



湖南大学
HUNAN UNIVERSITY

课程实验报告

课 程 名 称: 编译技术
实验项目名称: 代码生成器
专 业 班 级: 软件 2005 班
姓 名: 邹佳骏
学 号: 202026010501
指 导 教 师: 杨金民
完 成 时 间: 2023 年 5 月 28 日

信息科学与工程学院

实验题目：代码生成器

实验目的：学习已有编译器的经典代码生成源程序，加深对代码生成的理解，学会编制代码生成器。

实验环境：Windows10、Visual Studio 2022

实验内容及操作步骤：

实验内容：这次实验分为两个部分，即：

1. 学习经典的代码生成器：

- 1) 选择一个编译器，如：TINY 或 PL/0。
- 2) 阅读 TM 虚拟机的有关文档。了解 TM 机的指令结构与寻址方式等相关内容。
- 3) 阅读代码生成源程序，加上自己的理解。尤其要求对相关函数与重要变量的作用与功能进行稍微详细的描述。
- 4) 测试代码生成器。请对生成的目标代码逐行加上注释，增强其可读性。

测试用例一：sample.tny。

测试用例二：用 TINY 语言自编一个程序计算任意两个正整数的最大公约数与最大公倍数。

2. 实现一门语言的代码生成器：































- 1) 语言确定：C-语言，其定义在《编译原理及实践》附录 A 中
- 2) 完成对 TM 机的改造设计，以满足 C-语言的要求。
- 3) 仿照前面学习的代码生成器，编写选定语言的代码生成器。
- 4) 准备 2~3 个测试用例，测试你的程序，并逐行解释生成的目标代码。

操作步骤：

学习经典的代码生成器

一、 选择一个编译器

选择 TINY 编译器。

	ANALYZE.C	2022/3/23 1:	
	ANALYZE.H	1998/8/1 14:	
	bison.exe	2005/3/18 16:	
	CGEN.C	2022/3/22 2:	
	CGEN.H	2022/3/23 1:	
	cifa.exe	2022/3/23 20:	
	CODE.C	2022/3/23 1:	
	CODE.H	2022/3/23 1:	
	flex.exe	2005/3/18 16:	
	GLOBALS.H	2022/3/23 20:	
	lex.yy.c	2022/3/23 20:	
	MAIN.C	2022/3/23 20:	
	PARSE.C	2022/3/23 1:	
	PARSE.H	2022/3/23 1:	
	README.DOS	1998/7/31 1:	
	SAMPLE.TM	1998/7/31 16:	
	SAMPLE.TNY	1996/8/25 1:	
	SCAN.C	2022/3/23 1:	
	SCAN.H	2022/3/23 1:	
	SYMTAB.C	2022/3/23 1:	
	SYMTAB.H	2022/3/23 1:	
	TINY.EXE	2022/3/22 22:44	
	TINY.L	1998/7/31 14:45	
	tiny.tab.c	2022/3/23 21:05	
	tiny.tab.h	2022/3/23 21:05	
	TINY.Y	1997/2/1 9:33	
	TINY.Y.tab.c	2022/3/23 21:00	
	TM.C	2022/3/23 17:27	
	TM.EXE	2022/3/22 22:46	
	UTIL.C	2022/3/23 17:27	
	UTIL.H	2022/3/23 17:27	

二、 阅读 TM 虚拟机的有关文档，了解 TM 机的指令结构与寻址方式等相关内容

Tiny Machine 由只读指令存储区、数据存储区、8 个通用寄存器

```

/***** const *****/
#define IADDR_SIZE 1024 /* increase for large programs */
#define DADDR_SIZE 1024 /* increase for large programs */
#define NO_REGS 8
#define PC_REG 7

```

```

INSTRUCTION iMem[IADDR_SIZE];
int dMem[DADDR_SIZE];
int reg[NO_REGS];

```

NO_REGS 表示通用寄存器的个数，PC_REG 表示程序计数器

iMem[IADDR_SIZE]定义了只读指令存储区大小

dMem[DADDR_SIZE]定义了数据存储区大小

reg[NO_REGS]定义了通用寄存器

上图为 TM 的指令集。基本指令格式有两种：寄存器，即 RO 指令。寄存器-存储器，即 RM 指令。

RO 指令

格式	<i>opcode r, s, t</i>
操作码	效果
HALT	停止执行(忽略操作数)
IN	$\text{reg}[r] \leftarrow \text{从标准读入整形值}(s \text{ 和 } t \text{ 忽略})$
OUT	$\text{reg}[r] \rightarrow \text{标准输出}(s \text{ 和 } t \text{ 忽略})$
ADD	$\text{reg}[r] = \text{reg}[s] + \text{reg}[t]$
SUB	$\text{reg}[r] = \text{reg}[s] - \text{reg}[t]$
MUL	$\text{reg}[r] = \text{reg}[s] * \text{reg}[t]$
DIV	$\text{reg}[r] = \text{reg}[s] / \text{reg}[t]$ (可能产生 ZERO_DIV)

RM 指令

格式	<i>opcode r, d(s)</i>
(a=d+reg[s]; 任何对 dmem[a] 的引用在 a<0 或 a≥DADDR_SIZE 时产生 DMEM-ERR)	
操作码	效果
LD	$\text{reg}[r] = \text{dMem}[a]$ (将 a 中的值装入 r)
LDA	$\text{reg}[r] = a$ (将地址 a 直接装入 r)
LDC	$\text{reg}[r] = d$ (将常数 d 直接装入 r, 忽略 s)
ST	$\text{dMem}[a] = \text{reg}[r]$ (将 r 的值存入位置 a)
JLT	if (reg[r]<0) reg[PC_REG] = a (如果 r 小于零转移到 a, 以下类似)
JLE	if (reg[r]<=0) reg[PC_REG] = a
JGE	if (reg[r]>0) reg[PC_REG] = a
JGT	if (reg[r]>0) reg[PC_REG] = a
TEQ	if (reg[r]==0) reg[PC_REG] = a
JNE	if (reg[r]!=0) reg[PC_REG] = a

