《编译技术》课程设计

文 档

学号：\_\_\_\_\_\_15061200\_\_\_\_\_\_

姓名：\_\_\_\_\_\_ 李明轩\_\_\_\_\_\_\_

2019年 1 月 4 日

**一．需求说明**

**1．文法说明**

我抽取的文法是while-if-switch。这个文法一些需要注意的地方在于，

1. switch可以没有default；空语句有多种形式，诸如空大括号，单独分号等；
2. 后续约定无返回值函数的return语句可以为 return; 或可以完全没有return语句（因此四元式生成中我加入了检测并添加return语句的机制）；
3. 所有函数调用的时候都需要有括号，包括无参函数；
4. 类型需要严格匹配，整形不能用于条件式中等
5. 标识符区分大小写
6. 输入输出语句按照原样输出，也即无须考虑转义字符（处理中扫描字符串常量并将转义字符替换为双反斜杠存储即可）
7. 由于没有break因此switch语句每个子情况执行完毕后需自动跳转结束

**2．目标代码说明**

生成的目标代码是MIPS指令

**3. 优化方案\***

1. 窥孔优化（常量替换，常量表达式计算）

主要包括在四元式生成过程中即对常量进行替换，并对字符型做类型转换。同时在生成Mips code过程中如果识别到四元式的两个操作数均为常量的话，则直接在代码中计算并将结果作为立即数加载到原先分配的结果寄存器中。

2. 个人设计中的一些创新（主要针对减少访存和回写而做）

个人设计中为了避免栈和寄存器上保存同样的重复变量，我在编译器中设计了两种不同的读写模式，其一是变量在寄存器中，则可以通过move指令完成取值或写入；另一种则是变量在栈上，那么此时再通过符号表信息计算栈偏移写入对应位置或从对应位置读取即可。

参数寄存器方面，本人使用了a1-a3一共三个参数寄存器，这样的话在运行全过程中使用前三个参数都会直接和寄存器发生交互（没有a0主要是考虑到a0会作为syscall的传参寄存器，容易有额外存取产生）。而这点在要求参数必须从左到右计算并传入后对我原先的设计提出了挑战。因为在函数嵌套调用中（比如func1作为func2的一个参数传入），多个函数都需要用到a1-a3那么确定存储和恢复位置就很重要。因此我对每次函数调用都记录了一个唯一的编号，然后设计了一个栈式的存储记录，只有当一个函数所有的参数都入栈后才将栈顶记录弹出，而冲突的参数寄存器则根据栈中记录计算对应偏移，并将参数存入正确的位置。

为了减少函数切换现场的寄存器存取，我在生成四元式后会对所有四元式中使用到的临时变量（以#开头）做一次扫描，记录各个临时变量使用到的四元式位置，当某个函数调用子函数时，编译器会查询记录只存储那些之后还会被用到的临时变量，这样大大减少了需要保存的临时寄存器数量。而全局寄存器是由子函数负责在使用中保存，故而也不存在冗余的保存。

另外，全局寄存器写回机制设计中，只有在所有全局寄存器都有脏位时才会触发，算法将选择一个全局寄存器写回栈中并将这个sreg分配出去。

细节请参见后文优化部分描述和运行栈设计。

**二．详细设计**

**1．程序结构**

程序结构可以按照功能划分为几大块：词法分析，语法分析，以语法分析为主体的语义分析和MIPS代码生成，而几种优化方法则各自单独一个模块。定义的全局变量包含：全局错误记录，词法分析的当前词语相关信息，语法分析相关中间变量，符号表，保留字表，目标代码序列，四元式序列等。

总的来说，按照整个编译程序的运行顺序来看，本编译程序采用一遍扫描，语法分析为中心的模式。读入整个代码文件后，将其逐行存入String数组（此时便可以关闭文件了，且记录下了代码行列信息，方便错误处理），然后开始词法分析，词法分析过程中对遇到的词法方面的错误可以进行跳读/自动纠错/集中记录并报错。词法分析结束后将存储下来的分割好的各个词法成分输入语法分析模块，语法分析由递归下降子程序实现，在识别完某个语法成分后，继续调用词法分析程序。包含的功能有识别各种类型的语句，并和文法中的生成式进行匹配检查，如不合法则进行记录/尝试自动改正/报错；填充符号表；对语句中的各个项进行简单的计算，将值一起存入符号表，比如带有正负号的整数，在表达式中的字符类型变量等。然后进行语义分析，将不符合规则的语句检查出来，比如变量使用先于声明，使用变量不符合作用域规则等。这一阶段之后便可以生成中间代码，本项目中要求的是四元式，具体规范见下文。

