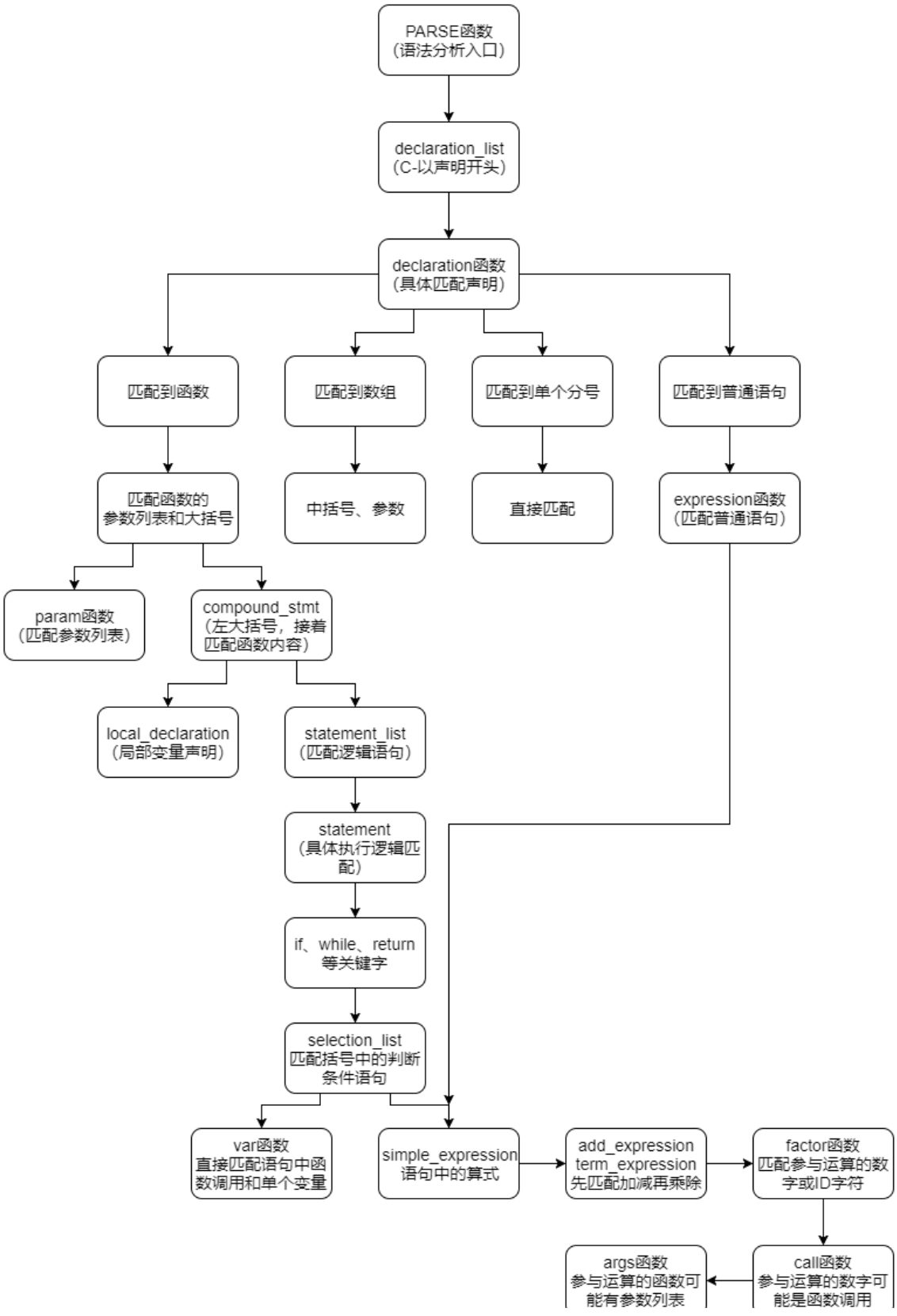
《编译原理》课程设计实验报告书

班 级：1618204
学 号：161820127
姓 名：廖俊轩
指导老师：李静
课设地点：实验楼103
课设时间：2020年12月6日

2020年 12月 1日

c-语言的语法描述



系统设计

符号表的实现

符号表采用了哈希表的形式，可以方便地查找、插入和删除，但是问题也随之而来，就是符号的作用于较难跟踪。很有可能同一名称的变量在不同作用于出现，如果孤立地对待每一个变量便会出现变量冲突。因此的符号表将作用域做了栈式处理。再将栈中每一个元素，即每一个作用域做延伸，对每一个作用域建立一个哈希表，将范围缩小。这样一来可以对每一个变量框在一个作用域中进行跟踪。

中间代码生成

的中间代码生成运用一遍扫描的方法自下而上归约，但是和课本的方法有出入，进行了一些改变。首先，建立一个链表存储所有的生成的四元式，每个四元式的本质是一个结构体。其中，在对if-else或是while这种需要回填的部分处理时，先对判断条件进行判断，结果存在一个临时变量`t`之中，这个变量的值非0即1.

当面临if语句规约时，判断的是`t`的值，真出口就是继续执行的下一条语句，执行完语句之后，生成了又一个临时变量L，并且新增一个操作叫做`label`，它用来指示整个if-else语句的结尾，这个label四元式是在控制语句的结尾，当生成`（label，L2，-，-）`的同时，顺序查找四元式链表中`j_if_false`后紧跟的`（j,-,-,-）`，并将其填为`（j,L2,-,-）`，代表无条件跳转到L2。