TM 执行一个常用的取指令-执行循环。

在开始点, TM 将所有寄存器和数据区设为 0, 然后将最高正规地址的值 (名为 DADDR_SIZE-1) 装入到 dMem[0] 中。(由于程序可以知道在执行时可用内存的数量, 所以它允许将存储器很便利地添加到 TM 上)。TM 然后开始执行 iMem[0] 指令。机器在执行到 HALT 指令时停止。可能的错误条件包括 IMEM_ERR (它发生在取指步骤中若 reg[PC_REG]<0 或 reg[PC_REG]≥IADDR_SIZE 时), 以及两个条件 DMEM_ERR 和 ZERO-DIV。

三、阅读代码生成源程序

代码生成源程序主要位于 CODE.C、CODE.H、CGEN.C、CGEN.H 中, 接下来将进行阅读分析。

首先看 CODE.H 和 CODE.C 文件

首先是寄存器值的定义

```
/* pc = program counter */
#define pc 7

/* mp = "memory pointer" points
 * to top of memory (for temp storage)
 */
#define mp 6

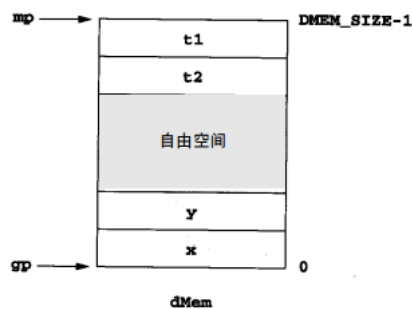
/* gp = "global pointer" points
 * to bottom of memory for (global)
 * variable storage
 */
#define gp 5

/* accumulator */
#define ac 0

/* 2nd accumulator */
#define ac1 1
```

pc 寄存器用来记录当前命令的位置，是必不可少的。

mp 和 gp 两个寄存器分别代表内存指针和全局指针，用来指示存储区的顶部和底部。mp 将用于访问临时变量，并且包含最高正规内存位置，而 gp 则用于访问所有变量。



而 ac 和 ac1 则是累加器，通常计算结果放入 ac 中。

除此之外，寄存器 2、3、4 没有进行命名。

```
/* Procedure emitComment prints a comment line
 * with comment c in the code file
 */
void emitComment(char *c) { if (TraceCode) fprintf( Stream: code, Format: "%s\n", c); }
```

该函数是用来打印注释行。

```
/* Procedure emitRO emits a register-only
 * TM instruction
 * op = the opcode
 * r = target register
 * s = 1st source register
 * t = 2nd source register
 * c = a comment to be printed if TraceCode is TRUE
 */
void emitRO(char *op, int r, int s, int t, char *c) {
    fprintf( Stream: code, Format: "%3d: %5s %d,%d,%d ", emitLoc++, op, r, s, t);
    if (TraceCode) fprintf( Stream: code, Format: "\t%s", c);
    fprintf( Stream: code, Format: "\n");
    if (highEmitLoc < emitLoc) highEmitLoc = emitLoc;
} /* emitRO */

/* Procedure emitRM emits a register-to-memory
 * TM instruction
 * op = the opcode
 * r = target register
 * d = the offset
 * s = the base register
 * c = a comment to be printed if TraceCode is TRUE
 */
void emitRM(char *op, int r, int d, int s, char *c) {
    fprintf( Stream: code, Format: "%3d: %5s %d,%d(%d) ", emitLoc++, op, r, d, s);
    if (TraceCode) fprintf( Stream: code, Format: "\t%s", c);
    fprintf( Stream: code, Format: "\n");
    if (highEmitLoc < emitLoc) highEmitLoc = emitLoc;
} /* emitRM */
```

emitRO 和 emitRM 作用分别是对应 RO 和 RM 指令的代码打印函数。除了指令串和

3 个操作数之外，每个函数还带有 1 个附加串参数，它被加到指令中作为注释。

```
/* Function emitSkip skips "howMany" code
 * locations for later backpatch. It also
 * returns the current code position
 */
int emitSkip(int howMany) {
    int i = emitLoc;
    emitLoc += howMany;
    if (highEmitLoc < emitLoc) highEmitLoc = emitLoc;
    return i;
} /* emitSkip */
```

该函数的作用是跳过指定数量的指令。在一些跳转指令时会用到，因为跳转指令会在之后回填。

```
/* Procedure emitBackup backs up to
 * loc = a previously skipped location
 */
void emitBackup(int loc) {
    if (loc > highEmitLoc) emitComment(c: "BUG in emitBackup");
    emitLoc = loc;
} /* emitBackup */
```

该函数的作用是将某个特定的指令位置回填到之前的空处。

```
/* Procedure emitRestore restores the current
 * code position to the highest previously
 * unemitted position
 */
void emitRestore(void) { emitLoc = highEmitLoc; }
```

该函数用于将当前指令位置回填到最高最早的空白处。

```
/* Procedure emitRM_Abs converts an absolute reference
 * to a pc-relative reference when emitting a
 * register-to-memory TM instruction
 * op = the opcode
 * r = target register
 * a = the absolute location in memory
 * c = a comment to be printed if TraceCode is TRUE
 */
void emitRM_Abs(char *op, int r, int a, char *c) {
    fprintf(Stream: code, Format: "%3d: %5s %d,%d(%d) ",
            emitLoc, op, r, a - (emitLoc + 1), pc);
    ++emitLoc;
    if (TraceCode) fprintf(Stream: code, Format: "\t%s", c);
    fprintf(Stream: code, Format: "\n");
    if (highEmitLoc < emitLoc) highEmitLoc = emitLoc;
} /* emitRM_Abs */
```

该函数的作用是当出现 RM 指令时，将绝对地址转换为相对地址。
接下来分析 CGEN.C 和 CGEN.H

```
/* 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);
```

该函数的作用是遍历语法树，在语法树上调用 cGen 生成代码并且最终产生一个 HALT 指令，输出到文件中。

```
/* Procedure cGen recursively generates code by
 * tree traversal
 */
static void cGen(TreeNode *tree) {
    if (tree != NULL) {
        switch (tree->nodekind) {
            case StmtK:
                genStmt(tree);
                break;
            case ExpK:
                genExp(tree);
                break;
            default:
                break;
        }
        cGen(tree->sibling);
    }
}
```