四元式生成后进行窥孔优化和启发式寄存器分配。最后进行MIPS代码生成采取的具体策略将在优化部分详细描述。

**2．类/方法/函数功能**

本程序并未设计为面向对象形式，主要的几大方法均为面向过程。

1. 词法分析 LexicalAnalyze
   1. Reserver() 查询标识符哈希Map，检查是否为保留字并返回对应的标识符类型
   2. lexicalanalyze(vector<string>::const\_iterator&) 词法分析主函数，输入为读取的文件行字符串列表
2. 语法分析 SyntacticAnalyze
   1. cons\_declaration(LinePtr&, Funcinfo&) 常量声明处理
   2. var\_declaration(LinePtr&, Funcinfo&) 变量声明处理
   3. function\_declaration(LinePtr&) 函数声明处理
   4. paralist(LinePtr&, Funcinfo&) 函数声明中的参数列表处理
   5. value\_paralist(LinePtr&, Funcinfo&, int) 函数调用中的值参数列表处理
   6. func\_call(LinePtr&, bool, string&, string&) 函数调用语句处理
   7. func\_main(LinePtr&) 主函数处理
   8. func\_compound(LinePtr&, Funcinfo&) 复合语句块处理
   9. statement(LinePtr&) <语句>处理（包含识别多种类别语句的综合功能模块）
   10. sta\_switch(LinePtr&) switch语句处理
   11. sta\_if(LinePtr&) 条件语句处理
   12. sta\_while(LinePtr&) while循环语句处理
   13. sta\_return(LinePtr&) 返回语句处理
   14. sta\_printf(LinePtr&) 输出语句处理
   15. sta\_scanf(LinePtr&) 输入语句处理
   16. sta\_assign(LinePtr&) 赋值语句处理
   17. expression(LinePtr&, string&, string&) 表达式处理
   18. condition(LinePtr&, string&) 条件式处理
   19. case\_list(LinePtr&, const string&, const string&, const string&) 情况子语句处理
   20. def\_list(LinePtr&) default语句处理（switch）
   21. term(LinePtr&, int, string&, string&) <项>处理
   22. factor(LinePtr&, string&, string&) <因子>处理
   23. enter(string&, Tabinfo, int kind, int type, Funcinfo, Arrayinfo, int) 写入符号表
   24. isrequiredch(Enum2str, string) 检查下一个符号是否是要求的给定字符
   25. lookuptab(Tabinfo&, string, string, int&, set<int>, set<int>) 给定类型集合和名字查符号表
   26. getsym(LinePtr&) 获取下一个词法分析得到的单词
   27. save(), load() 为方便语法分析回溯，保存/读取当前词法/语法分析指针现场
3. 语义分析和四元式生成 SemanticAnalyze
   1. genQuat(string op, string, string, string, Tabinfo, Tabinfo, Tabinfo) 生成四元式
   2. printQuat() 格式化打印输出四元式
   3. gentmpvar() 按顺序生成临时变量编号
   4. genlabel() 按顺序生成跳转标签
4. MIPS代码生成GenMips
   1. genMips() 生成mips代码（以下如无说明均为生成对应语句mips的函数）
   2. genMips\_const(vector<Quat>::const\_iterator& qiter, bool isglob)
   3. genMips\_var(vector<Quat>::const\_iterator& qiter, bool isglob)
   4. genMips\_array(vector<Quat>::const\_iterator& qiter, bool isglob)
   5. genMips\_funcdec(vector<Quat>::const\_iterator& qiter)
   6. genMips\_call(vector<Quat>::const\_iterator& qiter)
   7. genMips\_push(vector<Quat>::const\_iterator& qiter)
   8. genMips\_exp(vector<Quat>::const\_iterator& qiter)
   9. genMips\_arraycall(vector<Quat>::const\_iterator& qiter)
   10. genMips\_assign(vector<Quat>::const\_iterator& qiter)
   11. genMips\_bool(vector<Quat>::const\_iterator& qiter)
   12. genMips\_switch(vector<Quat>::const\_iterator& qiter)
   13. genMips\_label(vector<Quat>::const\_iterator& qiter, string)
   14. genMips\_jump(vector<Quat>::const\_iterator& qiter)
   15. genMips\_ret(vector<Quat>::const\_iterator& qiter)
   16. genMips\_printf(vector<Quat>::const\_iterator& qiter)
   17. genMips\_scanf(vector<Quat>::const\_iterator& qiter)
   18. printMips() 打印输出mips代码
   19. scanMultiuse() 记录各个临时变量被哪些四元式引用，方便临时寄存器分配
   20. allocateTempReg(vector<Quat>::const\_iterator) 临时寄存器分配算法
   21. genMips\_recordreg(string, string, string, string, bool) 记录使用的寄存器，置脏位
   22. genMips\_findvar(string, string&, bool, vector<Quat>::const\_iterator, string); 链接四元式中某变量到一个制定的寄存器
   23. genMips\_findreg(Reginfo& findresult, string levelname, string varname, string subscript) 查找给定变量是否已经存储在寄存器中
   24. genMips\_lastareguser(int aregnum, string caller) 查询最后一个使用某参数寄存器的函数实例（实现函数嵌套调用中参数寄存器的正确写入写回）
5. 错误信息输出 printError(vector)
   1. printError(vector<Errormsg>, vector<Errormsg2>); 格式化打印所有错误信息
6. 工具类函数
   1. strreplace(const string&, const char&, const string&) 使用给定字符串替换输入字符串中的某个字符
   2. split(const string&, const char&) 使用给定字符分割输入字符串，返回分割结果
   3. printSyntacResult(string syntac, int count, bool iserror, string syntac\_prefix) 输出语法分析中间结果

**3．调用依赖关系**

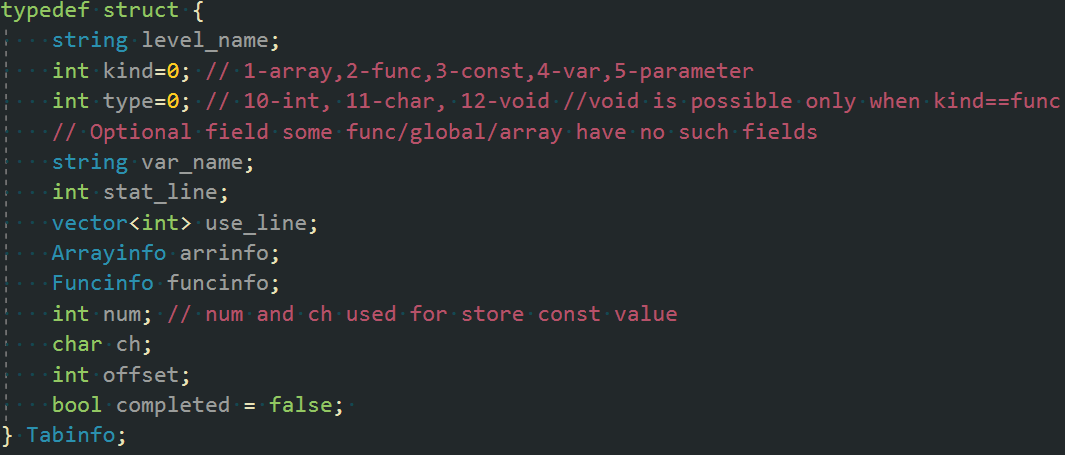
以SyntacticAnalyze.cpp为中心，调用其他头文件中的各个函数。关系如下：

