《编译原理课程设计》项目

类 C 语言编译器实验报告

205XXXX GTY

指导教师：WZH 老师

2023 年 3 月

目录

同濟大學
1907
TONGJI UNIVERSITY

一、实验内容

本课程设计项目旨在对高级语言编译的基本过程有全面了解后，自主设计一款可以将类 C 语言代码转换成汇编代码的编译程序。

该项目总共分为四大部分：词法分析、语法分析、语义分析和中间代码生成、汇编代码生成。其中，前三部分在去年秋季学期完成，汇编代码生成在本年春季学期完成。

编译程序需要支持 C 语言标准的子集，拥有对分支结构、循环结构、函数调用等语句的翻译能力。最终生成的汇编程序应能由汇编器翻译成真正的二进制码。

二、需求分析

2.1 前阶段提供的基础

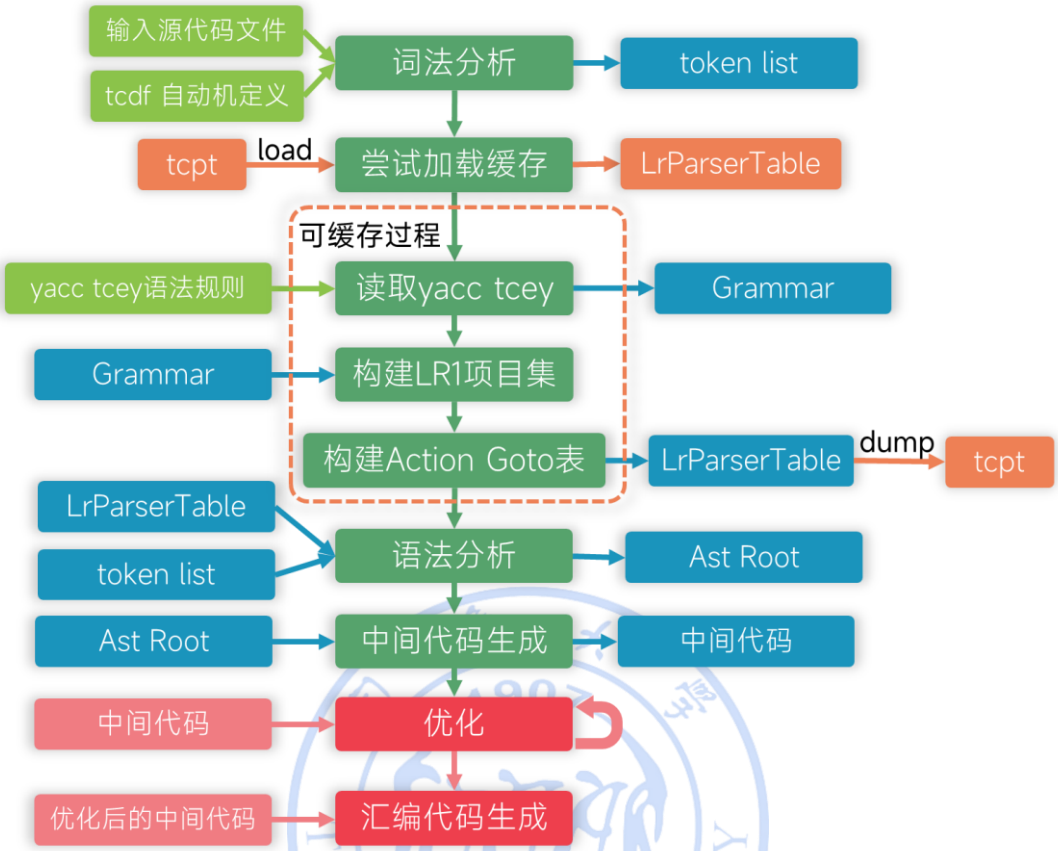
前阶段，我们实现三个模块。

在第一个模块中，我们成功实现 C++11 语言词法分析。

在第二个模块中，我们基于词法分析结果，按照 C99 文法 [4] 完成语法树构建（文法规则见附录 1），得到语法树的根节点。该节点类型为“翻译单元（translation unit）”，沿该节点遍历，可以得到整个源程序的结构，含每个符号的内容和位置，及它在语法中的含义。

在第三个模块中，我们参考多个资料，设计一个比四元式更为复杂的中间代码格式。我们称之为 TCIR（ToyCompile Intermediate Language），并将语法树转换成该中间语言形式。

2.2 本阶段在整个项目中的位置



上图为本项目的整体任务流程，需要对源代码文件进行词法分析，根据特定文法做语法分析，将语法树转为中间代码，在优化后转换成汇编代码。

本阶段在图中以茶花红颜色标注。

我们尽力令使中间代码更为通用，生成器结构更为简洁，这导致生成的中间代码效率较差。我们需要一个简单的优化器，对中间代码进行多轮修剪，在不改变程序执行结果的前提下降低代码量，提升运行效率。