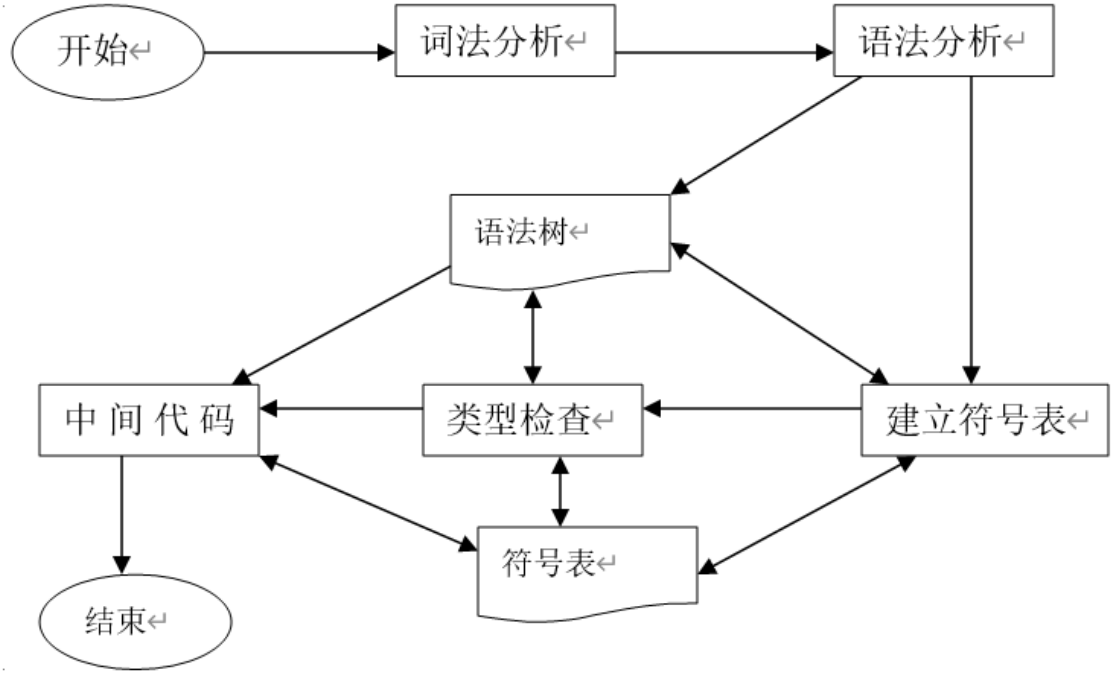
对于假出口的处理同理，只不过是在else语句开始之前就生成一个`（label，L1，-，-）`，记下标号，并且在生成之后顺序查找四元式链表，找到`j_if_false`开头的代码进行回填。这样，真假出口的回填就完成了。其他的四元式根据不同节点的类型直接生成不同的四元式，最后将四元式链表输出到文件当中即可

```
********** Intermediate Code **********

1: (function, f1, _, _)
2: (return, 1, _, _)
3: (function, f2, _, _)
4: (get_param, y, _, _)
5: (=, y, 2, _)
6: (return, _, _, _)
7: (function, f3, _, _)
8: (=, y, 3, _)
9: (return, _, _, _)
10: (function, main, _, _)
11: (param_list, 0, _, _)
12: (call, f1, 0, t1)
13: (param_list, 0, _, _)
14: (call, f1, 0, t2)
15: (+, 2, t2, t3)
16: (=, x, t3, _)
```

