

SysY，一种 C 语言子集的编译器设计文档

1 总体设计

1.1 实现编译器的语言选择

在之前的课内学习中，使用的语言主要为 C 和 Java，但从未使用过 C++。当前主流编译器也多由 C 或 C++实现。因此决定使用现代 C++实现该编译器，以此为契机熟悉使用现代 C++实现复杂工程的方法。

“现代 C++”指的是充分考虑或利用课程组所允许的最高标准 C++11 的新特性，如移动语义（Move Semantics），在需要时正确对类定义用于左值（l-value）的拷贝构造（Copy Constructor）和拷贝赋值（Copy Assignment）特殊成员函数，以及用于右值（r-value）的移动构造（Move Constructor）和移动赋值（Move Assignment）特殊成员函数；以及 RAII（Resource Acquisition Is Initialization），使用智能指针避免手动管理内存可能造成的内存泄漏、错误释放等各种问题等。实现时注重代码的规范和简洁，确保可读性和可维护性，同时与编译器的性能进行权衡，如选用合适的 STL 容器和数据结构、模板、运算符重载、类方法指针、lambda 表达式等。

1.2 基本架构

由于各种原因，选择解释执行中间代码作为本编译器的最后实现方案，由词法分析、语法分析、语义分析、符号表构建、错误处理、中间代码生成、解释执行七大任务，建立 Tokenizer、Parser、Error 和 StackMachine 四个大类，简化类图如图 1 所示。由 Tokenizer 进行词法分析，其结果传递给 Parser 进行语法分析。语义分析与错误处理伴随词法分析和语法分析进行，中间代码生成伴随语义分析进行。最后解释执行中间代码，输出中间代码和程序的执行结果。

此处顺便对课程组的要求细节提一个建议：为了更贴近编译器的真实使用场景，被编译程序的源文件名、输出文件名、以及输出选项（如分别打开、关闭输出语义分析、语法分析、错误处理、中间代码生成各部分的结果）应通过命令行参数形式给出，不应要求在程序内写死。