基于优化后的中间代码，我们需要按照一定规则将它转换成汇编代码。在此，我们选择以 IA-32 架构下的汇编格式作为生成目标，并在生成后用 nasm 对其进行汇编，以检查其准确性。

2.3 本阶段任务

含 2 个主要子模块：优化器和汇编代码生成器。事实上，本阶段还需要完成将文本格式的中间代码加载到内存格式的任务。我们称之为中间代码加载器。

中间代码加载器的输入内容为文本格式的中间代码。加载器分析其内容，还原生成中间代码的过程中，内存里保存的全局符号表、块符号表树等结构。同时，需要对它们做一些额外处理，完成栈地址偏移计算等工作。加载完毕，将中间代码的执行代码部分交给优化器模块。

优化器的输入是中间代码的可执行部分。它按照指定的几条规则，检查中间代码，在合适的位置进行优化。

汇编代码生成器的输入是优化后的中间代码可执行部分。同时，它可以访问内存中保存的各种符号表结构。

三、概要设计

3.1 中间代码加载器

该部分输入为文本格式的中间代码。由于中间代码仅在程序内部传递，不向用户提供直接编辑的能力，我们可以直接认为输入的中间代码是严格遵循我们设计的格式的。

TCIR 将中间代码切分成多个子区域。其中，可执行代码区是最后一个区域，即在处理可执行代码前，必须保证外部链接区、全局符号表区、块符号表区等区域已经加载完毕。事实上，中间代码生成器在生成中间代码时，会保障以上顺序要求。

根据不同区域，激活不同部分的构造器，填充汇编代码生成器的内部辅助结构。最后，提取可执行代码部分，传递给下一个模块。

3.2 优化器

该部分输入为中间代码加载器处理得到的中间代码可执行代码部分。它会对中间代码进行多轮扫描，检测多个可优化点。一旦发现，便执行预设的优化方案，在不影响执行结果的前提下减少代码量，提高运行效率。

3.3 IA-32 汇编生成器

汇编生成器的任务是将中间代码转换成汇编指令。由于汇编器技术已经十分成熟，且汇编代码与二进制指令对应关系十分简单，本项目仅实现到输出汇编的后端，点到为止。

汇编生成器应支持多种不同架构的后端。本项目中，我们仅实现 Intel 386 格式的汇编。

四、详细设计

4.1 中间代码加载器

中间代码加载器的输入内容为文本格式的中间代码。该中间代码由我们的程序内部产生，可以保证其严格遵循预先设计的中间代码格式规范。

加载器首先向目标文件输出基本信息，如“[bits 32]”和“section .text”。之后，不断读入中间代码，并在检测到代码区域属性后，调用不同的处理子程序。