系统的总体结构

结构框图如图所示，共分为五大部分以及两个辅佐部分，五大部分为词法分析、语法分析、建立符号表、类型检查和中间代码，辅佐部分为符号表和语法树。



主要功能模块的设计

词法分析和语法分析：利用flex & bison工具实现，在`cminus.l`文件中写好词法规则和`getToken`函数，在`cminus.y`文件中写好BNF语法和其他相关函数即可，flex和bison可以帮助生成`lex.yy.c`文件和`.tab.`文件。

语义分析：生成语法分析树和符号表，借助这两个数据结构分析语义。

中间代码生成：通过一遍扫描和回填技术实现中间代码生成。

系统运行流程

C-minus编译器运行于Linux系统。编译完成后即可运行，运行命令

```
./{编译好后的程序的名字}  ./{目标程序的位置}
```

结果存放于同名txt文件中

系统实现

系统主要函数说明（主要功能、输入\输出、实现思想）

CMINUS.L

这一文件的主要功能为flex分析器的文件。第一部分为定义，flex工具在处理这一部分的时候，会将第一部分的所有代码原封不动的插入到 lex.yy.c（词法分析代码）中，在这一部分中插入了程序需要用到的头文件等内容。第二部分是词法规则部分：正则表达式+代码片段。当匹配相对应的正则表达式时，这些代码片段就会被执行。的代码片段为一个单词符号的编码。最后一个部分包括辅助函数，它们用于在第2个部分被调用且不在任何地方被定义的辅助程序。

CMINUS.Y

这一文件的主要功能是产生语法分析代码。首先是定义部分：定义部分包括bison需要用来建立分析程序的有关记号、数据类型以及文法规则的信息，还包括了一些头文件。规则部分：包括修改的BNF格式中的文法规则以及将在识别出相关的文法规则时被执行的C代码中的动作。根据每一条产生式执行不同的语义动作，例如生成节点、赋上相应属性等等。最后是用户自定义函数部分，定义了几个插入节点的函数，可以和语义动作对应起来。最后这个文件会被建立成两个`.tab.`程序，实现语法分析和语法树的建立。

BUILDSYMTAB

Cminus的语义分析器在`analyze.c`和`symtab.c`中，其对应的主要功能，`analyse.c`是用于语义分析本身，而`symtab.c`则是用于生成其对应的符号表。

进入语义分析部分的代码在main.c中：

```
#if !NO_ANALYZE
    if (! Error) {
        if (TraceAnalyze) fprintf(listing, "\nBuilding the symbol table...\n");
        buildSymtab(syntaxTree);
        if (TraceAnalyze) fprintf(listing, "\nChecking type...\n");
        typeCheck(syntaxTree);
        if (TraceAnalyze) fprintf(listing, "\nType check completed!\n");
    }
```