1. main()🡪cons\_declaration(), var\_declaration(), function\_declaration(), func\_main(), genMips(), printSymtab(), printQuat(), printMips(), printError()
2. cons\_declaration()🡪getsym(), genQuat(), enter()
3. var\_declaration()🡪getsym(), genQuat(), enter(), save(), load()
4. function\_declaration()🡪getsym(), genQuat(), enter(), save(), load(), func\_compound(), paralist()
5. func\_compound()🡪cons\_declaration(), var\_declaration(), statement()
6. statement()🡪sta\_xxx(所有sta开头的，用于识别各种语法成分的函数)
7. func\_main()🡪getsym(), genQuat(), enter(), save(), load(), func\_compound()
8. genMips()🡪genMips\_xxx（所有以此开头的函数，用于将对应类型的四元式转换为mips代码）
9. 一些工具类函数，如split(), isNumCh(), strreplace()等，具体功能见上节函数功能说明。

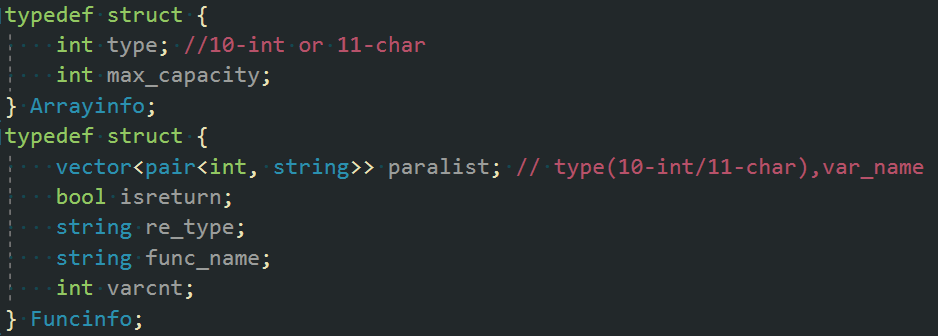
**4．符号表管理方案**

本人设计中符号表数据结构如下

其中Tabinfo为自定义struct



其中Funcinfo和Arrayinfo均为自定义struct

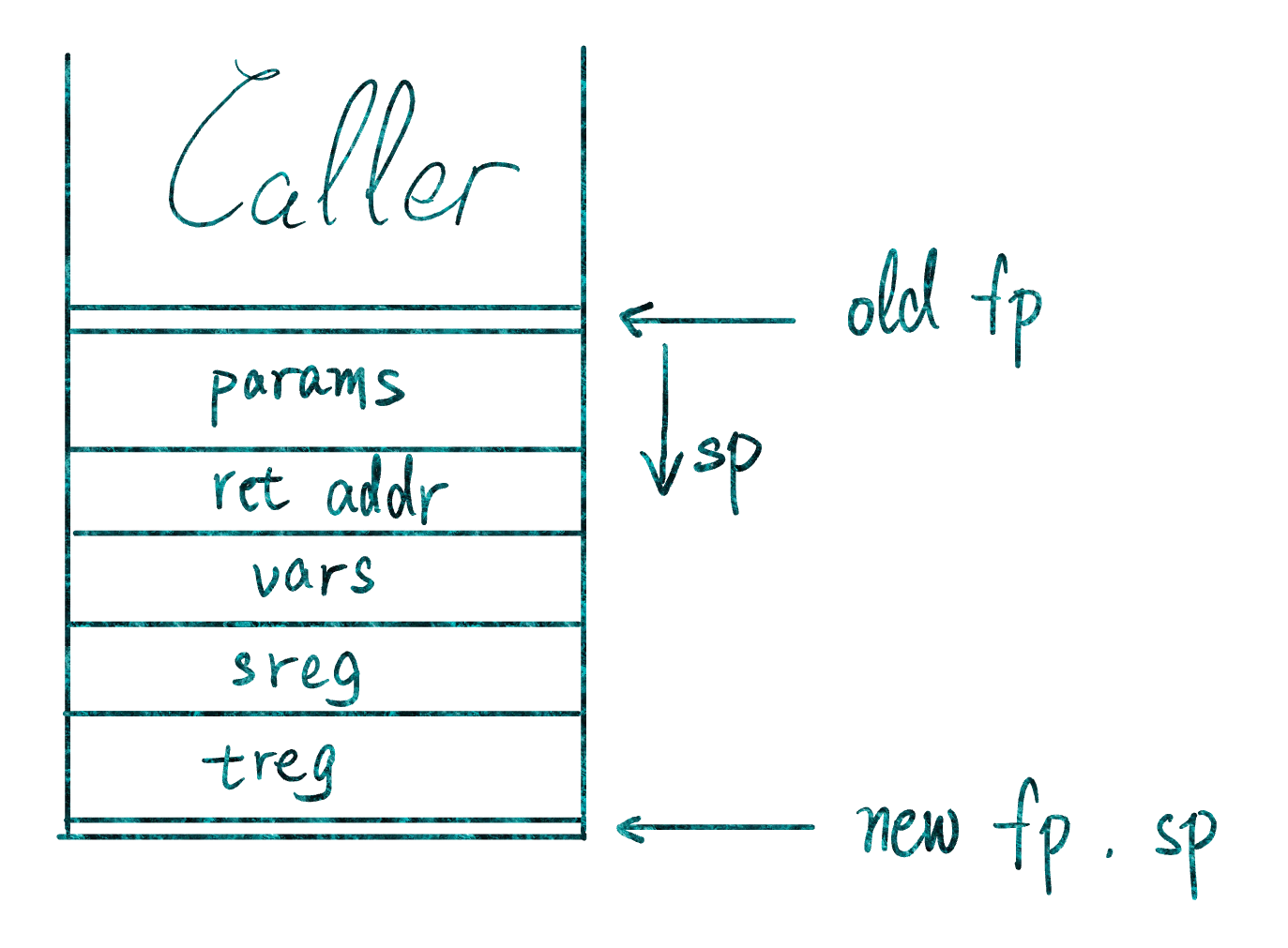


总体来说，符号表设计中本人为了保证结构清晰和信息的完善，并考虑到可能的重名问题及查表便捷性的需求，将符号表设计为以上结构。最外层为hashmap因每个函数层次之间实际没有太多嵌套关系，本文法中只有两个层次(全局和各个函数包括main)。键即为该层次名字，值为该层次的符号表。而每个层次的符号表则是一个有序列表。这个列表的每个元素都由该标识符名称+标识符信息组成。每个层次列表的第一个元素都是该层次本身（也即该函数本身，全局则为虚设的global func代替）之后按照声明顺序依次插入列表。

深入到某个具体标识符的Tabinfo中来看，每个标识符都有一些必填项和根据类型功能决定的选填项。必填包括变量所在层次名（同前文函数名），其种类和类型。标识符名称，声明行，使用行等。数组信息和函数信息为可选的，而num及ch则是为了记录常量的数值，方便后续生成代码时直接查表替换常量。Offset是一个相当重要的变量，记录了该变量相对于函数内部所有非常量变量声明的偏移（包含数组元素个数，用于之后计算运行栈偏移）

进一步深入，数组信息Arrayinfo包含了数组类型和最大容量；函数信息Funcinfo包含了参数列表，是否返回，返回类型，函数名及包含的变量数量（接下来计算运行栈偏移会用到）

**5．存储分配方案**



首先本人设计中运行栈地址空间是从高地址开始往下递减的，弹出记录时则是地址加回偏移量的。

如图所示，对于一个新函数的活动记录，我的fp和sp会从caller高地址开始逐渐减少，当sp到达最下面留出了新函数所需要的所有空间后，将fp挪至sp同样的位置，并固定下来，作为当前函数所有变量寻址的基地址。

而一开始函数压栈时，old fp作为caller活动记录基地址，方便计算参数值，然后挪动sp**按顺序**压栈参数（前三个只留出空间，数值在areg中）。如果压栈过程中需要调用别的函数，则fp保持不变，仍旧使用sp直接下移，计算参数并按照常规调用函数即可，返回时sp和fp的偏移则需要根据当前参数未完全压栈的函数记录来确定，属于特殊情况。