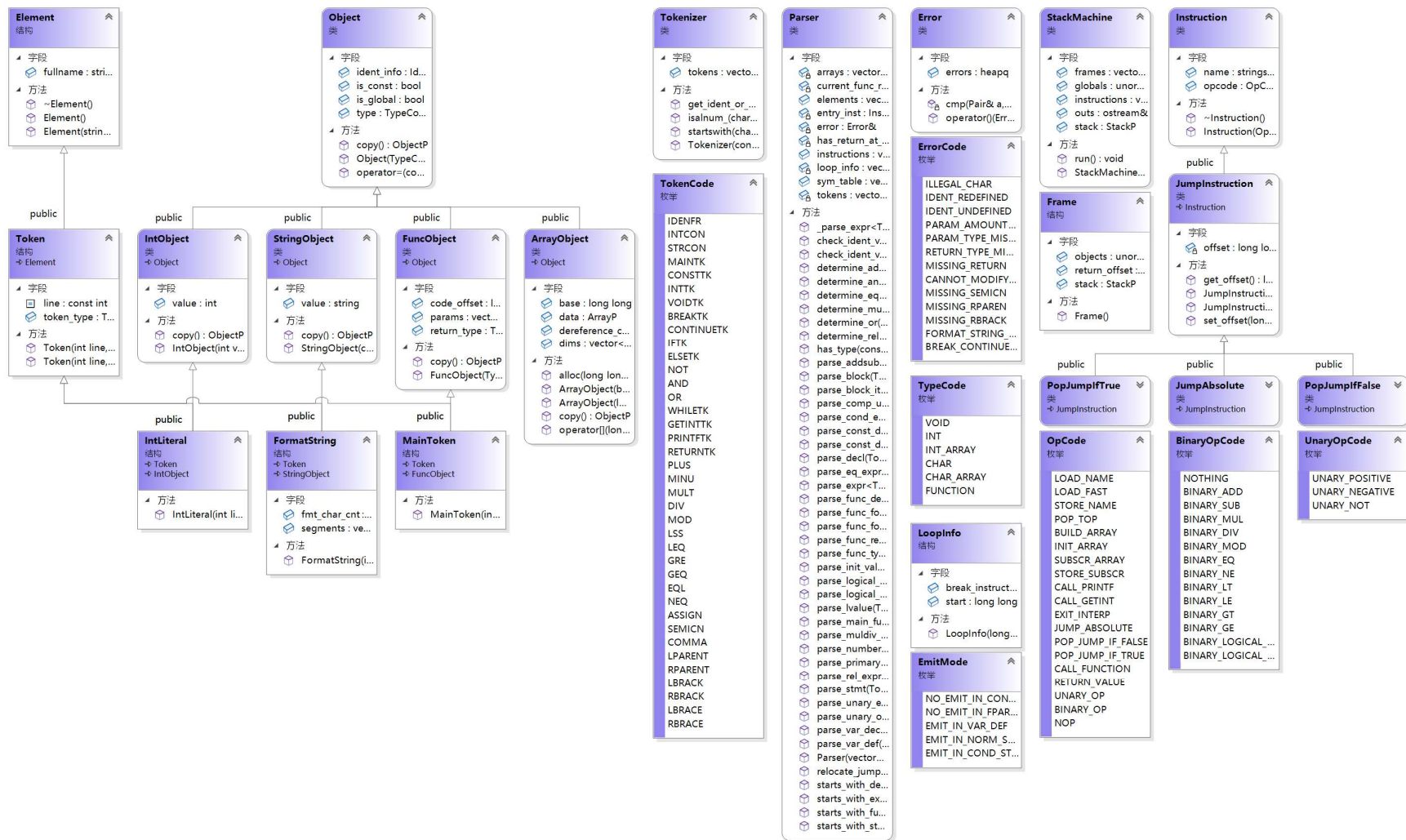


图 1 架构简化类图

2 词法分析部分设计

词法分析部分用于从被编译的源程序文件读入原始字符串,根据词法将其切分为单位符号(对应图 1 中的 Token 类),即“分词”。评测要求输出每个 token 的信息。由于不是所有 token 的信息都是固定的,且 C++没有用于输出枚举(enum)类型名的反射(reflection)机制,因此考虑为每个 token 创建一个类,包含该 token 所在的行号(用于错误处理,还可以存储所在列号)。将每个 token 实例放入 vector 中并储存所需信息,遍历即可按要求输出结果。遇到词法错误时,将该错误加入错误列表中,以待输出。尽管一些错误课程组没有要求,但在实现词法分析时,防止了可能导致越界的情况。

3 语法分析部分设计

语法分析部分使用递归下降法,将词法分析部分输出的 token 按照语法,切分为语法元素(对应图 1 中的 Element 类),即“断句”。评测要求输出每个 token 的信息,加上部分语法元素的信息。为了简便起见,暂不建立语法树,仅在每个递归函数返回前将 element 实例和 token 放入 vector 中,遍历即可按要求输出结果。遇到语法错误时,将该错误加入错误列表中,以待输出。

4 语义分析与错误处理部分设计

由于出于简便没有建立语法树,语义分析伴随语法分析进行。语义分析及其错误处理部分用于分析程序的语义,需要建立符号表,表中存了标识符的实例(对应图 1 中的 Identifier 类)。使用栈式符号表,进入语法块压栈,离开语法块弹栈,全局标识符在栈底。检查类型匹配时,由栈顶向栈底搜索,搜索到后比较类型信息。遇到类型不匹配等语义错误时,将该错误加入错误列表中,以待输出。

由于要求报错顺序按行号递增,输出错误前需要排序,或者使用优先队列,出于学习目的,使用后者。

5 中间代码生成和解释执行部分设计

5.1 基本内容

由于出于简便没有建立语法树,中间代码生成伴随语义分析进行。该部分依赖语义分析的结果和符号表,生成中间代码以供之后解释执行。

首先需要确定中间代码的格式。由于选择将中间代码用于解释执行,而栈式虚拟机的中间代码实现起来较为简洁,经参阅 CPython 和 C4 解释器的中间代码,

发现 CPython 的中间代码可读性、可维护性和可扩展性更好,故选择了与 CPython 类似的中间代码实现,如表 1 所示。为方便调试,生成中间代码后可以输出各指令信息。