该函数的作用是遍历语法树，判断当前节点是句子还是表达式，并且根据节点类型调用不同的函数，最后递归调用自身，从而实现遍历。

```
/* Procedure genStmt generates code at a statement node */
static void genStmt(TreeNode *tree) {
    TreeNode *p1, *p2, *p3;
    int savedLoc1, savedLoc2, currentLoc;
    int loc;
    switch (tree->kind.stmt) {
        case IfK:
            if (TraceCode) emitComment(c: "<- if");
            p1 = tree->child[0];
            p2 = tree->child[1];
            p3 = tree->child[2];
            /* generate code for test expression */
            cGen(tree->p1);
            savedLoc1 = emitSkip(howMany: 1);
            emitComment(c: "if: jump to else belongs here");

/* Procedure genExp generates code at an expression node */
static void genExp(TreeNode *tree) {
    int loc;
    TreeNode *p1, *p2;
    switch (tree->kind.exp) {
        case ConstK:
            if (TraceCode) emitComment(c: "<- Const");
            /* gen code to load integer constant using LDC */
            emitRM(op: LDC, r: ac, d: tree->attr.val, s: 0, c: "load const");
            if (TraceCode) emitComment(c: "<- Const");
            break; /* ConstK */

        case IdK:
            if (TraceCode) emitComment(c: "<- Id");
            loc = st_lookup(name: tree->attr.name);
            emitRM(op: LDR, r: ac, d: loc, s: gp, c: "load id value");
            if (TraceCode) emitComment(c: "<- Id");
            break; /* IdK */

        case OpK:
            if (TraceCode) emitComment(c: "<- Op");
            /* gen code to load integer constant using LDC */
            emitRM(op: LDC, r: ac, d: tree->attr.val, s: 0, c: "load const");
            if (TraceCode) emitComment(c: "<- Const");
            break; /* ConstK */
    }
}
```

上图为 genStmt 函数和 genExp 函数，作用分别是对于句子节点，创建代码，或者是表达式节点，创建其代码。

至此，中间代码生成的文件分析结束。

四、测试代码生成器

```
1 { Sample program
2   in TINY language -
3   computes factorial
4 }
5 read x; { input an integer }
6 if 0 < x then { don't compute if x <= 0 }
7   fact := 1;
8   repeat
9     fact := fact * x;
10    x := x - 1
11  until x = 0;
12  write fact { output factorial of x }
13 end
```

上图为 SAMPLE.TNY 的代码

其生成的中间代码及注释如下图所示

```
1  * TINY Compilation to TM Code
2  * File: sample.tm
3  * Standard prelude:
4  0: LD 6,0(0) load maxaddress from location 0 * mp清0
5  1: ST 0,0(0) clear location 0
6  * End of standard prelude.
7  2: IN 0,0,0 read integer value * 读入 放入reg[0] (read x;)
8  3: ST 0,0(5) read: store value * 将 reg[0] 存入 dMem[0 + reg[5]] gp
9
10 * -> if
11 * -> Op
12 4: LDC 0,0(0) load const * 加载常量0 reg[0] = 0
13 * <- Const
14 5: ST 0,0(6) op: push left * 将 reg[0] 存入 dMem[0 + reg[6]] mp
15 * -> Id
16 6: LD 0,0(5) load id value * 将 dMem[0 + reg[5]] 装入 reg[0]
17 * <- Id
18 7: LD 1,0(6) op: load left * 将 dMem[0 + reg[6]] 装入 reg[1]
19 8: SUB 0,1,0 op < * 做减法 reg[0] = reg[1] - reg[0]
20 9: JLT 0,2(7) br if true * reg[0] == 0 ? reg[7] += 3 ==> 12
21 10: LDC 0,0(0) false case * reg[0] = 0 false情况
22 11: LDA 7,1(7) unconditional jmp * 无条件跳转reg[7] = reg[7] + 2 ==> 13
23 12: LDC 0,1(0) true case * reg[0] = 1 true情况 (if 0 < x then)
24 * <- Op
25 * if: jump to else belongs here
26 * -> assign
27 * -> Const
28 14: LDC 0,1(0) load const * reg[0] = 1
29 * <- Const
30 15: ST 0,1(5) assign: store value * reg[0] 存入 dMem[1 + reg[5]] gp
31 * <- assign * (fact := 1)
32 * -> repeat
33 * repeat: jump after body comes back here
34 * -> assign
35 * -> Op
36 * -> Id
37 16: LD 0,1(5) load id value * reg[0] = dMem[1 + reg[5]] (fact)
38 * <- Id
39 17: ST 0,0(6) op: push left * reg[0] 存入 dMem[0 + reg[6]]
40 * -> Id
41 18: LD 0,0(5) load id value * reg[0] = dMem[0 + reg[5]] (x)
42 * <- Id
43 19: LD 1,0(6) op: load left * reg[1] = dMem[0 + reg[6]]
44 20: MUL 0,1,0 op * * reg[0] = reg[1] * reg[0]
45 * <- Op * (fact := fact * x)
46
47 21: ST 0,1(5) assign: store value * reg[0] 存入 dMem[1 + reg[5]] (fact)
48 * <- assign
49 * -> assign
50 * -> Op
51 * -> Id
52 22: LD 0,0(5) load id value * reg[0] = dMem[0 + reg[5]] (x)
53 * <- Id
54 23: ST 0,0(6) op: push left * reg[0] 存入 dMem[0 + reg[6]] (x)
55 * -> Const
56 24: LDC 0,1(0) load const * reg[0] = 1
57 * <- Const
58 25: LD 1,0(6) op: load left * reg[1] = dMem[0 + reg[6]] (x)
59 26: SUB 0,1,0 op - * reg[0] = reg[1] - reg[0] (x := x - 1)
60 * <- Op
61 27: ST 0,0(5) assign: store value * reg[0] 存入 dMem[0 + reg[5]] (更新x)
62 * <- assign
63 * -> Op
64 * -> Id
65 28: LD 0,0(5) load id value * reg[0] = dMem[0 + reg[5]] (x)
66 * <- Id
67 29: ST 0,0(6) op: push left * reg[0] 存入 dMem[0 + reg[6]]
68 * -> Const
69 30: LDC 0,0(0) load const * reg[0] = 0
70 * <- Const
71 31: LD 1,0(6) op: load left * reg[1] = dMem[0 + reg[6]] (x)
72 32: SUB 0,1,0 op == * reg[0] = reg[1] - reg[0]
73 33: JEQ 0,2(7) br if true * reg[0] == 0 ? reg[7] += 2 ==> 35
74 34: LDC 0,0(0) false case * reg[0] = 0
75 35: LDA 7,1(7) unconditional jmp * reg[7] = reg[7] + 2 ==> 37
76 36: LDC 0,1(0) true case * reg[0] = 1
77 * <- Op
78 37: JEQ 0,-22(7) repeat: jmp back to body * reg[0] == 0 ? reg[7] -- 21 ==> 26
79 * <- repeat
80 * -> Id
81 38: LD 0,1(5) load id value * reg[0] = dMem[1 + reg[5]]
82 * <- Id
83 39: OUT 0,0,0 write ac * write fact
84 * if: jump to end belongs here
85 13: JEQ 0,27(7) if: jmp to else * reg[0] == 0 ? reg[7] += 28 ==> 41
86 40: LDA 7,0(7) jmp to end * reg[7] = reg[7] + 1 ==> 41
87 * <- if
88 * End of execution.
89 41: HALT 0,0,0 * HALT
```