在语义分析之前，编译器最开始的输入是一段代码，经过词法分析，输出的是词法单元，从而被语法分析单元所识别；语法分析的输入是一个个词法单元，通过分析这些词法单元之间的逻辑，利用递归下降等方法，形成一棵语法树，并将语法树的根结点存储在一个TreeNode类中，从而通过根结点就可以实现对于整个语法树的遍历（一般是前序遍历）；之后，来到了语义分析部分，语义分析的输入是一个语法树，这里我的输入是根结点；语义分析的输出，则是符号表和语法报错信息。

符号表的意义在于，分析代码中所有的声明，比如变量函数等内容；而语法报错信息，则会通过语法树结点关系，检测相邻词法单元是否符合文法规则：比如，int 1和int a两种输入，在语法分析阶段均可通过，但是在语义分析阶段，int 1会被识别为一个错误，因为根据语法规则，int是一个声明，声明后面只能跟着一个变量名ID，而词法单元1的属性是NUM，int后面是不允许接着一个NUM的。

函数buildSymtab构造符号，通过语法树的遍历构建。

```
/* Function buildSymtab constructs the symbol
 * table by preorder traversal of the syntax tree
 */
void buildSymtab(TreeNode * syntaxTree) {
    globalScope = sc_create((char *) "ESCOPO_GLOBAL");
    sc_push(globalScope);
    traverse(syntaxTree, insertNode, afterInsertNode);
    sc_pop();
    sys_free();
    if(mainDeclaration == FALSE) {
        fprintf(listing, "Declaration Error: Undeclared main function\n");
        return;
    }
    if (TraceAnalyze) {
        fprintf(listing,"\nSymbol Table:\n");
        printSymTab(listing);
    }
}
```

该函数共分为四个步骤，首先调用sc_create（）函数创建一个作用域栈，作用域栈用于储存一个符号的作用域，如果给出作用域名称作为参数，它将分配内存并填写每个属性。在初始化的时候，如果是全局变量，则它就是最大的全局作用域，如果不是，则把指针设置为该作用域所属的较大作用域的父作用域。

l例如，名为main的作用域将global作为其父作用域。 在main中创建另一个作用域时，它以相同的方式堆叠在栈上。 当main结束时（遇到复合语句时），main作用域弹出并查看全局作用域中的其余TreeNodes。

```
Scope sc_create(char * funcName) {
    Scope newScope = (Scope) malloc(sizeof(struct ScopeRec));
    newScope->funcName = funcName;
    newScope->tamanhoBlocoMemoria = 0;
    if(!strcmp(funcName, "ESCOPO_GLOBAL")) {
        newScope->parent = NULL;
    } else {
        newScope->parent = globalScope;
    }
    scopes[nScope++] = newScope;
    return newScope;
}
123456789101112
```

之后，调用sc_push（）函数，将全局作用域先推入栈中：

```
void sc_push(Scope scope) {
    scopeStack[nScopeStack++] = scope;
}
123
```

接着，调用traverse（）函数，它采用深度遍历，即先遍历preProc（t）以及它的子节点，再遍历postProc（t）。

```
/* Procedure traverse is a generic recursive
 * syntax tree traversal routine:
 * it applies preProc in preorder and postProc
 * in postorder to tree pointed to by t
 */
static void traverse( TreeNode * t,
               void (* preProc) (TreeNode *),
               void (* postProc) (TreeNode *) )
{ if (t != NULL)
  { preProc(t);
    { int i;
      for (i=0; i < MAXCHILDREN; i++)
        traverse(t->child[i],preProc,postProc);
    }
    postProc(t);
    traverse(t->sibling,preProc,postProc);
  }
}
12345678910111213141516171718
```

最后，调用printSymTab（）打印函数，间接调用printSymTabRows（）：打印符号表的一行和insertNode（）：把一个节点插入到符号表中，最终将打印出所有的函数名和每个函数里所有的变量名，打印项目有名字、id类型、数据类型、作用域、大小（以字为单位）、相对位置和在源程序中出现的行数。