表 1 栈式虚拟机指令

Operation	Operand	Execution Explanation
LOAD_NAME	IdentP, 即建立符号表时获取的唯一标识符	若 IdentP 为全局变量, 从全局变量区取出该变量; 若 IdentP 为局部变量, 从当前帧的变量区中取出该变量 (若需优化性能, 可采取与 CPython 一致的方案, 即该指令仅用于全局变量)
LOAD_FAST	ObjectP, 即常量或字面量	直接读取 ObjectP 对象
STORE_NAME	IdentP, 即建立符号表时获取的唯一标识符	若 IdentP 为全局变量, 弹出栈顶对象, 并存入全局变量区; 若 IdentP 为局部变量, 弹出栈顶对象, 并存入当前帧的变量区 (若需优化性能, 可采取与 CPython 一致的方案, 即该指令仅用于全局变量, 另加 STORE_FAST 指令用于局部变量)
POP_TOP	无	将栈顶对象弹出 (用于表达式的值未使用时)
BUILD_ARRAY	无	从栈顶弹出数组大小, 按此大小创建数组对象, 并推入栈中 (该数组大小为展平后的总大小)
INIT_ARRAY	无	创建数组对象, 将栈中的对象依次弹出并写入数组, 再将数组推入栈中
SUBSCR_ARRAY	无	从栈中弹出下标、数组, 对数组取下标, 再推入栈中

Operation	Operand	Execution Explanation
STORE_SUBSCR	无	从栈中弹出取值对象和赋值对象，并将所取的值赋给要赋的对象
CALL_PRINTF	FormatStringP，即格式化字符串对象	从栈中弹出要输出的对象，并按格式输出
CALL_GETINT	无	从输入流中读入整数并构建对象推入栈中
EXIT_INTERP	无	退出解释器
JUMP_ABSOLUTE	IntObjectP，其中存的是跳转的绝对地址	跳转到相应地址
POP_JUMP_IF_FALSE	IntObjectP，其中存的是跳转的绝对地址	弹出栈顶对象，若该对象的值为 <code>false</code> ，跳转到相应地址
POP_JUMP_IF_TRUE	IntObjectP，其中存的是跳转的绝对地址	弹出栈顶对象，若该对象的值为 <code>true</code> ，跳转到相应地址
CALL_FUNCTION	FuncP，即函数对象	创建新帧，将实参依次从栈中弹出，复制到新帧中，在新帧中保存返回地址为下一条指令的地址，并跳转到函数入口地址
RETURN_VALUE	无	从当前帧读取返回地址，若栈为空，弹出当前帧，并跳转到返回地址；若栈不为空，弹出剩余的返回对象，弹出当前帧，将返回对象推入返回后的帧，并跳转到返回地址（此处与 CPython 处理不同，CPython 为了支持动态类型，“没有”返回值的函数会返回 <code>None</code> ）
UNARY_OP	UNARY_POSITIVE	弹出栈顶对象，做相应一元运算，并推入栈中（若需优化性能，可采取与 CPython 一致的方案，即为每个操作符单独分配指令）
	UNARY_NEGATIVE	
	UNARY_NOT	

Operation	Operand	Execution Explanation
BINARY_OP	BINARY_ADD	弹出栈顶对象两次，做相应二元运算，并推入栈中（若需优化性能，可采取与 CPython 一致的方案，即为每个操作符单独分配指令。另外，没有逻辑与和逻辑或的指令，因为需支持短路求值，仅用跳转指令即可实现）
	BINARY_SUB	
	BINARY_MUL	
	BINARY_DIV	
	BINARY_MOD	
	BINARY_EQ	
	BINARY_NE	
	BINARY_LT	
	BINARY_LE	
	BINARY_GT	
	BINARY_GE	

由于函数形参若有高维数组，必须指定低维的维数，而低维的维数可以是编译器能确定的常量表达式，所以统一所有常量在编译器求值，不再生成虚拟机指令，而全局变量依然生成虚拟机指令。（C 语言中全局常量不能用常量数组中的值初始化，因此 SysY 在这点上并不符合 C 语言标准）

参阅 CPython 时发现 JUMP_IF_FALSE_OR_POP 和 JUMP_IF_TRUE_OR_POP 指令，发现是因为 Python 支持 Operator Chaining，这两个指令可以支持含 Operator Chaining 的逻辑表达式，但本编译器无需支持。还发现 CPython 从 3.10 版本起，对于 while 循环采取了 if + do-while 的优化，用以减少长跳转造成指令 cache 缺失的性能折损。简便起见，本编译器只使用了常规 while 的代码生成方式。

解释执行部分使用栈式虚拟机（对应图 1 中的 StackMachine 类），首先模拟操作系统加载全局变量、指令并设置初始栈帧和程序计数器（Program Counter, PC），然后在循环中模拟 CPU 进行取指、译码、执行、访存、写回等步骤，执行指令并输出结果。每一个栈帧中有当前帧的临时变量区、运行栈和返回地址。由于为全局变量生成虚拟机指令，且各函数的指令顺序排布，需要将 main 作为函数处理，并在全局变量的虚拟机指令后生成跳到 main 函数的指令，并紧接着生成一条退出解释器的指令。

