编译课设申优文档

# 语言的选择

## C or C++ ?

对我们这个课程设计来讲，强烈建议使用C++语言，**如果你以前从来没用过C++，正好借这个机会熟悉一下，这是个不可多得的好机会**．

推荐C++的理由是，你不必手动实现如：变长字符串，变长数组，链表，二叉树等等数据结构，直接使用STL，事半功倍．使用STL是C++程序员的必备技能，提前掌握是好的．

## MIPS or x86 ?

如果你不是十分熟悉x86体系结构，千万不要尝试x86！

如果你不是十分熟悉x86体系结构，千万不要尝试x86！

如果你不是十分熟悉x86体系结构，千万不要尝试x86！

我就是入了这个坑，想像C++一样，借这个机会学一学．事实证明，根本就是给自己找麻烦！

理由：

* x86中的指令是两操作数格式，不是如MIPS一样的三操作数格式．最开始会不习惯．
* x86寄存器太少，而且它的通用寄存器有时并不＂通用＂，比如除法指令，这比MIPS蛋疼得多．
* 如果你要做优化，做了全局寄存器分配之后，只有三个临时寄存器可用了，再用临时寄存器池显得太浪费了．但是一般都是先做目标代码生成后做优化，所以我就入了这个坑，先写了临时寄存器池，把六个寄存器都当临时寄存器用．（后来我没有时间做优化了，但是我想到了这个问题）当临时寄存器只剩三个的时候，还不如没有寄存器池．
* **最大的坑就是寄存器太少，这会影响太多方面．**

# 词法分析

第一次阶段作业，是只交一个词法分析程序．强烈建议把给语法分析的接口提前留出来．有几个小问题：

## 读到非法字符怎么办？

非法字符单独作为一类，反上去，语法分析get到错误类型时，要报错．词法分析最好不要自己把非法字符吞了．

## 字符串怎么办？

字符串不能直接调用symbol的读取程序，因为支持的字符不一样．我是单独留了一个函数，专门读取字符串两个双引号之间的部分（不含双引号），这样既方便语法分析报错，又方便读取．

## 字符怎么办？

如果你的文法里，单引号表示的字符常量不支持空格符，那么一定要注意，词法分析读取一个字符时要跳过空白符．与字符串一样，有一个单独的函数读取单引号之间的部分．

## 数字怎么办？

这个就有问题了：如果读取时出现的溢出，比如用户给了一个20位数，要不要处理．我们用的编译器都会给一个警告的，但是我们自己写的编译器可以不处理，这样可以简化问题．如果非要处理，那么一定要想明白，这个错误只能是词法分析来识别，还要想办法把这个错误返给语法分析．

# 符号表设计

**推荐使用树形符号表**．理由是：符号表在生成中间代码之后还要继续存活，在目标代码生成时也会用到符号表．用树形，既能保持它的层次关系，又能让它永久存活．相应的数据结构需要手动实现，不过维护的代码并不多，注意不能是二叉树，每个节点可能有多个子节点．子节点可以用map以提高效率．**要敢于使用指针**．

# 语法分析　＆　错误处理

语法分析本身不会有什么问题，难点是错误处理．**强烈建议写语法分析时就把错误处理写出个大概．**因为当你写完代码生成之后，再回头看那坨像屎一样的语法分析代码时，根本不知道往里加什么东西．写语法分析时，你自己头脑里就已经清楚地知道，这里会有什么错，对什么错应该怎么办．

## 错误处理的思路

按照我们阅读的源代码中的思路，再改良一下，就可以做得比较漂亮了．他用的是一个集合，进每个语法成分之前，都改一个这个集合里的东西，出来后再还原回来．集合表示的意义是，当出错时，要往后跳，直到跳到集合中的符号为止，接着分析．