一般在一个函数的所有参数计算完毕并压栈后，将函数返回地址存入ret中，然后按照符号表信息和四元式中的声明，将普通变量和数组**按照顺序压入栈中**，最下面则是为了切换现场保存全局寄存器和临时寄存器留下的空间，每次函数调用时，caller保存treg，待子函数返回后复原treg。而caller自身在分配了sreg的时候需要保存原先sreg中的数值，并在最后返回时复原现场。

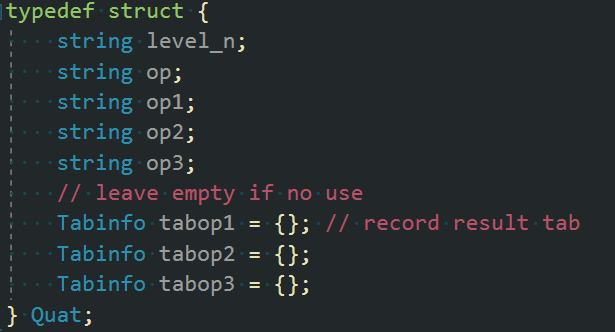
**6. 解释执行程序\***

使用Mars进行模拟运行

**7. 四元式设计\***

【对采用的四元式进行详细说明】

本方案中存储四元式采用的数据结构如图



基本遵循op1 = op2 op op3的格式，除少数特殊四元式之外，具体可见下文描述。

几个操作符因为是字符串形式，在本方案中均有比较灵活的使用，尤其是一些原本不需要用到三个操作数位置的操作，本方案使用空闲位置和op域传递相关四元式信息，方便生成mips代码。同理，对应三个位置的Tabinfo也是为了省去生成mips时重复的查表操作。

输出四元式时，打印格式已经按照课设要求进行格式化输出。

使用的四元式类型主要为以下几种

**Const Declaration (const int/char)**

genQuat("const "+map\_type[tmptab.type], tmptab.var\_name, op2);// op1 = op2

**Var/Array Declaration (int/char array/var)**

genQuat(map\_type[tmptab.type]+" "+map\_kind[tmptab.kind], tmptab.var\_name, to\_string(arrinfo.max\_capacity));

**Function Declaration (Put label)**

genQuat(funcinfo.re\_type+" func "+funcinfo.func\_name);

**Function Start (End of para list, save $ret, $sp if needed)**

genQuat("func start", tmptab.var\_name, "", ""); //start

**Function End (Maybe useful when checking branch return)**

genQuat("func end", tmptab.var\_name, "", "", syntac\_symtab[syntac\_level\_name][0].second); **Parameter List (Cal para count and leave spaces on stack)**

genQuat(map\_type[tmptab.type]+" "+map\_kind[tmptab.kind],tmptab.var\_name);

**Function Call**

genQuat("call "+findresult.funcinfo.re\_type+" func "+funccount, findresult.var\_name, result, syntac\_level\_name);

**Push Parameters into Stack (Function Call)**

genQuat("push "+resulttype, funcinfo.paralist[count - 1].second, tmpresult, syntac\_level\_name+” “+num+” ”+callee\_level\_name);

**Expression (sign = { +, -, \*, / })**

genQuat(sign, op1, op2, op3); // op1 = op2 sign op3

**Array References**

genQuat("=[]", result, findresult.var\_name, subscript, {}, findresult, {});