5.2 性能

写了一个计算 π 的程序如图 3 所示，部分虚拟机指令及程序运行结果如图 4 所示。（本编译器支持“变长数组”，即数组声明的维数可以是任意返回整数的表达式。）CPython 运行等价程序的结果如所示。可以发现，本编译器的性能还有很

大提升空间。性能不如 CPython 的原因主要是智能指针使用较多，运行 Profile 后发现大部分时间全是内核态时间，真正用户态时间很少，说明开销主要在动态分配内存上。如访问数组，至少要从数组对象 `ArrayObjectP` 指针跳到其内部的数据 `ArrayP` 指针，再从该指针跳到 `vector<ObjectP>` 内部的 `data *` 指针，再从该指针跳到相应位置取出值。之所以有那么多指针，是因为数组对象传参时其内部的数据 `vector<ObjectP>` 需被共享，而 STL 的 `vector` 内的 `data *` 指针不是智能指针，无法处理共享的情况，只有 `swap` 函数用于交换，无法满足需求。若需改进，可以减少智能指针的使用，如手写相关数据结构、在一些地方替换为值类型等。但为了代码的简洁，同时防止考试时因考虑不周引入 bug，目前所有需要共享的对象全部使用智能指针。另外 CPython 由于用 C 实现，对数据的控制权更强大；而本编译器大量使用 C++ 的 STL，对于 STL 内部是如何实现的、C++ 编译器能否将其优化到接近 C 的水平并不清楚。

```

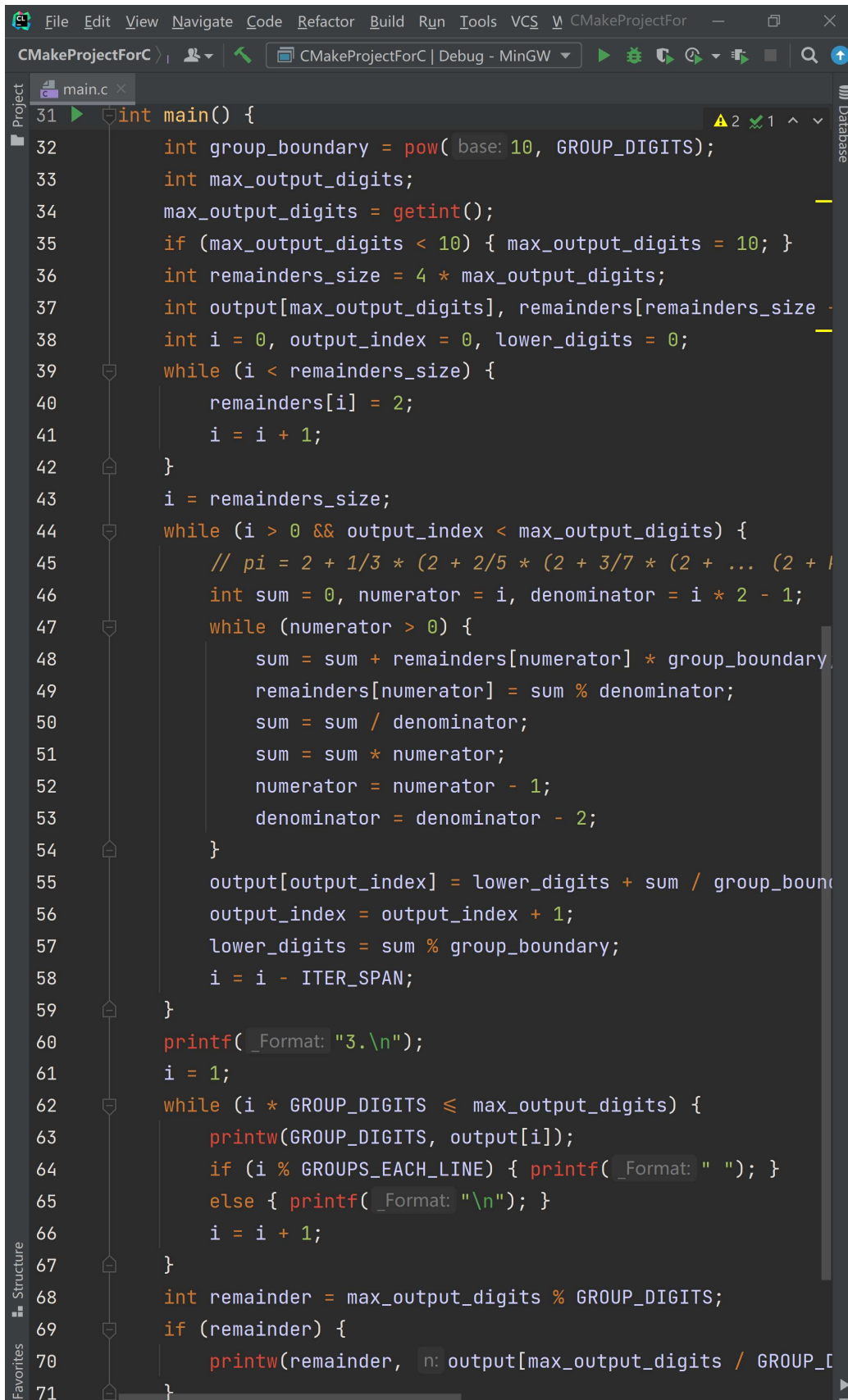
kevin@KevinTSQ: /mnt/c/Users/ × +
$ python3 -m cProfile test5.py
3.
14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679
82148 08651 32823 06647 09384 46095 50582 23172 53594 08128
48111 74502 84102 70193 85211 05559 64462 29489 54930 38196
44288 10975 66593 34461 28475 64823 37867 83165 27120 19091
45648 56692 34603 48610 45432 66482 13393 60726 02491 41273
72458 70066 06315 58817 48815 20920 96282 92540 91715 36436
78925 90360 01133 05305 48820 46652 13841 46951 94151 16094
33057 27036 57595 91953 09218 61173 81932 61179 31051 18548
07446 23799 62749 56735 18857 52724 89122 79381 83011 94912
98336 73362 44065 66430 86021 39494 63952 24737 19070 21798
60943 70277 05392 17176 29317 67523 84674 81846 76694 05132
00056 81271 45263 56082 77857 71342 75778 96091 73637 17872
14684 40901 22495 34301 46549 58537 10507 92279 68925 89235
42019 95611 21290 21960 86403 44181 59813 62977 47713 09960
51870 72113 49999 99837 29780 49951 05973 17328 16096 31859
50244 59455 34690 83026 42522 30825 33446 85035 26193 11881
71010 00313 78387 52886 58753 32083 81420 61717 76691 47303
59825 34904 28755 46873 11595 62863 88235 37875 93751 95778
18577 80532 17122 68066 13001 92787 66111 95909 21642 01989
      828 function calls in 0.096 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      1      0.000      0.000      0.096      0.096 test5.py:1(<module>)
     200      0.000      0.000      0.001      0.000 test5.py:14(printw)
      1      0.095      0.095      0.096      0.096 test5.py:22(main)
     201      0.000      0.000      0.000      0.000 test5.py:6(pow)
      1      0.000      0.000      0.096      0.096 {built-in method builtins.exec}
     423      0.001      0.000      0.001      0.000 {built-in method builtins.print}