下图是用 TINY 语言编一个程序计算任意两个正整数的最大公约数与最大公倍数

```
1  read x;
2  read y;
3  S:=x*y;
4
5  if x < y then
6    cur := x;
7    x := y;
8    y := cur
9  end;
10
11 repeat
12   cur := x - (x / y)*y;
13   x := y;
14   y := cur
15 until y = 0;
16
17 write x;
18 write (S / x)
```

下图是生成的代码以及注释

```
1  * TINY Compilation to TM Code
2  * File: SAMPLE1.tm
3  * Standard prelude:
4  0: LD 6,0(0) load maxaddress from location 0
5  1: ST 0,0(0) clear location 0
6  * End of standard prelude.
7  2: IN 0,0,0 read integer value
8  3: ST 0,0(5) read: store value
9  4: IN 0,0,0 read integer value
10 5: ST 0,1(5) read: store value
11 * -> assign
12 * -> Op
13 * -> Id
14 6: LD 0,0(5) load id value
15 * <- Id
16 7: ST 0,0(6) op: push left
17 * -> Id
18 8: LD 0,1(5) load id value
19 * <- Id
20 9: LD 1,0(6) op: load left
21 10: MUL 0,1,0 op *
22 * <- Op
23 11: ST 0,2(5) assign: store value
24 * <- assign
25 * -> if
26 * -> Op
27 * -> Id
28 12: LD 0,0(5) load id value
29 * <- Id
30 13: ST 0,0(6) op: push left
31 * -> Id
32 14: LD 0,1(5) load id value
33 * <- Id
34 15: LD 1,0(6) op: load left
35 16: SUB 0,1,0 op <
36 17: JLT 0,2(7) br if true
37 18: LDC 0,0(0) false case
38 19: LDA 7,1(7) unconditional jmp
39 20: LDC 0,1(0) true case
40 * <- Op
41 * if: jump to else belongs here
42 * -> assign
43 * -> Id
44 22: LD 0,0(5) load id value
45 * <- Id
46 23: ST 0,3(5) assign: store value
47 * <- assign
48 * -> assign
49 * -> Id
50 24: LD 0,1(5) load id value
51 * <- Id
52 25: ST 0,0(5) assign: store value
53 * <- assign
54 * -> assign
55 * -> Id
56 26: LD 0,3(5) load id value
57 * <- Id
58 27: ST 0,1(5) assign: store value
59 * <- assign
60 * if: jump to end belongs here
61 21: JEQ 0,7(7) if: jmp to else
62 28: LDA 7,0(7) jmp to end
63 * <- if
64 * -> repeat
65 * repeat: jump after body comes back here
66 * -> assign
67 * -> Op
68 * -> Id
69 29: LD 0,0(5) load id value
70 * <- Id
71 30: ST 0,0(6) op: push left
72 * -> Op
73 * -> Op
74 * -> Id
75 31: LD 0,0(5) load id value
76 * <- Id
77 32: ST 0,-1(6) op: push left
78 * -> Id
79 33: LD 0,1(5) load id value
80 * <- Id
81 34: LD 1,-1(6) op: load left
82 35: DIV 0,1,0 op /
83 * <- Op
84 36: ST 0,-1(6) op: push left
85 * -> Id
86 37: LD 0,1(5) load id value
87 * <- Id
88 38: LD 1,-1(6) op: load left
89 39: MUL 0,1,0 op *
90 * <- Op
91 40: LD 1,0(6) op: load left
92 41: SUB 0,1,0 op -
93 * <- Op
94 42: ST 0,3(5) assign: store value
95 * <- assign
96 * -> assign
97 * -> Id
98 43: LD 0,1(5) load id value
99 * <- Id
100 44: ST 0,0(5) assign: store value
101 * <- assign
102 * -> assign
103 * -> Id
104 45: LD 0,3(5) load id value
105 * <- Id
106 46: ST 0,1(5) assign: store value
107 * <- assign
108 * -> Op
109 * -> Id
110 47: LD 0,1(5) load id value
111 * <- Id
112 48: ST 0,0(6) op: push left
113 * -> Const
114 49: LDC 0,0(0) load const
115 * <- Const
116 50: LD 1,0(6) op: load left
117 51: SUB 0,1,0 op ==
118 52: JEQ 0,2(7) br if true
119 53: LDC 0,0(0) false case
120 54: LDA 7,1(7) unconditional jmp
121 55: LDC 0,1(0) true case
122 * <- Op
123 56: JEQ 0,-28(7) repeat: jmp back to body
124 * <- repeat
125 * -> Id
126 57: LD 0,0(5) load id value
127 * <- Id
128 58: OUT 0,0,0 write ac
129 * -> Op
130 * -> Id
131 59: LD 0,2(5) load id value
132 * <- Id
133 60: ST 0,0(6) op: push left
134 * -> Id
135 61: LD 0,0(5) load id value
136 * <- Id
137 62: LD 1,0(6) op: load left
138 63: DIV 0,1,0 op /
139 * <- Op
140 64: OUT 0,0,0 write ac
141 * End of execution.
142 65: HALT 0,0,0
143
```