它用的是集合，也就是STL中的stack，经过我的改良，我们要用list也就是链表，里面存的东西是一个pair，分别为符号和语法成分．跳到一个符号时，我们的递归下降程序也得变到相应的一层．这个语法成分就是这个意思．然后我们在每个子程序的开头，往表中加几个东西，结尾再把加过的东西吐出来．

这里引出一个问题：为什么要用list？用map为什么不行．事实上，笔者之前用的就是map，但是后来发现了一个漏洞，不得已必须使用list．当我们递归进去时，在某一层往里加了一个符号，定义了它对应的语法成分；而再深入进去时，很有可能还会再插一遍这个符号，这不要紧，我们用符号作为键，这是插不进去的．但是当我们在最里层清理插入的符号时，会把外层的那个清理掉．这会导致我们的出错误处理不完整，甚至很乱．用list就是允许符号重复，而且我们可以这样使用：插入时都往最后插，检索时用逆向的迭代器从后往前检索，这样能找到最里层的．清理时，从后往前找．这样不会有错．

## 合理使用宏

使用宏可以让我们的代码更清晰．比如笔者用的这个：

#define TEST(item) { \

if (err != ErrorEnd::NORMAL) { \

if (err == item) \

CLEAR; \

goto END; \

} \

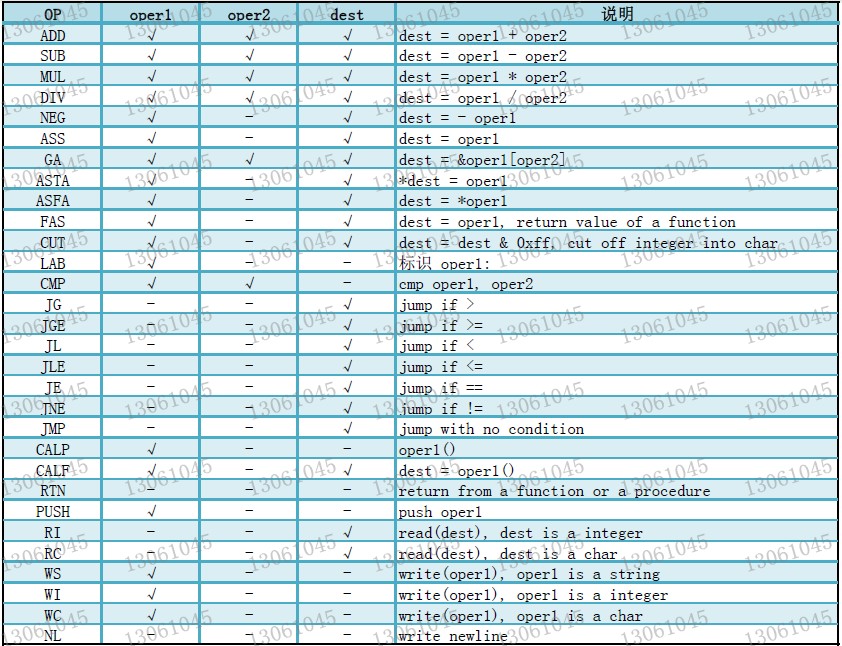
}

大大提高了代码的可读性和可维护性．

# 语义分析和中间代码生成

## 四元式

笔者本来是想做到最高难度的，所以中间代码是四元式．贴上笔者的四元式的最终设计：



直到目标代码生成的最后Debug阶段，四元式的设计还在微调，可见这部分不可能提前把所有的问题都想明白．下面做几点说明（说明可能不全面，有些东西笔者已经忘了为什么了，但是形成现在这个样子，一定是有它的道理的）：