**Array Assign**

genQuat("[]=", op1, op2, op3, findresult); //op1[op2]=op3

**Assign Statement**

genQuat("=", op1, op2, "", findresult); //tab1 = findresult

**Bool Expression (op = {==, !=, <=, >=, <, >} jump to label if op not satisfied)**

genQuat(op, label, result\_bool, result\_bool2); // op1 op op2

**Switch Head**

genQuat("switch", result, resulttype);

**Set Label**

genQuat(lastlabel + ":");

**GOTO**

genQuat("GOTO", lastlabel);

**Return Statement**

genQuat("Return", "", syntac\_level\_name);

genQuat("Return", result, syntac\_level\_name);

**Printf & Scanf**

genQuat("Printf", resulttype, str, result); //if result is "" then just str

genQuat("Printf", resulttype, "", result);

genQuat("Scanf", lex\_output.key, "", "", findresult);

**8. 目标代码生成方案\***

【说明代码生成有关的数据结构、关键算法】

指令主要包括一些常用算术指令，但是为了避免溢出，add和sub替换为addu, subu。

其他使用的指令主要包括add, addi, mul, div, j, jr, jal, bne, bge, ble, bgt, blt, beq, li, la, sw, lw, move

本方案中使用的寄存器主要包括

|  |  |
| --- | --- |
| 全局寄存器 $s0-$s7： | 主要存储由启发式算法分配的活跃变量（若无优化则存储计算中取出的全局量和局部量） |
| 临时寄存器 $t0-$t9： | 存储中间计算结果 |
| 参数寄存器 $a0-$a3： | 输入的前3个参数常驻a1-a3，为了减少复杂度，由于a0常被用作传递系统调用参数的寄存器，故而不写入函数参数 |
| $sp/$gp/$fp： | $sp存储当前运行栈栈底（有新的函数调用语句需要压栈参数时会下移）  $gp存储需要读取的全局量地址  $fp固定在当前函数的运行栈栈底（细节参见前文运行栈设计） |
| 函数返回值寄存器$v0-$v1： | 考虑到v0常用作系统调用返回值寄存器，故函数返回值均存放在v1中 |

**9. 优化方案\***

1. 窥孔优化：常数替换和常量表达式计算
2. 个人设计的一些创新：前文已经描述
3. 启发式优化

考虑到优化程序可以专门面向特定测试集合优化，故而在时间有限的情况下，我优先选择了启发式优化。具体来说，就是指定将某些函数和其中的某些变量分配给空闲的寄存器，并保证在整个函数运算过程中，这些变量都存储在这些寄存器中。由此即可大大减少访存，并在程序单个函数规模不大时尽可能地优化程序。

设计的关键在于确定有多少空闲的全局寄存器和局部寄存器可以使用，目前出于保守考虑，我选用了sreg3-7, treg5-9一共十个寄存器作为针对性存放变量的寄存器。在优化时需要人工确认优先级，将影响较大的前十个变量和对应函数名称写入代码中，之前的设计安排是自动扫描代码将函数中名字较长以及i, j, k, x, y等常用名优先分配寄存器，后改为人工确定变量。

时间关系，此优化设计较为简陋，望老师见谅。

**10. 出错处理**

本方案中将错误大致分为词法阶段可检测及语法/语义阶段可检测，并且一旦错误列表不为空则在完成四元式生成后格式化输出所有错误并停止运行（程序运行停止在优化和mips生成之前）

词法阶段的错误大部分可以自动纠正或者跳读的均作了对应处理，错误记录中会输出对应信息，包括：输入，期望输入，行号，列号，对应代码，自动更正结果，更正后识别结果，错误信息等。