```

图 2 CPython 的运行结果



```
31 int main() {
32     int group_boundary = pow(base: 10, GROUP_DIGITS);
33     int max_output_digits;
34     max_output_digits = getint();
35     if (max_output_digits < 10) { max_output_digits = 10; }
36     int remainders_size = 4 * max_output_digits;
37     int output[max_output_digits], remainders[remainders_size];
38     int i = 0, output_index = 0, lower_digits = 0;
39     while (i < remainders_size) {
40         remainders[i] = 2;
41         i = i + 1;
42     }
43     i = remainders_size;
44     while (i > 0 && output_index < max_output_digits) {
45         // pi = 2 + 1/3 * (2 + 2/5 * (2 + 3/7 * (2 + ... (2 + 1/...)))
46         int sum = 0, numerator = i, denominator = i * 2 - 1;
47         while (numerator > 0) {
48             sum = sum + remainders[numerator] * group_boundary;
49             remainders[numerator] = sum % denominator;
50             sum = sum / denominator;
51             sum = sum * numerator;
52             numerator = numerator - 1;
53             denominator = denominator - 2;
54         }
55         output[output_index] = lower_digits + sum / group_boundary;
56         output_index = output_index + 1;
57         lower_digits = sum % group_boundary;
58         i = i - ITER_SPAN;
59     }
60     printf(_Format: "3.\n");
61     i = 1;
62     while (i * GROUP_DIGITS <= max_output_digits) {
63         printw(GROUP_DIGITS, output[i]);
64         if (i % GROUPS_EACH_LINE) { printf(_Format: " "); }
65         else { printf(_Format: "\n"); }
66         i = i + 1;
67     }
68     int remainder = max_output_digits % GROUP_DIGITS;
69     if (remainder) {
70         printw(remainder, n: output[max_output_digits / GROUP_DIGITS]);
71     }
```