* 由于笔者采用的是x86汇编，所以四元式与x86汇编长得很像．这无所谓，对照着这个四元式直接生成MIPS汇编写不难．
* 赋值语句，有四种：普通的赋值语句，给函数返回值赋值的语句，给指针赋值，从指针取值给变量赋值．PASCAL是一门非常蛋疼的语言，**给函数赋返回值之后，函数不立刻退出而是继续执行**．所以有了FAS，至于另两个，牵扯到另一个问题．
* 另一个问题就是，var类型的参数怎么处理．笔者反复尝试，最优的结果就是，像C语言一样，传地址，然后像C语言的指针一样寻址．有了这个之后，对数组元素的处理也迎刃而解了：也像C语言一样，把数组名当指针，先求偏移，再寻址．所以如果var类型的参数，给的实参是数组元素，这样做也没有任何影响．
* 函数和过程的调用是两种不同的指令，因为过程没有返回值，函数有．
* 读语句要准备两种：读字符，读整数；写语句要准备三种：写整数，写字符，写字符串．最后那个换行是笔者个人的理解，认为一条写语句就是一行．实际做时可以忽略这个．

## 临时变量

语义分析程序产生新的临时变量．然后这里引出一个大问题：程序的结构．语法分析与语义分析是单纯的调用关系，跟笔者最初想的不一样．它们不会互相调，而只是语法分析调语义分析．然后就是，每个表达式的值，要通过一种方式返回给调用者．笔者推荐使用C++的引用，这样程序可读性更好一点，对不需要改的部分，也用引用，这样可以提高效率，并加上const关键字，这样一目了然．

## 表达式处理

表达式，项，因子，这三者的递归关系．它们要从上一层知道，上一层需要的是值还是地址；还要向上一层返回，这一层读到的东西的类型，这一层读到的值／地址，这些东西都通过参数来传递．

# 目标代码生成

## 堆栈管理器

笔者感觉这个东西非常好用．凡是涉及到与运行栈有关的操作时，都通过堆栈管理器生成相应的代码．这样的好处是，程序的结构清楚，而且由于代码非常集中，维护起来非常方便．

## 临时寄存器池

对MIPS体系结构来说，一个好的临时寄存器分配方案非常重要．因为它的临时寄存器非常多，分配合理的话效率会很高．笔者的实现有点偏傻瓜，但是想实现的好一点，其实难度并不是很大，只不过笔者最开始没有想到而已．

比较好的实现方式就是，用＂最近最久未使用＂原则分配．维护两个队列即可，一个是空寄存器队列，如果这个队列非空，直接从里面取一个即可；一个是有值的寄存器队列，如果没有空寄存器，必须从这个队列中换掉一个，按照原则，我们这样做：每次访问某个寄存器时，就把这个寄存器移到最前面，当替换时，我们替换队尾的那个．这个数据结构推荐使用链表，而且并不复杂，建议手动实现，这样能提高效率．

史老师上课时留了一个大坑，他说：代码生成器要使用一个变量时，就跟临时寄存器池要，如果这个变量已经有了，就给它分配一个寄存器；如果没有，要先在栈上给它分配空间，再分配寄存器．**这句话是错的．．．错的．．．的．．．**

正确的做法是：**在每个函数／过程的开头，分配局部变量空间，分配本函数／过程要使用的所有临时变量空间．只有这样，才能保证程序是正确的**．因为有分支和循环，所以如果现用现分配，有时偏移就是错的．笔者就是入了这个坑，导致调试了好久好久才发现这个问题．

关于临时寄存器还有一个小问题：每个基本块都要让临时变量＂回家＂，由于PASCAL是一门非常蛋疼的语言，每个函数或过程都可以访问它的外层的变量.所以回家时有可能还要分配一个临时寄存器用来寻址.对于x86来说，这个过程有可能会无限循环下去，要想明白前因后果加倍小心；对于MIPS来说，这个问题完全可以通过分配一个别的无关紧要的寄存器来实现，反正MIPS的寄存器超多．

## 参数传递

这个要加一点小心：如果不注意，当一个函数调用语句本身作为另一个调用语句的参数时，必须保证不能把相关的参数清理掉．好多同学都在这里栽到了坑里．

# 关于优化

首先声明，我除了临时寄存器池之外没有做其他别的优化，原因是前面一直在Debug导致我只有一天时间写优化了，这还不如不写．但是，关于老师课上讲的几个优化算法，我都进行了深入的思考，这可以给＂后人＂留作参考的．