语法和语义阶段发现的错误会尽量跳读，并通过代码中详细的条件分支给出尽可能详尽的报错信息。输出时会包括：行号，对应代码，错误信息等。

详细错误信息如下：

|  |  |
| --- | --- |
| 错误编号/名称 | 报错信息 |
| 1 不等于号错误 | "! must be followed by = " |
| 2 非法字符 | "Not a valid char"/”Invalid Char” |
| 3 单引号缺失/错误的引号使用 | "Wrong Quote" |
| 4 空字符串 | "Empty STR" |
| 5 字符串引号不匹配 | "Wrong Quote Str" |
| 6 空字符串和单个引号 | "Empty String with no right quote" |
| 7 右引号缺失 | "Right quote not found" |
| 8 重复的函数声明 | "Duplicated Function Declaration. New Func declaration will be discared" |
| 9 重复的变量/常量/数组/参数声明 | "Duplicated var/para/array/const Declaration in the same level" |
| 10 非预期输入字符 | "Illegal Ch " + input.key + " Expected " + legal\_sy |
| 11 常量声明缺少数值 | "Expected a number following +/- in the const declaration" |
| 12 非法使用保留字作为常量名 | "Cannot use reserved name " + lex\_output.key + " as const name" |
| 13 未知类型的常量声明 | "Unknown type const declaration: " |
| 14 数组下标越界或数组下标类型错误（两种信息） | Array declaration can only use unsigned int as subscript  Array subscript should be bigger than 0 |
| 15 变量声明使用了保留字 | "Cannot use reserved name " + lex\_output.key + " as var/array name" |
| 16 函数声明使用了保留字 | "Cannot use reserved name " + lex\_output.key + " as function name" |
| 17 函数缺少返回值 | funcinfo.func\_name + " need a return value" |
| 18 非法参数名 | "Illegal parameter name: "+lex\_output.key |
| 19 非法参数列表格式 | "Illegal paralist form, expected ',' or ')'" |
| 20 非法语句（statement） | "Wrong statement in func compound" |
| 21 文件不完整（如程序最后缺少大括号） |  |
| 22 在表达式中调用无返回值函数 | "Called a void function while a return value is needed" |
| 23 参数过多 | "Too many parameters ! Expected paralist size:" |
| 24 参数类型错误 | "Wrong parameter type, expected: " + map\_type[funcinfo.paralist[count - 1].first] + "Given input: " + resulttype |
| 25 函数调用语句缺少括号 | "Expected ')' at the end of function call" |
| 26 参数过少 | "Too few parameters ! Expected paralist size:" |
| 27 数组下标类型错误 | "Array subscript can only be a int" |
| 28 使用了未定义的变量 | "Var/Array name not found: " |
| 29 数组未定义但有同名变量 | "Array not found but a wrong kind symbol with same name" |
| 30 数组下标越界 | "Array subscript out of range. Legal subscript range: 0-" + to\_string(findresult.arrinfo.max\_capacity - 1) + " Input: " + subscript |
| 31 变量未定义 | "Required kind var/para/const wiht given name not found" |
| 32 因子中正负号后变量类型错误 | "Input: "+lex\_output.key+"\n+/- in a factor must followed by a num" |
| 33 类型不匹配 | "Cannot assign a "+ resulttype +" value to a "+map\_type[findresult.arrinfo.type] |
| 34 给常量赋值 | "Cannot assign another value to a const " |
| 35 switch语句中case需要常量 | "Case list expected a number here" |
| 36 switch输入和case类型不匹配 | "Switch input type (" + inputtype + ") should be the same type with const case" |
| 37 重复的case情况 | "Duplicated cases: " + tmpcase |
| 38 非法的case情况（非常量） | "Illegal case list value: "+lex\_output.key |
| 39 空default语句（要么没有default要么冒号后需要语句） | "Empty default case not allowed" |
| 40 函数返回值类型不匹配（静态类型推导） | "Wrong return value type! Expected: " + expected\_type + " Given: " |
| 41 读取的目标变量未定义 | "Scanf var/para not found: " + lex\_output.key |
| 42 读取的目标变量未定义，但有同名函数或常量 | "Scanf required var/para not found, something else with same name found: " |
| 43 函数未定义 | "Function not found" |
| 44 条件式中操作数类型错误 | "Wrong op type op1 is " + resulttype + " op2 is " + resulttype2 |

**三．操作说明**

**1．运行环境**

本编译器使用C++ 11编写

平台：Microsoft Visual Studio Enterprise 2017

编写及测试环境：Windows 10

汇编编译器：Mars4\_5 (judge网站最新修改版)

**2．操作步骤**

在Visual Studio中运行后，弹出控制台窗口，提示输入需要进行编译的文件路径，请输入该文件所在的**绝对路径**，并按回车确认。如果直接回车，则使用默认测试程序15061200\_test.txt进行编译。

生成的四元式将存放在和代码同路径的syntac\_output­\_before.txt中(优化前)，如果输入文件没有错误，则生成的mips代码将存放在同路径的output\_before.asm中(优化前)。

优化后的中间代码将输出到同路径的syntac\_output\_after.txt， mips代码输出在output\_after.asm。

输入程序的符号表信息会打印在控制台上。

部分优化和生成Mips代码部分函数相结合，故而保留双版本目标代码生成。若使用无优化版本，则将SyntacticAnalyze.cpp首行#include语句改为

|  |  |
| --- | --- |
| 无优化 | #include "genMips.h" |
| 优化 | #include "genMips\_opt.h" |

在myconst.h中有三个宏变量用户可根据需求自行修改

|  |  |  |
| --- | --- | --- |
| 类型/名称 | 默认值 | 含义 |
| bool Verbose | true | 设置为true则不输出语法分析中间结果（语法成分） |
| string Subfix | “before” | 四元式和Mips代码输出文件名后缀，优化后可替换为“after” |
| string Newline | “” | 设置为 “\\n” 即可在每个输出语句后自动换行 |

**四．测试报告**

**1．测试程序及测试结果**

具体测试程序参见测试程序文件夹（输出结果部分，如果为正确程序，则为Mars运行截图，若为错误程序则为控制台输出界面。为节省空间，截图中均省略符号表输出）

**1测试1（正确）15061200\_test.txt**

**预期输出：**

Input two numbers to calculate GCD\*\*\*\* user input : 60

\*\*\*\* user input : 21

Two input for GCD 6021Greatest Common Division of 6021is 3empty parameter list & condition is a single expression2empty parameter list & condition is a single expression1as plain as daylight 1\nas plain as daylight 2apples[2] is567894967 !#$%&'()\*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^\_`abcdefghijklmnopqrstuvwxyz{|}~Global value testp is 0y is nz is nx is 1apples[0] is 1apples[1] is 12340apples[2] is 567891102

-- program is finished running --

**2测试2（正确）test\_funcretpara.txt**

**预期输出：**

param using function return value test passa=720

-- program is finished running --

**3测试3（正确）test\_backtrack.txt**

**预期输出：**

test duplicated name32

-- program is finished running --

**4测试4（正确）test\_func&leftright.txt**

**预期输出：**

1612244

-- program is finished running --

**5测试5（正确）test\_multifunc.txt**

**预期输出：**

38All test passed

-- program is finished running --

**6测试6（错误）test\_wrong1.txt**

**预期输出：**

########Errors Record########

1 Row 5 Syntactic/Semantic error

Code: a = -+'a';

MSG : Input: a

+/- in a factor must followed by a num

2 Row 5 Syntactic/Semantic error

Code: a = -+'a';

MSG : Cannot assign a value to a char

3 Row 7 Syntactic/Semantic error

Code: a = +-'a';

MSG : Input: a

+/- in a factor must followed by a num

4 Row 7 Syntactic/Semantic error

Code: a = +-'a';

MSG : Cannot assign a value to a char

5 Row 9 Syntactic/Semantic error

Code: a = ('a');

MSG : Cannot assign a int value to a char

6 Row 35 Syntactic/Semantic error

Code: if (a != b){

MSG : Wrong op type op1 is char op2 is int

7 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Function not found

8 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Illegal Ch print Expected ;

9 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Unknown Statement type

10 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Wrong Statement in multi statements

11 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Unknown Statement type

12 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Wrong Statement in multi statements

13 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Unknown Statement type

14 Row 36 Syntactic/Semantic error

Code: print("not equal(wrong comparison)");

MSG : Wrong Statement in multi statements

15 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Function not found

16 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Illegal Ch print Expected ;

17 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Unknown Statement type

18 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Wrong Statement in multi statements

19 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Unknown Statement type

20 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Wrong Statement in multi statements

21 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Unknown Statement type

22 Row 39 Syntactic/Semantic error

Code: print("not equal(right comparison)");

MSG : Wrong Statement in multi statements

22 errors, mission failed !