ST_INSERT（）、ST_CREATE（）

将bucketList添加到作用域栈中， 使用hash函数获取哈希值。 从scope_top搜索指定的作用域名称，直到找到正确的作用域，并将bucketList放在指定作用域的bucket数组中的哈希值处。 此时，如果没有相同的bucketList，则添加一个新的bucketList。

```
void st_insert(char * name, int lineno, int loc, TreeNode * treeNode, int
tamanho) {
    int h = hash(name);
    Scope top = sc_top();

    if(top->hashTable[h] == NULL) {
    // Add scope to syntax tree node
        treeNode->kind.var.scope = top;
        top->hashTable[h] = st_create(name, lineno, loc, treeNode, tamanho);
        treeNode->kind.var.scope->tamanhoBlocoMemoria += tamanho;
    } else {
        BucketList l = top->hashTable[h];
        while ((l->next != NULL) && (strcmp(name, l->name))) {
            l = l->next;
        }
        if (l->next == NULL && (strcmp(name, l->name))) { /* Variable not yet in
table */
            //Add scope to syntax tree node
            treeNode->kind.var.scope = top;
            //Add a new item to the symbol table
            l->next = st_create(name, lineno, loc, treeNode, tamanho);
            treeNode->kind.var.scope->tamanhoBlocoMemoria += tamanho;
        }
    }
} /* st_insert */

BucketList st_create(char * name, int lineno, int loc, TreeNode * treeNode, int
tamanho) {
    BucketList l = (BucketList) malloc(sizeof(struct BucketListRec));
    l->name = name;
    l->lines = (LineList) malloc(sizeof(struct LineListRec));
    l->lines->lineno = lineno;
    l->lines->next = NULL;
    l->memloc = loc;
    l->tamanho = tamanho;
    l->treeNode = treeNode;
    l->next = NULL;
    return l;
} /* st_create */
12345678910111213141516171819202122232425262728293031323334353 6
```

ST_LOOKUP()、ST_LOOKUP_TOP（）

st_lookup()从当前堆栈顶部的范围中查找父范围。st_lookup_top仅在当前作用域内查找，它以范围名称作为参数，并仅检查该范围的BucketList。

TYPECHECK（）

在建立完符号表之后，语义分析的下一个任务是对于语法错误的检查，该部分的实现原理是，通过检查相邻语法树结点的词法属性，确保是符合常规的。具体的实现函数在`analyse.c`中的`type check`函数：

```
/* Procedure typeCheck performs type checking
 * by a postorder syntax tree traversal
 */
void typeCheck(TreeNode * syntaxTree) {
    traverse(syntaxTree, beforeCheckNode, checkNode);
}
123456
```

typeCheck直接调用了一个遍历函数traverse，传入的三个参数，第一个是syntaxTree，代表语法树，第二个beforeCheckNode，它是一个函数，代表遍历的终止条件，即遍历到叶子结点时，语法树遍历停止，第三个函数则是需要重点分析的checkNode，它制定了语法规则分析的过程。

```
static void beforeCheckNode(TreeNode * t) {
    if (t->node == EXPK) {
        if (t->kind.var.varKind == FUNCTIONK) {
            funcName = t->kind.var.attr.name;
        }
    }
}
```