实现一门语言的代码生成器

一、语言确定

选择 C-语言

二、编写选定语言的代码生成器

为了便于调试，这次实验将所有代码都合成为一个文件

对于原 CODE.C 和 CODE.H 文件，其代码几乎未被修改，直接沿用即可，因为和 TINY 语言的完全相同，为了避免冗余，这里就不再次进行代码展示了。

```
//中间代码生成器
void codeGen(TreeNode *syntaxTree, char *codefile) {
    char *s = new char(strlen( Str codefile) + 7);
    strcpy( Destination: s, Source: "File: ");
    strcat( Destination: s, Source: codefile);
    emitComment( c: (char *) "C- Compilation to TM Code");
    emitComment( c: s);
    /* generate standard prelude */
    emitComment( c: (char *) "Standard prelude:");
    emitRM( op: (char *) "LD", r: mp, d: 0, s: ac, c: (char *) "load maxaddress from location 0");
    emitRM( op: (char *) "ST", r: ac, d: 0, s: ac, c: (char *) "clear location 0");
    emitComment( c: (char *) "End of standard prelude.");
    /* generate code for TINY program */
    cGen( tree: syntaxTree);
    /* finish */
    emitComment( c: (char *) "End of execution.");
    emitRO( op: (char *) "HALT", r: 0, s: 0, t: 0, c: (char *) "");
}
```

该函数作用是遍历语法分析树，生成中间代码，并在最后添加 HALT 指令。是中间代码生成器的入口函数。

```
//分开处理表达式和陈述语句的三地址代码：
static void cGen(TreeNode *tree) {
    if (tree != NULL) {
        switch (tree->nodekind) {
            case StmtK:
                genStmt(tree);
                break;
            case ExpK:
                genExp(tree);
                break;
            case TypeK:
                genDecl( tree: tree->child[0]);
                break;
            case DeclareK:
                genDecl(tree);
                break;
            default:
                break;
        }
        cGen( tree: tree->sibling);
    }
}
```

将语句分为四种，分别是 StmtK、ExpK、TypeK、DeclareK，对其调用不同的中间代码生成程序，最后再调用自身，从而实现遍历语法树的目的。

对于 StmtK，要调用 genStmt 函数进行中间代码生成。

其代码如下：

```

/* Procedure genStmt generates code at a statement node */
static void genStmt(TreeNode *tree) {
    TreeNode *p1, *p2, *p3;
    int savedLoc1, savedLoc2, currentLoc;
    int loc;
    int h;
    switch (tree->kind.stmt) {
//IfK, WhileK, ReturnK, FunctionK, CompoundK, AssignK

```

这是一个很重要的函数，将在该函数中，将陈述语句分为 5 种来处理。
首先是 Compound 类型，对于该种类型，由于之前并没有使用，所以，直接进行输出即可。

```

case CompoundK:
    cout << "****CompoundK****" << endl;
    break;

```

ReturnK 类型，直接遍历返回语句的子节点即可。

```

case ReturnK:
    emitComment( c: (char *) "-> return");
    cGen( tree: tree->child[0]);
    emitComment( c: (char *) "<- return");
    break;

```

FunctionK 类型，首先要记录当前的代码(三地址码)的位置，然后在再 hash 表中查找该函数的位置，跳转到该位置。

```

case FunctionK:
    emitComment( c: (char *) "-> function use");
    currentLoc = emitSkip( howMany: 0);
    h = hash(tree->attr.name);
    savedLoc1 = hashTable[h]->codevalue;
    emitComment( c: (char *) "save currentLoc");
    emitRM_Abs( op: (char *) "JMP", r: currentLoc + 1, a: savedLoc1, c: (char *) "fun: jmp to function");
    emitComment( c: (char *) "<- function use");
    break;

```

IfK 类型，首先要递归处理 if 语句后的判读条件，处理完毕后，保存该三地址码的地址，并且留出一个空位，该空位将用来跳转，之后会对该空位进行回填。然后处理 if 后的语句，再保存其代码的位置，同样，也要留出位置来跳转到 else 之后。

然后回到 if 条件之后的空位上，进行回填，跳转 else 处。接下来使用之前提到的 emitRestore 函数，恢复当前的三地址码地址，进行 else 里的代码生成处理。

else 的代码生成结束后，要对之前第二个空着的跳转处，即 if 体结束后的地方进行回填，回填当前三地址码的地址。

最后，通过 emitRestore 函数重新回到此位置。

```
case IfK :
    emitComment( c: (char *) "-> if");
    p1 = tree->child[0];
    p2 = tree->child[1];
    p3 = tree->child[2];
    cGen( tree: p1);
    savedLoc1 = emitSkip( howMany: 1);
    emitComment( c: (char *) "if: jump to else belongs here");
    cGen( tree: p2);
    savedLoc2 = emitSkip( howMany: 1);
    emitComment( c: (char *) "if: jump to end belongs here");
    currentLoc = emitSkip( howMany: 0);
    emitBackup( loc: savedLoc1);
    emitRM_Abs( op: (char *) "JEQ", r: ac, a: currentLoc, c: (char *) "if: jmp to else");
    emitRestore();
    cGen( tree: p3);
    currentLoc = emitSkip( howMany: 0);
    emitBackup( loc: savedLoc2);
    emitRM_Abs( op: (char *) "LDA", r: pc, a: currentLoc, c: (char *) "jmp to end");
    emitRestore();
    emitComment( c: (char *) "<- if");
    break; /* if_k */
```