图 3 计算 π 的程序


```

Run: Code x
195 LOAD_NAME remainder (INT, decla
196 LOAD_NAME output (INT_ARRAY, declare
197 LOAD_NAME max_output_digits (IN
198 LOAD_FAST GROUP_DIGITS (INT, decla
199 BINARY_OP /
200 LOAD_FAST 1
201 BINARY_OP +
202 SUBSCR_ARRAY
203 LOAD_FAST 10
204 LOAD_FAST GROUP_DIGITS (INT, decla
205 LOAD_NAME remainder (INT, decla
206 BINARY_OP -
207 CALL_FUNCTION pow (2 args, offset 2,
208 BINARY_OP /
209 CALL_FUNCTION printw (2 args, offset 19,
210 POP_TOP
211 LOAD_FAST 0
212 RETURN_VALUE

3.
14159 26535 89793 23846 26433 83279 50288 41971 69399 37510
58209 74944 59230 78164 06286 20899 86280 34825 34211 70679
82148 08651 32823 06647 09384 46095 50582 23172 53594 08128
48111 74502 84102 70193 85211 05559 64462 29489 54930 38196
44288 10975 66593 34461 28475 64823 37867 83165 27120 19091
45648 56692 34603 48610 45432 66482 13393 60726 02491 41273
72458 70066 06315 58817 48815 20920 96282 92540 91715 36436
78925 90360 01133 05305 48820 46652 13841 46951 94151 16094
33057 27036 57595 91953 09218 61173 81932 61179 31051 18548
07446 23799 62749 56735 18857 52724 89122 79381 83011 94912
98336 73362 44065 66430 86021 39494 63952 24737 19070 21798
60943 70277 05392 17176 29317 67523 84674 81846 76694 05132
00056 81271 45263 56082 77857 71342 75778 96091 73637 17872
14684 40901 22495 34301 46549 58537 10507 92279 68925 89235
42019 95611 21290 21960 86403 44181 59813 62977 47713 09960
51870 72113 49999 99837 29780 49951 05973 17328 16096 31859
50244 59455 34690 83026 42522 30825 33446 85035 26193 11881
71010 00313 78387 52886 58753 32083 81420 61717 76691 47303
59825 34904 28755 46873 11595 62863 88235 37875 93751 95778
18577 80532 17122 68066 13001 92787 66111 95909 21642 01989
Process finished in 0.604316 seconds

```

图 4 程序部分虚拟机指令及结果

6 总结与感想

这是我第一个完整的 C++ 项目和编译器项目。通过这个项目，我熟悉了现代 C++ 的基本概念和使用方法，熟悉了亲手从头构建一个编译器的过程，也对此保持着浓厚的兴趣与热情。由于在设计上比较谨慎，参考了 CPython 和 C4 这两个优秀的项目，架构比较容易维护，调试的过程还算比较顺利。在此向课程组推荐这两个项目，很有启发性，特别是 C4 项目用短短几百行只有 4 个函数的 C 代码就实现了一个能自己编译自己的 C 语言解释器，还有人在此基础上加了 86 行代码使之成为了一个 x86 架构下的即时（Just-in-Time, JIT）编译器，尽管写法有些“炫技”。我也认识到了编译器的复杂性，虽然自己亲手从头做一个并不难，但要做一个工业上能用的也不简单。我还认识到了编译技术的重要性，除了编译器，在游戏、音视频处理软件、科学绘图计算等专业软件中都有编译技术的身影。遗憾就是由于各种原因，未能实现生成 MIPS 代码并进行优化。我会不断了解最新的编译技术，并将这些知识服务于日常编程中，写出更高效的、对编译器优化友好的代码。

7 参考资料

- [1] <https://github.com/python/cpython/blob/main/Python/ceval.c>
- [2] <https://github.com/python/cpython/blob/main/Lib/opcode.py>
- [3] <https://github.com/rswier/c4/blob/master/c4.c>
- [4] <https://github.com/AoiKuiyuyou/AoikC4x86Study/blob/master/c4x86.c>