项目设计目标

受到笔者本学期选择的电子音乐课上提到的 Live Coding：在 Livehouse 等场景现场写函数式语言，用编程来控制声音 这一艺术形式的启发，我将我的项目目标与课上做的报告相比进行了一些改动，变为：

1. ~~在课程 SysY 要求的基础上实现课程中的部分数据流分析，具体包括：可达定值分析，可用表达式分析。~~ 由于音乐生成器已经有大量工作量，故不准备继续实现数据流分析，但已经实现了一个框架，详见 optimize_exp.rs。
2. 实现一个简单的音乐生成器“SysY Live”，具体用处是：根据支持根据以文本形式给定的乐谱以及以 .wav 形式的音频，进行“sing”操作：对音频利用外部库 rubberband 进行调音，利用 sox 库进行拼接，实现“用 a.wav 的声音唱曲谱”的结果。

SysY Live 程序示例如下所示：

```
int main() {
        Score x(syllablename = ":1 :1 :5 :5 | :6 :6 :5 - | :4 :4 :3 :3 | :2 :2 :1 - ");
    Score x1(syllablename = "1 - - -  | 4 - - - | 5$ - 7$ -  | 2 5 1 - ");
    Score x2(syllablename = "{::1 :5} {::1 ::3} {::5 ::1} {::3 ::5} | {::6 ::4} {:::1 ::6} {::5 ::
        int a[10] = {49, 46, 119, 97, 118}; // 1.wav
        int b[10] = {50, 46, 119, 97, 118}; // 2.wav
        int c[10] = {51, 46, 119, 97, 118}; // 3.wav
        int d[10] = {52, 46, 119, 97, 118}; // 4.wav
        x.sing(a, b, 44100, 16, 2);
    x1.sing(a, c, 44100, 16, 2);
    x2.sing(a, d, 44100, 16, 2);
    Track t1(b);
    Track t2(c);
    Track t3(d);
    t1.stack(t2);
    t1.stack(t3);
    return 0;
}
```

其中，1.wav 和 2.wav 的例子都将附于提交的代码中。

项目使用 rust 语言编写。

项目进度

由于数据流分析的意义只是为了试图增加排名分，没有新功能，故暂时还未完成实现。目前 SysY Live 部分已经完成实现。

项目的完整实现已经提交于下述链接。本次提交在教学网上的文件也有完整实现。

```
https://disk.pku.edu.cn/link/AA38EAA2B469C6410AB8B675F9DCB4D3D5
文件名：compiler-master.zip
有效期限：2026-07-19 01:27
```

目前，实现 SysY 本身约用了一星期，设计并实现 SysY Live 用了至少 6 天。

项目实现细节

SysY 本体

词法分析和语法分析

利用 LALRPOP 工具，我们只需给出语法树定义，以及语法树上每个结点对应 rust 中的类型以及从子结点构造这个点的方式，就可以构建出抽象语法树。在构建抽象语法树的过程中，会出现 Dangling Else 问题，可以将 Stmt 分为 Matched Statement 和 Unmatched Statement 分别考虑。

虽然不会出现修改 const 变量的语义错误，但仍然需要区分 VarDef 和 ConstVarDef，这是因为要实现下一部分的常量折叠。

语义分析和 IR 生成

遍历上一步骤的语法树。利用 koopa 库（及 koopa IR 自带类型的性质），可以设计一种遍历一次 AST 就能输出整个 IR 的算法，需要在每个结点上维护以下属性：

1. Function / BasicBlock，表示进入该结点表示的代码段之前，处于哪个函数 / 基本块。（L 属性）
2. Var，符号表：存储所有非 const 变量。（L 属性）
3. WhileContext，用于处理 break 和 continue。（L 属性）
4. Value，表示该代码段传出的唯一会被外部利用的值。（S 属性）

构建和输出 IR 都可以直接调用 koopa 库的接口。

在维护 Value 时可以自动进行常量折叠：如果传入运算的几个 Value 全是常量，则直接返回一个常量。这在数组长度计算时是必要的。

riscv 生成

对于每个函数，遍历 IR，对 IR 的每个变量分配一段栈空间。暂时实现的是每个变量都分配栈空间，只用三个临时寄存器。只需实现“把寄存器拷贝到栈”以及“把栈拷贝到寄存器”两个操作就可以简便实现所有运算的翻译。

数据流分析

预计的实现方法是迭代法。具体地，对每条 IR 语句维护 pre 表示这条语句开始前的信息，对每个基本块维护 tail 表示这个基本块结束时的信息，并且按照下面的方法实现：

```
While（存在信息需要更新）更新信息
```

其中的信息在 Koopa IR 上是指形如 `x = y op z` 或 `store x y` 的定义。由于 Koopa IR 除了 alloc 和 load 都是 SSA 的，所以只需存储所有目前活跃的值，如果遇到重复计算活跃值则可删除变量或删除 store/load 语句。

SysY Live

SysY Live 引用了外部工具 sox, rubberband，这两者无法在 riscv32 架构下运行。因此，笔者采取如下方法实现：

1. 在语义分析时，将所有 SysY Live 定义及函数调用转化为 `int` 和 `int*` 型的函数调用。
2. 如果要编译 SysY Live，则需要利用 koopa 库，将 SysY 代码转为 LLVM IR，再利用 LLVM 工具链编译成 `.o` 文件。
3. 函数用 C++ 在外部库中实现，并编译为 `imp.o`。
4. 链接 `imp.o` 和 SysY 编译得来的 `.o`，得到可执行文件。

`imp.o` 库中只需实现一些函数，维护每个音符的时值和音高即可。在 SysY Live 中会存在一些常量字符串，这些字符串也将被转为 `int` 类型的全局数组，进而传给 `imp.o`。

具体使用方法请见提交的使用方法以及使用方法教学视频。

达成实例效果

已通过 Lv1~Lv9 的在线评测所有测试点，目前无优化的项目在性能测试中需要 600s 左右。编写的 SysY Live 外部库已经经过测试。

SysY 简谱的上下文无关文法

```
score = bar (| bar)*
bar = (note | "-" | "_" | notes)*
notes = "{" (note | "-" | "_")+ "}" // 将音符等分，"-" "_" 表示延音
note = midnote$+ | :*midnote | "0"
midnote = (b | #)[1-7]
```

参考文献

Live Coding 项目 TidalCycles (基于 Haskell)：https://tidalcycles.org/

Rubberband：https://github.com/breakfastquay/rubberband

sox：https://github.com/chirlu/sox