WhileK 类型，对于该种类型，和 TINY 的 RepeatK 差别较大，参考价值较小。但是，却很类似于刚才所分析符 ifK 情况。只不过体内的语句创建完毕之后，跳转位置不是 else，而是跳转到最初的 if 判断处即可。

```
case WhileK:
    emitComment( c: (char *) "-> while");
    p1 = tree->child[1];
    p2 = tree->child[0];
    savedLoc1 = emitSkip( howMany: 0);
    cGen( tree: p1);
    savedLoc2 = emitSkip( howMany: 1);
    emitComment( c: (char *) "while: jump out belongs here");
    cGen( tree: p2);
    emitRM_Abs( op: (char *) "JMP", r: ac, a: savedLoc1, c: (char *) "while: jmp back to test");
    currentLoc = emitSkip( howMany: 0); //循环结束的位置
    emitComment( c: (char *) "while: jump after body comes back here");
    emitBackup( loc: savedLoc2);
    emitRM_Abs( op: (char *) "JEQ", r: ac, a: currentLoc, c: (char *) "while: jmp out");
    emitRestore();
    emitComment( c: (char *) "<- while");
    break; /* repeat */
```

AssignK 类型，赋值语句就比较简单了，直接处理等号右侧的表达式，然后对等号左边的 ID 在哈希表中的位置进行赋值即可。

```
case AssignK:
    emitComment( c: (char *) "-> assign");
    cGen( tree: tree->child[1]);
    loc = st_lookup( name: tree->child[0]->attr.name);
    emitRM( op: (char *) "ST", r: ac, d: loc, s: gp, c: (char *) "assign: store value");
    emitComment( c: (char *) "<- assign");
    break; /* assign_k */
```

接下来是 genExp，和 Tiny 类似。

```
/* Procedure genExp generates code at an expression node */
static void genExp(TreeNode *tree) {
    int loc;
    TreeNode *p1, *p2;
    switch (tree->kind.exp) {
        case ConstK :
            /* ... */
```

表达式包含三种单元，分别是 ID、常量、操作符。

若是 ID，则直接从 hash 表中取值装载即可。

```
case IdK :
    emitComment( c: (char *) "-> Id");
    loc = st_lookup( name: tree->attr.name);
    emitRM( op: (char *) "LD", r: ac, d: loc, s: gp, c: (char *) "load id value");
    emitComment( c: (char *) "<- Id");
    break; /* IdK */
```

如果是常量，则直接从树节点就可以获取该值，直接存入 ac 寄存器中即可。

```
case ConstK :
    emitComment( c: (char *) "-> Const");
    /* gen code to load integer constant using LDC */
    emitRM( op: (char *) "LDC", r: ac, d: tree->attr.val, s: 0, c: (char *) "load const");
    emitComment( c: (char *) "<- Const");
    break; /* ConstK */
```

如果当前节点是操作符，则递归左子节点和节点直到递归到 ID 或者是常量，然后将左子节点存到 ac 寄存器，将右子节点存入 ac1 寄存器，然后返回上层，将 ac 和 ac1 的值通过该节点的操作符连接起来，并进行计算，将结果存入 ac 中。

```
case OpK :
    emitComment( c: (char *) "-> Op");
    p1 = tree->child[0];
    p2 = tree->child[1];
    /* gen code for ac = left arg */
    cGen( tree: p1);
    /* gen code to push left operand操作数 */
    emitRM( op: (char *) "ST", r: ac, d: tmpOffset--, s: mp, c: (char *) "op: push left");
    /* gen code for ac = right operand操作数 */
    cGen( tree: p2);
    /* now load left operand */
    emitRM( op: (char *) "LD", r: ac1, d: ++tmpOffset, s: mp, c: (char *) "op: load left");

    switch (tree->attr.op) {
        case PLUS/*加*/ :
            emitR0( op: (char *) "ADD", r: ac, s: ac1, t: ac, c: (char *) "op +");
            break;
        case MINUS/*减*/ :
```

要分别处理每一种符号

```
switch (tree->attr.op) {
    case PLUS/*加*/ :
        emitR0( op: (char *) "ADD", r: ac, s: ac1, t: ac, c: (char *) "op +");
        break;
    case MINUS/*减*/ :
        emitR0( op: (char *) "SUB", r: ac, s: ac1, t: ac, c: (char *) "op -");
        break;
    case TIMES/*乘*/ :
        emitR0( op: (char *) "MUL", r: ac, s: ac1, t: ac, c: (char *) "op *");
        break;
    case OVER/*除*/ :
        emitR0( op: (char *) "DIV", r: ac, s: ac1, t: ac, c: (char *) "op /");
        break;
    case LE/*<=*/:
        emitR0( op: (char *) "SUB", r: ac, s: ac1, t: ac, c: (char *) "op <=");
        emitRM( op: (char *) "JLE", r: ac, d: 2, s: pc, c: (char *) "br if true");
        emitRM( op: (char *) "LDC", r: ac, d: 0, s: ac, c: (char *) "false case");
        emitRM( op: (char *) "LDA", r: pc, d: 1, s: pc, c: (char *) "unconditional jmp");
        emitRM( op: (char *) "LDC", r: ac, d: 1, s: ac, c: (char *) "true case");
        break;
    case LT/*<*/:
        emitR0( op: (char *) "SUB", r: ac, s: ac1, t: ac, c: (char *) "op <");
        emitRM( op: (char *) "JLT", r: ac, d: 2, s: pc, c: (char *) "br if true");
        emitRM( op: (char *) "LDC", r: ac, d: 0, s: ac, c: (char *) "false case");
        emitRM( op: (char *) "LDA", r: pc, d: 1, s: pc, c: (char *) "unconditional jmp");
        emitRM( op: (char *) "LDC", r: ac, d: 1, s: ac, c: (char *) "true case");
        break;
    case LT/*<*/:
        emitR0( op: (char *) "SUB", r: ac, s: ac1, t: ac, c: (char *) "op <");
        emitRM( op: (char *) "JLT", r: ac, d: 2, s: pc, c: (char *) "br if true");
        emitRM( op: (char *) "LDC", r: ac, d: 0, s: ac, c: (char *) "false case");
        emitRM( op: (char *) "LDA", r: pc, d: 1, s: pc, c: (char *) "unconditional jmp");
        emitRM( op: (char *) "LDC", r: ac, d: 1, s: ac, c: (char *) "true case");
        break;
}
```

```

emitRM( op: (char *) "LDC", r: ac, d: 1, s: ac, c: (char *) "true case");
break;
case GT/*>*/:
emitRO( op: (char *) "SUB", r: ac, s: ac1, t: ac, c: (char *) "op >");
emitRM( op: (char *) "JGT", r: ac, d: 2, s: pc, c: (char *) "br if true");
emitRM( op: (char *) "LDC", r: ac, d: 0, s: ac, c: (char *) "false case");
emitRM( op: (char *) "LDA", r: pc, d: 1, s: pc, c: (char *) "unconditional jmp");
emitRM( op: (char *) "LDC", r: ac, d: 1, s: ac, c: (char *) "true case");
break;
case NE/*!=*/:
emitRO( op: (char *) "SUB", r: ac, s: ac1, t: ac, c: (char *) "op !=");
emitRM( op: (char *) "JNE", r: ac, d: 2, s: pc, c: (char *) "br if true");
emitRM( op: (char *) "LDC", r: ac, d: 0, s: ac, c: (char *) "false case");
emitRM( op: (char *) "LDA", r: pc, d: 1, s: pc, c: (char *) "unconditional jmp");
emitRM( op: (char *) "LDC", r: ac, d: 1, s: ac, c: (char *) "true case");
break;
case IFEQ /*==*/:
emitRO( op: (char *) "SUB", r: ac, s: ac1, t: ac, c: (char *) "op ==");
emitRM( op: (char *) "JEQ", r: ac, d: 2, s: pc, c: (char *) "br if true");
emitRM( op: (char *) "LDC", r: ac, d: 0, s: ac, c: (char *) "false case");
emitRM( op: (char *) "LDA", r: pc, d: 1, s: pc, c: (char *) "unconditional jmp");
emitRM( op: (char *) "LDC", r: ac, d: 1, s: ac, c: (char *) "true case");
break;
default:
emitComment( c: (char *) "BUG: Unknown operator");