check node函数的工作原理是，一边遍历语法树，遍历到当前的一个结点之后，检查该结点的相邻结点是否符合语法规则，如果不符合就报错，如果符合就可以继续。

例如：如果该节点是`t->node == EXPK`，那么它有三种情况，第一种是赋值语句，那么它的孩子节点的类型必须相同；如果是判断表达式，那么结点类型必须是void；如果是算术表达式，它的类型必须是int，这样对应起来。

```
/* Procedure checkNode performs
 * type checking at a single tree node
 */
static void checkNode(TreeNode * t) {
    if (t->node == STMTK) {
        switch (t->kind.stmt) {
            case INTEGERK: t->child[0]->type = INTEGER_TYPE; break;
            case VOIDK:
                if (t->child[0]->kind.var.varKind == IDK) {
                    typeError(t, "Variable cannot be void!");
                } else {
                    t->child[0]->type = VOID_TYPE;
                }
                break;
            case IFK: break;
            case WHILEK: break;
            case RETURNK: break;
            case COMPK: break;
        }
    } else if (t->node == EXPK) {
        switch (t->kind.exp) {
            case ATRIBK:
```

```
                if ((t->child[0]->type != INTEGER_TYPE) || (t->child[1]->type !=
INTEGER_TYPE)) {
                    typeError(t, "Invalid assignment, int value expected and
void received");
                } else  {
                    t->type = INTEGER_TYPE;
                }
                break;
            case RELK: t->type = VOID_TYPE; break;
            case ARITHK: t->type = INTEGER_TYPE; break;
        }
    } else if (t->node == VARK) {
        switch (t->kind.var.varKind) {
            case IDK: t->type = INTEGER_TYPE; break;
            case VECTORK: t->type = INTEGER_TYPE; break;
            case CONSTK: t->type = INTEGER_TYPE; break;
            case FUNCTIONK: break;
            case CALLK: break;
        }
    }
}
```

至此，语义分析结束了。

CODEGEN（）

接下来的中间代码生成则也是对于语法树进行操作，传入一棵语法树，从根结点，根据该节点的词法属性，分析词法结点之间的逻辑，翻译成四元式。

进入中间代码生成部分的代码在main.c中，将中间代码写入同名的txt文件中，strcspn函数的作用是，在pgm字符串中查找到第一个“.”，并返回它之前的所有字符作为子串，这样做的目的在于，传入给编译器的文件是一个文件，、中间代码的输出结果也需要保存在一个文件中，输出文件在这里这样做，是为了保持和输入文件同名。strncpy函数的作用是拷贝字符串，这里用于将strcspn提取的文件名存入字符串供输出文件使用。

```
#if !NO_CODE
    if (! Error) {
        char * codefile;
        int fnlen = strcspn(codeInfo.pgm, ".");
        codefile = (char *) calloc(fnlen + 4, sizeof(char));
        strncpy(codefile, codeInfo.pgm, fnlen);
        strcat(codefile, ".txt");
        code = fopen(codefile, "w");
        if (code == NULL) {
            printf("Cannot open the file %s\n", codefile);
            exit(1);
        }
        if (TraceCode) fprintf(listing, "\nGenerate intermediate code...\n");
        codeGen(syntaxTree, codefile, codeInfo);
        free(syntaxTree);
        fclose(code);
        if (TraceCode) fprintf(listing, "\nIntermediate code generation is
complete!\n");
    }
```

codeGen函数传入了语法树根结点以及需要写入的文件，进行后续操作

```
/* Procedure codeGen generates code to a code
 * file by traversal of the syntax tree. The
 * second parameter (codefile) is the file name
 * of the code file, and is used to print the
 * file name as a comment in the code file
 */
void codeGen(TreeNode * syntaxTree, char * codefile, CodeInfo codeInfo) {
    cGen(syntaxTree);
        // If it is a Common Program code, add SYSCALL to the end of the code.
    insertQuad(createQuad(SYSCALL, NULL, NULL, NULL));
    emitCode("********** Intermediate Code **********\n");
    printIntermediateCode();
}
12345678910111213
```

这和函数主要调用了 cGen（）、insertQuad（）和printIntermediateCode（）

INSERTQUAD（）

生成四元式函数。四元式由一个结构体构成。

```
Quadruple * insertQuad(Quadruple q) {
    Quadruple * ptr = (Quadruple *) malloc(sizeof(struct Quad));
    if(head == NULL) {
        head = q;
        head->next = NULL;
        ptr = &head;
    } else {
        Quadruple temp = head;
        while(temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = q;
        temp->next->next = NULL;
        ptr = &temp->next;
    }
    return ptr;
}
12345678910111213141516171819
```

CGEN()

```
static void cGen(TreeNode * tree) {
    if (tree != NULL) {
        switch (tree->node) {
            case STMTK:
                genStmt(tree);
                break;
            case EXPK:
                genExp(tree);
                break;
            case VARK:
                genVar(tree);
                break;
            default:
                break;
```

```
        }
        /*If the number of parameters is greater than 0, cGen () will be called
automatically*/
        if(paramHead == NULL) {
            cGen(tree->sibling);
        } else {
            if(paramHead->count == 0) {
                cGen(tree->sibling);
            }
        }
    }
}
```