**7测试7（错误）test\_wrong2.txt**

**预期输出：**

########Errors Record########

1 Row 2 Syntactic/Semantic error

Code: char a, b, c;

MSG : Duplicated var/para/array/const Declaration in the same level

2 Row 2 Syntactic/Semantic error

Code: char a, b, c;

MSG : Duplicated var/para/array/const Declaration in the same level

3 Row 2 Syntactic/Semantic error

Code: char a, b, c;

MSG : Duplicated var/para/array/const Declaration in the same level

4 Row 9 Syntactic/Semantic error

Code: a[-1] = 9;

MSG : Array subscript out of range. Legal subscript range: 0-99 Input: -1

5 Row 10 Syntactic/Semantic error

Code: a[100] = 10;

MSG : Array subscript out of range. Legal subscript range: 0-99 Input: 100

5 errors, mission failed !

**8测试8（错误）test\_wrong3.txt**

**预期输出：**

########Errors Record########

1 Row 5 Syntactic/Semantic error

Code: }

MSG : func need a return value

1 errors, mission failed !

**9测试9（错误）test\_wrong4.txt**

**预期输出：**

########Errors Record########

1 Row 6 Syntactic/Semantic error

Code: b=1;

MSG : Cannot assign a int value to a char

2 Row 9 Syntactic/Semantic error

Code: b = a;

MSG : Cannot assign a int value to a char

3 Row 10 Syntactic/Semantic error

Code: a = b;

MSG : Cannot assign a char value to a int

3 errors, mission failed !

**10测试10（错误）test\_wrong5.txt**

**预期输出：**

########Errors Record########

1 Row 11 Syntactic/Semantic error

Code: if (a){

MSG : Wrong op type op1 is char op2 is

1 errors, mission failed !

**2．测试结果分析**

**1 测试1（正确）15061200\_test.txt**

本程序是最基本的一个测试程序。首先重点测试了语法和词法分析对于文法的覆盖性，然后是对于嵌套的while, if, switch，scanf, printf语句等的功能性测试。同时还包括了基本的表达式运算，函数调用，函数返回等。同时测试了基础的函数递归调用。

程序的功能是输入两个数字即可输出对应的最大公约数。如果输入的第一个数字是100的话，会有额外的测试信息输出。结果正确。

**2测试2（正确）test\_funcretpara.txt**

本程序重点测试函数返回值作为下一个函数调用参数的情况。

程序的功能是默认计算6的阶乘并输出结果，输出720。结果正确。

**3测试3（正确）test\_backtrack.txt**

本程序重点测试语法分析中有部分文法成分需要回溯检测的情况，以及混合了局部变量和全局变量的表达式运算。

程序的功能是计算全局变量和局部变量的相关表达式，结果正确。

**4测试4（正确）test\_func&leftright.txt**

本程序重点测试表达式和条件式运算顺序的正确性，也即是否遵循从左到右的原则。

程序的功能是对代码中的表达式进行求值并输出结果，结果正确。

**5测试5 （正确）test\_multifunc.txt**

本程序综合测试了多个有参或者无参函数互相嵌套调用的情况，着重测试了参数寄存器的使用和多个参数混合函数返回值作为参数的情况。

程序功能为计算代码中的表达式，结果正确。

**6测试6（错误）test\_wrong1.txt**

本程序主要测试正负号和<项>。

文法要求<项>之前可以有正负号，但是<因子>中只有整数带有额外的符号，故而只有整数前面可能出现两个符号。

**7测试7（错误）test\_wrong2.txt**

本程序主要测试数组越界和变量重复声明。

**8测试8（错误）text\_wrong3.txt**

本程序主要测试函数返回值缺失的问题。

**9测试9（错误）test\_wrong4.txt**

本程序主要测试表达式计算中类型不匹配的问题。

根据论坛约定，字符型只有单独出现的时候才能保持字符型，否则任何计算都会导致其类型自动转换。同时任何计算中都要求保证类型严格匹配。故而在赋值中报错。

**10测试10（错误）test\_wrong5.txt**

本程序主要测试条件式中类型错误的问题。

同上，根据论坛约定，条件式中只能是整型。

总的来说，本测试着重覆盖了一些论坛约定以及比较细节的错误，对于比较普通和常见的语法语义错误，已经直接在文档中给出，便不再单独另行测试，本编译器支持的全部错误类型和输出信息请参考前文第二章节第10部分。