```

加减乘除则直接处理运算器的两个寄存器即 ac 和 ac1 即可。如果是比较符号，则仅需要更改操作名即可，因为对于比较来说，其汇编代码的原理是将两个寄存器中的值相减，然后和 0 比较即可。

对于 TypeK，其实并不需要处理，因为 type 后必定是定义，直接处理定义即可。

```

case TypeK:
    genDecl( tree: tree->child[0]);
    break;

```

接下来是 DeclareK 类型，使用 genDecl 函数来处理

```

static void genDecl(TreeNode *tree) {
    int h;
    switch (tree->kind.declare) {
        case FuncK:
            emitComment( c: (char *) "-> function declare");
            h = hash(tree->attr.name);
            hashTable[h]->codevalue = emitSkip( howMany: 0);
            cGen( tree: tree->child[1]);
            emitComment( c: (char *) "pop");
            emitComment( c: (char *) "<- function declare");
            break;
        default:
            break;
    }
}

```

只需要记录一下地址，然后将该函数的地址存入 hash 表中即可，注意，最后要有 pop 弹出语句。

以上就是所有的代码。接下来进行测试。

三、 准备 2~3 个测试用例，测试你的程序，并逐行解释生成的目标代码

测试 1:

测试代码如下:

```
1  int main(){
2      int x;
3      x=0;
4      return 0;
5  }
```

结果如下:

```
1  * C- Compilation to TM Code
2  * File: D:\大学\大三下\编译技术\实验四\lab4\Cminus\test3.txt
3  * Standard prelude:
4  | 0:    LD  6,0(0)   load maxaddress from location 0
5  | 1:    ST  0,0(0)   clear location 0
6  * End of standard prelude.
7  * -> function declare
8  * -> assign
9  * -> Const
10 | 2:    LDC  0,0(0)   load const
11 * <- Const
12 | 3:    ST  0,1(5)   assign: store value
13 * <- assign
14 * -> return
15 * -> Const
16 | 4:    LDC  0,0(0)   load const
17 * <- Const
18 * <- return
19 * pop
20 * <- function declare
21 * End of execution.
22 | 5:    HALT 0,0,0
```

测试 2:

测试代码如下:

```
/* A program to perform selection sort on a 10
element array. */
int x[10];
void main ()
{ int i;
  i = 0;
  while (i < 10)
  {
    i = i + 1;
    while (i < 10)
    {
      i = i + 1;
    }
  }
}
```

结果如下:

```
1  * C- Compilation to TM Code
2  * File: D:\大学\大三下\编译技术\实验四\lab4\Cminus\test.txt
3  * Standard prelude:
4  0: LD 6,0(0) load maxaddress from location 0
5  1: ST 0,0(0) clear location 0
6  * End of standard prelude.
7  * -> function declare
8  * -> assign
9  * -> Const
10 2: LDC 0,0(0) load const
11 * <- Const
12 3: ST 0,2(5) assign: store value
13 * <- assign
14 * -> while
15 * -> Op
16 * -> Id
17 4: LD 0,2(5) load id value
18 * <- Id
19 5: ST 0,0(6) op: push left
20 * -> Const
21 6: LDC 0,10(0) load const
22 * <- Const
23 7: LD 1,0(6) op: load left
24 8: SUB 0,1,0 op <
25 9: JLT 0,2(7) br if true
26 10: LDC 0,0(0) false case
27 11: LDA 7,1(7) unconditional jmp
28 12: LDC 0,1(0) true case
29 * <- Op
30 * while: jump out belongs here
31 * -> assign
32 * -> Op
33 * -> Id
34 14: LD 0,2(5) load id value
35 * <- Id
36 15: ST 0,0(6) op: push left
37 * -> Const
38 16: LDC 0,1(0) load const
39 * <- Const
40 17: LD 1,0(6) op: load left
41 18: ADD 0,1,0 op +
42 * <- Op
43 19: ST 0,2(5) assign: store value
44 * <- assign
```

```
45 * -> while
46 * -> Op
47 * -> Id
48 20: LD 0,2(5) load id value
49 * <- Id
50 21: ST 0,0(6) op: push left
51 * -> Const
52 22: LDC 0,10(0) load const
53 * <- Const
54 23: LD 1,0(6) op: load left
55 24: SUB 0,1,0 op <
56 25: JLT 0,2(7) br if true
57 26: LDC 0,0(0) false case
58 27: LDA 7,1(7) unconditional jmp
59 28: LDC 0,1(0) true case
60 * <- Op
61 * while: jump out belongs here
62 * -> assign
63 * -> Op
64 * -> Id
65 30: LD 0,2(5) load id value
66 * <- Id
67 31: ST 0,0(6) op: push left
68 * -> Const
69 32: LDC 0,1(0) load const
70 * <- Const
71 33: LD 1,0(6) op: load left
72 34: ADD 0,1,0 op +
73 * <- Op
74 35: ST 0,2(5) assign: store value
75 * <- assign
76 36: JMP 0,-17(7) while: jmp back to test
77 * while: jump after body comes back here
78 29: JEQ 0,7(7) while: jmp out
79 * <- while
80 37: JMP 0,-34(7) while: jmp back to test
81 * while: jump after body comes back here
82 13: JEQ 0,24(7) while: jmp out
83 * <- while
84 * pop
85 * <- function declare
86 * End of execution.
87 38: HALT 0,0,0
88
```