可以看到，传入的语法树在一边遍历的同时，检查结点的类型，分为三种语句结点，一种是STMTK，主要分析保留字和复合语句；一种是EXPK，主要分析表达式，一种是VARK，主要分析不同类型的变量。

genStmt（）：该函数的作用主要是处理Cminus中所包含的关键字——int，void，if，while，return和compound。以`if`作为一个例子来分析说明这个函数需要做的工作：`if`结点包括三个子结点：`if`本身的判断表达式、复合语句、以及else。对于每个if的子结点递归分析，因为if的子结点可能也会是一个表达式，比如if的条件判断，这样的话就需要对它进行递归分析。

```
 case IFK:
            p1 = tree->child[0];
            p2 = tree->child[1];
            p3 = tree->child[2];
            /* Generate code for test expression */
            cGen(p1);
            /* Assigns as the first operand*/
            op1 = operandoAtual;
            /* Assigns instruction type */
            instrucaoAtual = JP;
            /* Create and insert a new intermediate code representation*/
            q = insertQuad(createQuad(instrucaoAtual, op1, NULL, NULL));
            /* Save if's IR to update with label representing end of block */
            pushLocation(q);
            /* Generate code for block */
            cGen(p2);
            /* set second operand */
            op2 = createOperand();
            op2->kind = String;
            op2->contents.variable.name = createLabelName();
            op2->contents.variable.scope = tree->kind.var.scope;
            /* update if intermediate code instruction */
            updateLocation(op2);
            popLocation();
            if(p3 != NULL) {
                q = insertQuad(createQuad(GOTO, NULL, NULL, NULL));
                pushLocation(q);
            }
            /* Label used to mark end of block */
            insertQuad(createQuad(LBL, op2, NULL, NULL));
            cGen(p3);

            if(p3 != NULL) {
                op1 = createOperand();
                op1->kind = String;
                op1->contents.variable.name = createLabelName();
```

```
                op1->contents.variable.scope = tree->kind.var.scope;
                /* update if intermediate code instruction */
                updateLocation(op1);
                popLocation();

                /* Label used to mark end of block else*/
                insertQuad(createQuad(LBL, op1, NULL, NULL));
            }
            break; /* IFK */
```

`instrucaoAtual`用来标记四元式的第一区间，即操作类型，它是一个枚举类型，记录在cgen.h中：

```
typedef enum instrucao {ADD, SUB, MULT, _DIV, MOD,
    VEC, VEC_ADDR,
    EQ, NE, _LT, LET,_GT, GET, ASN,
    FUNC, RTN, GET_PARAM, SET_PARAM, CALL, PARAM_LIST,
    JP, GOTO, LBL, SYSCALL, HALT} InstructionKind;
```

pushLocation（）用于保存if的地址，记录递归的返回位置，待分析完这一条路径之后，就可以找到函数在哪里被调用了，在调用的过程中，由于各个节点都需要递归地访问，实现回填。

其他的处理也是类似的，比如在while语句里面，包含的是两个结点，一个是循环它本身，第二个是与之对应的条件，同if一样，分为两块进行分别的一个递归处理。

genExp（）：处理表达式结点的逻辑比较简单，如果表达式的结点类型是ID或者数字，就直接记录它，但有一个需要特殊处理的地方就是数组，在赋值中，如果左操作数是数组，则存储此变量的内存位置。如果结点是一个运算符，那么据所知，运算符是由子结点构成的，它的子结点就是运算的两个数字，或者是ID字符。需要使用LD命令将子结点里面的具体数字或字符读取出来，再根据运算符的类型构造相应的四元式。

genVar（）：对于CONSTK、IDK、VECTORK，FUNCTIONK，在四元式中记录他们的类型和名字、CALLK节点记录跳转的函数，参数个数和参数列表的地址。