间代码的恢复是生成了新的临时变量的，但是在我自己的实践中，重新生成临时变量十分繁琐，因此我使用与这个中间节点index相同的节点表节点的名字来作为中间代码的结果操作数，但是这样做的缺陷是：**在导出时可能没有节点表中的结点与这个中间节点相对应。**

例如：

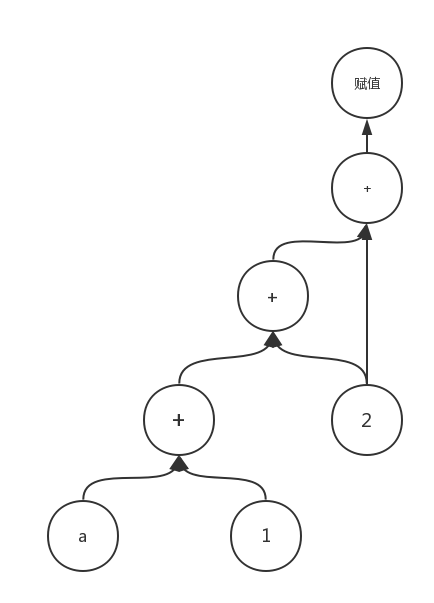
t1 = a + 1;

a = t1 + 2;

b = a + 2;

a = b;

其最后的dag图和节点表如下：



|  |  |
| --- | --- |
| 名称 | index |
| a | 6 |
| 1 | 1 |
| t1 | 2 |
| 2 | 3 |
| b | 5 |

在导出dag图时，就会发现没有与4号dag图中间节点（从0开始）相同index的节点表项。

为了解决这个问题，我为中间节点设置了name属性域，即每个中间节点自己保存着一个合法的名称，中间节点的name的维护方法如下：

①在初始化这个中间节点时，name就设置为相应中间代码的结果操作数，并将空缺标志位设置为false

②当节点表中一个节点的index被改变时，检查它是否是对应的中间节点的name，如果是，那么就设置空缺标志位为true，此时在重新导出中间代码时，这个节点的name还是可以使用的。

③当某一节点表中的结点的index需要被设置为这个中间节点的index时，检查标志位，如果为true，那么把这个中间节点的name换做这个节点的name，并把标志位设置为false

这里name的机制还有待完善，但是逻辑上大体是正确的。

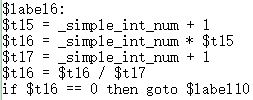
1. **多余变量的删除**

在dag图中一个中间节点通常会因为公共表达式，会有多个节点表节点与之对应，这些节点表节点每一个都对应了一个操作数，这些操作数是保留还是删除，如果没有做全局数据流分析的话，是无法保证操作数是否还会被使用的，只能保守的全部都做补偿性的赋值。做了全局数据流分析，就可以通过查看该变量的活跃情况来判断是否舍弃，具体的做法如下：

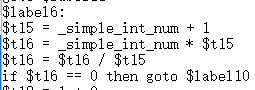
①如果该变量是一个全局变量，对于全局变量需要保证它的有效性，必须要生成补偿赋值语句

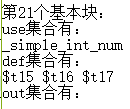
②如果不是一个全局变量，那么就需要看这个查看这个变量是否还在其他基本块活跃，只需要查看这个基本块的out集合中有没有这个变量即可，如果有，那么需要保留这个变量，如果没有，那么直接删去即可。

如下面的优化前的基本块所示：



在优化后，

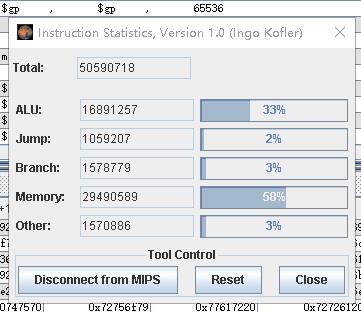




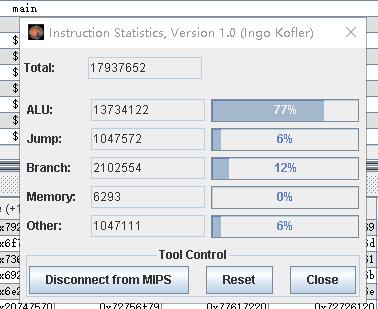
可以看到，因为$t17不在这个基本块的out集合里，所以它被删除了

**（5）优化效果示例**

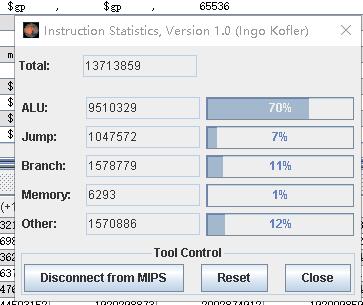
dag图优化和全局寄存器分配的效果是很明显的，测试老师的优化程序，一开始是这样的：



在做了优化，分配了全局寄存器之后，统计如下：



在对dag图算法进行进一步优化后，统计如下：



指令条数进一步降到了1300万

之后我又做了一些赋值的小优化后，统计如下：