测试 3:

测试代码如下:

```
1  int x;
2  int a;
3  int b;
4  int max(){
5      if(a>=b){
6          return a;
7      }
8      else{
9          return b;
10     }
11 }
12 int main(){
13     int n;
14     int i;
15     int count;
16     n = 10;
17     i = 0;
18     count = 0;
19     while( i < n ){
20         count = count + i;
21         i = i + 1;
22     }
23     return 0;
24 }
```

结果如下:

```
1 * C- Compilation to TM Code
2 * File: D:\大学\大三下\编译技术\实验四\lab4\cminus\test1.txt
3 * Standard prelude:
4 | 0: LD 6,0(0) load maxaddress from location 0
5 | 1: ST 0,0(0) clear location 0
6 * End of standard prelude.
7 * -> function declare
8 * -> if
9 * -> Op
10 * -> Id
11 | 2: LD 0,1(5) load id value
12 * <- Id
13 | 3: ST 0,0(6) op: push left
14 * -> Id
15 | 4: LD 0,2(5) load id value
16 * <- Id
17 | 5: LD 1,0(6) op: load left
18 | 6: SUB 0,1,0 op >=
19 | 7: JGE 0,2(7) br if true
20 | 8: LDC 0,0(0) false case
21 | 9: LDA 7,1(7) unconditional jmp
22 | 10: LDC 0,1(0) true case
23 * <- Op
24 * if: jump to else belongs here
25 * -> return
26 * -> Id
27 | 12: LD 0,1(5) load id value
28 * <- Id
29 * <- return
30 * if: jump to end belongs here
31 | 11: JEQ 0,2(7) if: jmp to else
32 * -> return
33 * -> Id
34 | 14: LD 0,-1(5) load id value
35 * <- Id
36 * <- return
37 | 13: LDA 7,1(7) jmp to end
38 * <- if
39 * pop
40 * <- function declare
41 * -> function declare
42 * -> assign
43 * -> Const
44 | 15: LDC 0,10(0) load const
45 * <- Const
46 | 16: ST 0,5(5) assign: store value
47 * <- assign
48 * -> assign
49 * -> Const
50 | 17: LDC 0,0(0) load const
51 * <- Const
52 | 18: ST 0,6(5) assign: store value
53 * <- assign
54 * -> assign
55 * -> Const
56 | 19: LDC 0,0(0) load const
57 * <- Const
58 | 20: ST 0,7(5) assign: store value
59 * <- assign
60 * -> while
61 * -> Op
62 * -> Id
63 | 21: LD 0,-1(5) load id value
64 * <- Id
65 | 22: ST 0,0(6) op: push left
66 * -> Id
67 | 23: LD 0,5(5) load id value
68 * <- Id
69 | 24: LD 1,0(6) op: load left
70 | 25: SUB 0,1,0 op <
71 | 26: JLT 0,2(7) br if true
72 | 27: LDC 0,0(0) false case
73 | 28: LDA 7,1(7) unconditional jmp
74 | 29: LDC 0,1(0) true case
75 * <- Op
76 * while: jump out belongs here
77 * -> assign
78 * -> Op
79 * -> Id
80 | 31: LD 0,7(5) load id value
81 * <- Id
82 | 32: ST 0,0(6) op: push left
83 * -> Id
84 | 33: LD 0,6(5) load id value
85 * <- Id
86 | 34: LD 1,0(6) op: load left
87 | 35: ADD 0,1,0 op +
88 * <- Op
89 | 36: ST 0,7(5) assign: store value
90 * <- assign
91 * -> return
92 * -> Const
93 | 37: LDC 0,0(0) load const
94 * <- Const
95 * <- return
96 | 38: JMP 0,-18(7) while: jmp back to test
97 * while: jump after body comes back here
98 | 30: JEQ 0,8(7) while: jmp out
99 * <- while
100 * pop
101 * <- function declare
102 * End of execution.
103 | 39: HALT 0,0,0
104
```

所有结果都正确!

遇到的问题:

在处理符号时,最初把单个“=”作为一种操作符,归类到表达式中,这就造成了很大的问题,生成三地址码时在这里会出现错误,有时甚至会在这里停止。最后,选择将单个“=”设置为 assign 节点,然后在其中进行相应的处理即可。这样就解决了该问题。

在进行回填时,很容易就会出现问题,如果没有思考清楚,则将会导致不可预知的错误,因此,尤其是 if-else 结构、while 结构时,必须多次思考,思考清晰后再去编写代码,即可提高质量与效率。

收获与体会:

通过这次实验,收获很多。首先,这次实验的难度很大,刚开始时没有任何想法,不知道怎么去完成它。后来通过对 tiny 编译器的深入学习再结合上课时的 ppt,学到了如何进行中间代码生成。然而,在经过学习后再开始时,还是遇到了很多困难,毕竟理论和实践还是有差别的。在一筹莫展时,再次阅读 tiny 编译器的代码,类比仿照后,

<p>终于写出了这次实验。在开始后，本来认为很难的东西，发现难度并没有那么大，最终，再解决掉很多问题后，终于完成了试验。中间代码生成还是有难度的，尤其是回填部分，何时回填，回填什么，都是非常值得思考的问题，认真将这些问题进行思考并得出答案后，对编译技术的学习是很有帮助的。总之，通过这次实验，掌握了中间代码生成，为今后的学习打下了坚实的基础。</p>	
实验成绩	