## 消除局部公共子表达式

建立DAG图，然后从DAG图恢复四元式．

在切流图时就要想好，函数／过程调用语句算不算同一个基本块？如果与上文算作同一个基本块，就引出一个问题：由于PASCAL是一门非常蛋疼的语言，函数或过程可以访问其外层的变量，有可能在调用语句之后，变量的值莫名其妙地变了，这样合并的公共子表达式就不正确了．所以正确的做法是：**函数调用语句与前面的语句不划为同一基本块．**

然后还有一个问题：后面的基本块有可能用到前一个基本块的临时变量．稍一想想就可以明白，用户源代码中的局部变量是必须恢复出来的，而临时变量一般来讲不需要都恢复出来，但是有了上面这个问题，那么就是有一部分临时变量是必须恢复出来的．我们怎么知道哪些临时变量需要恢复出来？这要遍历一遍流图．非常蛋疼．

## 定义－使用链

书上的定义－使用链有一个致命的Bug导致有它跟没它没有节省多少计算量：试想有这样的流图，B1 中有对变量 I 的定义，B2 中有对变量 J 的定义，B3中有对变量 I 的引用，那么我们知道，I 和 J 一定是冲突的，但是在书上的定义－使用链中，体现不出来，要想发现这种冲突关系，史老师说要遍历流图．这就引来了令我非常恼火的问题：**要是有了定义－使用链还需要遍历流图，我之前的活跃变量分析和到达定义分析难道白做了吗**？

答案当然是没有！只是史老师听不进去我的观点，但是可以想明白，我的思路是没有问题的：

我们扩充一下定义－使用链的定义：**同时使用到达定义分析和活跃变量分析的结果来构建定义－使用链**．

定义点没有问题；至于使用点，如果一个块没有对某个变量的直接使用点，就看到达定义分析的结果，如果那个定义点的定义能到达这个块，那么看活跃变量分析的结果，如果在这个块的入口处这个变量仍是活跃的，那么说明这个变量的那个定义点在这个块不能被占用，操作方法是：**人为在定义－使用链上给这个块加上一个使用点，可以把使用点看作这个块的最后一条语句**．

当根据定义－使用链建网时，只需要看同一个块的关系就好了：如果一个定义点在另一个使用点之后，那么在这一个块没有冲突，如果在同一块中，定义点出现在使用点之前，那么就冲突．

使用上面的算法，可以避免重复遍历流图的问题．

## 全局寄存器分配

不管用什么方法，做了全局寄存器分配之后，都会面临一个问题：由于PASCAL是一门非常蛋疼的语言，每个函数或过程都可以直接访问它外层的变量，那么我们分配好全局寄存器分配的变量，就有可能在里层的函数被访问．但是，比如在第０层定义的变量，有可能在第１层被压到了运行栈上，而且不是第０层的运行栈，而是第一层的运行栈，所以如果第２层需要访问它，就要到第１层的运行栈上去找．

这非常蛋疼，但是也并非不可解决．解决的方法是：我们在符号表上记录变量的位置，而且位置有两种：一种是寄存器中，一种是运行栈上，对于运行栈，要给出层数和偏移，以便通过display区找到那个变量．但是这里还有一个问题，如果函数是递归调用的，我们就不知道那个变量在具体哪层放着了．

笔者想到的一种方法是，改变运行栈的结构，在运行栈上加一些东西以标记相应的信息，比如可以标记，参与全局寄存器分配的局部变量所在的栈的基址．但是笔者没有想到更优的方法．如果局部变量还要多次访问内存的话，那么全局寄存器分配还不如不做．这也就是为什么笔者反复说：PASCAL是一门非常蛋疼的语言．