基于 C 语言实现的 FMJ 编译器

各阶段语言规则

该编译器将 FMJ 语言翻译为汇编语言共经历了 3 个阶段，即 FMJ -> AST -> IR -> assem 。其中：

- FMJ 语法见 `FMJ1.4.pdf` 文件（删去了 `Exp.length`）
- AST 各模块见 `src/fmjAST.h` 文件
- IR 各模块见 `src/tree.h` 文件
- assem 各模块见 `src/assem.h` 文件

下面将分阶段进行进一步介绍。（代码默认在 `src/` 目录下）

阶段 1：FMJ -> AST

主要基于 lex 和 yacc 来实现，lex 进行词法分析，yacc 进行语法分析

具体代码见 `lexer.l` 和 `parser.yacc`

`prog.c` 中的 `prog` 函数提供了一个端到端的接口，传入文件名，返回 AST。

```
A_prog prog(string fn) {
    freopen(fn, "r", stdin);
    (yyparse());
    return root;
}
```

阶段 2：AST -> IR

`translate.c` 中的 `trans_prog` 函数提供了一个接口，传入 AST ，返回一个 `StmList` ，其中每一项表示一个 method 的 IR。

翻译 `class` 时，扫描所有 class 的成员变量和函数，分别分配一个 offset。为处理继承类的问题，还建立了一个映射用于记录各个 class 的 method 对应哪个被定义的 method。

```
S_table vars2offset;     // plused 1
S_table methods2offset;  // plused 1
S_table classMethods2label;
```

由于 `S_table` 会将 0 视为 NULL，因此在存储 offset 时加上 1 以示区别。

规范化

转换出的 IR 将通过 `main.c` 中的 `canon_method` 函数进行规范化，具体来说，一个 method 的 IR tree 会先被展开成 list，分成从 label 到 jump 的一个个 block，再重新进行整合。

阶段 3：IR -> assem

`tile.c` 中的 `tileSl` 函数提供了一个接口，传入 IR，返回 assem。

使用的 `tile` 见 `ARM tiles for FMJ` 中标注的部分。

寄存器分配

得到的 assem 每条指令都表明了其使用、声明的寄存器信息和跳转信息，该信息将用于流图分析，以计算每条指令中可能活跃的寄存器，计算出 interference graph，从而将寄存器分配归约为图染色问题。

寄存器 `r0-r10` 和 `lr` 是可分配的。在本项目中，首先用染色法分配 `r0-r8` 和 `lr`，然后将 `r9-r10` 用于改写 spill 出的寄存器。见 `color.c` 中的 `COL_color` 函数，节选如下：

```
    while (cnt) {
        // printf("round:\n");
        bool flag = FALSE;
        for (G_nodeList l = ig; l; l = l->tail) {
            G_node gn = l->head;
            Temp_temp n = G_nodeInfo(gn);
            string tn = Temp_look(Temp_name(), n);
            if (atoi(tn) < 100) continue;
            int d = atoi(Temp_look(degs, n));
            if (d >= 0 && d < 10) {
                // printf("[[%d]]\n", d);
                flag = TRUE;
                Temp_enter(degs, n, i2str(-1));
                sta = G_NodeList(gn, sta);  // simplify
                eraseNode(gn, degs);
                cnt--;
            }
        }
        if (flag == FALSE) {
            for (G_nodeList l = ig; l; l = l->tail) {
                G_node gn = l->head;
                Temp_temp n = G_nodeInfo(gn);
                string tn = Temp_look(Temp_name(), n);
                if (atoi(tn) < 100) continue;
                int d = atoi(Temp_look(degs, n));
                if (d >= 0) {
                    // printf("[%d]\n", d);
                    assert(d >= 10);
                    Temp_enter(degs, n, i2str(-1));
                    Temp_enter(cr.coloring, n, String("Spill"));
                    assert(n);
                    cr.spills = Temp_TempList(n, cr.spills);
                    eraseNode(gn, degs);
                    cnt--;
                    break;
                }
            }
        }
    }
```

> 每当还有寄存器未分配时，寻找度数小于 10 的寄存器存入栈中。若未找到，随机选取一个寄存器进行 Spill

`main.c` 中的 `alloc` 函数提供了寄存器分配的接口，输出的代码即为真正的汇编代码。

编译器编译运行方法

见 `src/` 目录下的 `README.md` 文件。