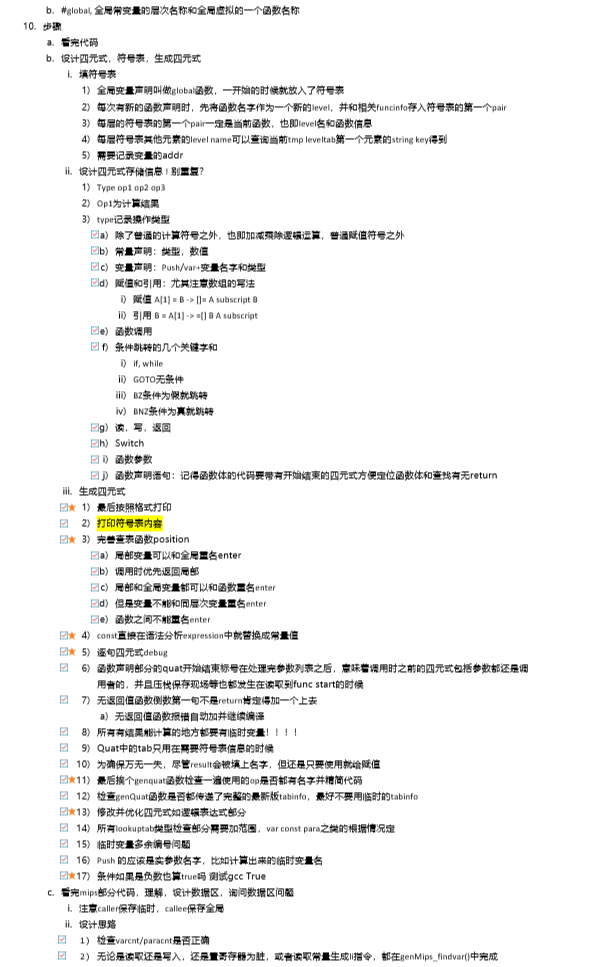
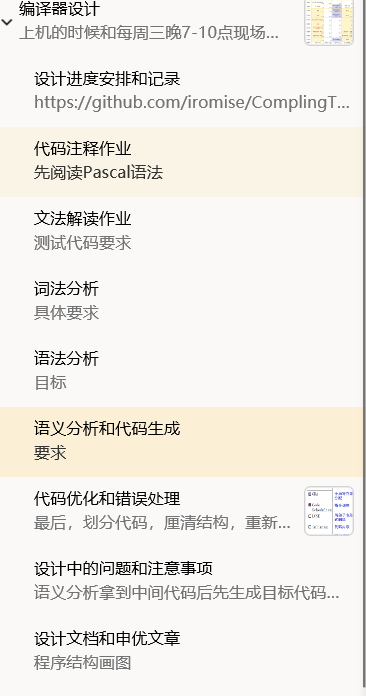
**五．总结感想**

我是15级同学，由于赴香港交换导致延迟选课（香港科技大学竟然不开设编译原理，因为学生们连年投诉课程太难）。虽然我也曾后悔没能在交换时避开编译，但这也让我似乎成为了今年唯一一个非重修选编译课设的大四同学，从而拥有了一段奇妙的体验和经历。现在再次回顾这个学期，我反而开始庆幸自己修习了这里的编译原理理论和课设。

首先编译课设是计算机学院核心专业大课中综合程度最高的一个，在学习了计组，操统之后，编译器实现中将以上两门课程的一些精华思想都融入了进来，我需要理解运行栈和机器运行时的内存空间，我需要理解mips体系结构的一些特征和相关机器指令（虽然最后优化囿于现实条件只能统计指令运行数量），我需要理解从阅读Pl0和pascal编译器中得到的设计经验。而巨大的工程量和较长的工程周期，又对我们的软件设计和时间安排提出了很高的要求，从另一个层面上考察了我们的数据结构，算法，综合编码能力和debug能力。所有的这一切我认为都是对于我们大学四年学习成果最好的一次综合测试，因此我很感谢学院的课程安排和课程组老师们设计出来的如此精彩的课设。

另外一个感受是编译课设和我们的理论学习结合程度很高，尤其是在蒙头写了一段时间后，出现了奇怪的bug，debug良久不出结果，回去翻理论课本，发现是自己某些地方理解出了问题。这种理论和实践紧密的结合和循环反馈，不仅夯实了我对于理论的理解，也促进了我对于整个编译理论知识体系的构建。同时也是一种思维训练，在纠正自己设计的同时，根据理论来寻找自己思考过程中的漏洞，提升自己的逻辑思维。比如在第一次测试前夕我的代码中最严重的一个错误是我在没有做基本块划分和数据流分析的时候就想当然地分配了全局寄存器，并很认真地逐句mips代码模拟debug了一遍，当我以为设计了很完善的写回和记录机制的时候，我的代码却一直通不过自己的测试程序，一直到提交前一天我才发现全局寄存器的数值在循环中是错误的。翻书再三确认，我才发现在数据流分析之前sreg基本无法使用，只能从栈上直接读写。这也给我敲响了警钟，设计的时候不能想当然，每一步都应该有完善的考虑和理论支撑，有时候自以为精妙的设计实际上只是一个隐藏得更深的bug，而这种bug一般是很难改正的，因为设计的时候我们都已经用上了自己100%的“小聪明”（快速改正它可能需要3个我）。类似这种苦笑不得的时刻，在整个学期中还出现了很多次，也算是对自己抗压能力的一次锻炼了。

从工程实现的角度来说，我发现自己容易掉进一开始就优化的陷阱中。我在一些编译器的设计选择上容易患得患失，总想一开始就尽可能设计一个尽善尽美的成品出来，导致一则工期缓慢，二则细节考虑过多实现难度大，三则考虑过多可能反而误入歧途（如一开始的分配全局寄存器）。后来我不断提醒自己，一步步来，先实现一个能跑的，在逐渐完善，将任务拆解成阶段性目标，不要妄想一口气完成。这样我的工程质量和实现速度才逐渐稳定下来。另外一个感受就是在项目内容很多的时候，及时记录详细的笔记和任务清单是非常有必要的，下面是两张我的onenote笔记截图



我认为，这次课设的顺利完成很大一部分都要归功于我完整的信息记录和笔记。通过笔记和任务清单，我可以方便地梳理自己的思路并厘清繁杂的工程任务中最关键的部分（当然也保证了自己不忘掉一些细节）。

最后，我很高兴能够完成这个学期的编译原理课设，虽然优化部分因为期末申请加生病的缘故没能很好地完成，不过前半学期的设计/debug/和同学的交流，已经让我获益匪浅。再次感谢课程组为我们提供的这次宝贵的学习机会，希望这门课程可以越来越好。