每区域由两个以“@”开头的行组成。开始部分由外部识别机制处理，子程序需要负责清除结尾部分。详见附录 2。

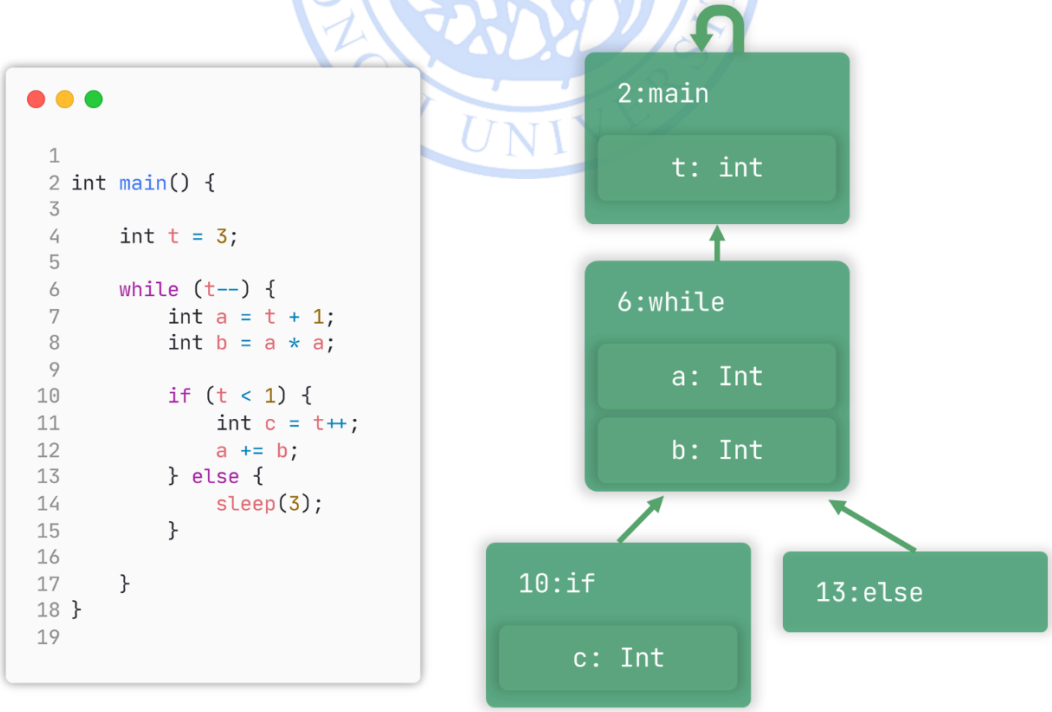
4.1.1 构建全局描述符表

全局描述符表负责登记全局区域的函数和变量。

根据中间代码，不断处理表中的每个子项。识别当前处理的是函数或变量。

对于函数，需要读取其参数表长度，将每个参数的信息读入，并保存。由于参数表内允许出现“void”，我们需要对这种情况做特殊处理，不应让“void”被登记到参数表内。

4.1.2 构建局部描述符表树



我们为程序内的每个复合代码（代码块）构造一个局部描述符表，描述符表之间构成一个树状结构。

生成中间代码时，若检测到某个变量，程序将从当前符号表开始，一路向上查找，直到在某个符号表内找到该变量符号。若无法找到，则前往全局符号表查找。若仍无法找到，则向用户报错。

在输出中间代码时，块符号表的唯一识别标记（id）、内部成员信息和父节点唯一识别标记被登记。我们需要将这些内容还原到内存里。

选用基于红黑树的 map 作为容器，id 作为 key，在容器内构造符号表。所有符号表构造完毕，对容器做遍历，绑定每张表的亲子关系信息。

之后，我们借助深度优先搜索，为每个符号分配栈上偏移空间。不难意识到，同级符号可以使用栈上同一位置的内存，因为它们的生存期没有重叠。但不同级之间的符号不能使用同一个内存，因为它们的生存期是重叠的。

4.1.3 外部链接处理

外部链接包含“导入”和“导出”两大部分。

对于导入的符号，向目标文件输出“extern”，再输出符号名称。

对于导出的符号，向目标文件输出“global”，再输出符号名称。

4.1.4 静态数据处理

静态数据区主要是全局变量。我们直接将它们存储到代码内。

由于我们已经规定，所有变量的类型都是 int32，这部分变得十分简单。我们首先输出一个“align 4”，再借助“db”和位运算，将变量固定到文件内。

4.1.5 处理可执行指令区

优化器希望读到的代码没有令人烦恼的空格等空白符号，希望每条指令已经被很好地切分开。这些工作正是可执行指令区处理程序的任务。

程序从输入流不断读取字符，过滤其中空白符号，提取语句中的一个个词，并根据换行分割成不同语句。将结果列表交给优化器执行后续流程。

4.2 优化器

本优化器作用于中间代码之上。前文已经描述，中间代码生成器可能会生成一些低效率的代码，这些代码将由优化器进行处理。

当检测到连续的函数返回语句，将靠后的那个语句删除。

当检测到成对的入栈和出栈语句，将两条语句删除。

当检测到重复移动语句，将下一条语句删除。

当检测到同目标移动语句，将当前语句删除。

当检测到循环移动语句，将下一条语句删除。

当检测到三步内出现借助栈实现的虚拟寄存器之间的值交换，将其替换成一条“xchg”指令即可。

当检测到三步内存在用栈实现虚拟寄存器值拷贝时，将其优化成简单的移动指令。

显然，我们的优化器实现的功能较少，仅在重复语句执行部分优化。后续，可以在中间代码规范中加入不设上限的虚拟寄存器，在生成实际代码时动态映射到物理寄存器和栈上，以充分利用现代 CPU 内的海量寄存器。对于一些执行结果固定的语句段，可以进行识别，在编译期计算得到结果，以减少运行时的计算开销。

4.3 中间代码生成器总体工作流程

中间代码生成器接收的输入为中间代码语句列表。该列表经过优化器处理。

目标设备是 Intel 386，中间代码内仅有 2 个虚拟寄存器。为简化，我们直接将它们分别映射到 eax 和 edx。

遵循 Intel 386 平台的 System V ABI[14]，函数调用参数通过栈传递。当我们需要读取传入参数时，根据参数在参数表内的位置，计算得到其相对 ebp 寄存器的偏移，并生成寻址代码。

对于局部变量，我们基于预先计算得到的栈上位置值，并生成基于 ebp 寄存器的寻址代码。

在函数入口处，我们需要提前计算函数执行过程需要为栈变量分配的最大内存，并预先将 esp 寄存器移动到界限处。该设计参考 gcc 的行为。当然，我们需要在函数返回前生成恢复栈帧的代码。

函数调用时，由于参数已经在前面的代码里传递（对应中间代码的“pushfc”指令），我们只需要简单生成一个 call 指令即可。在 call 之后，我们需要根据被调用函数的参数表大小，生成恢复 esp 的代码。

由于我们的中间代码和 Intel 汇编较为相似，其他中间代码的转换流程较为简单。

五、调试分析

5.1 正确的样例：简单代码

```
int a;

int f(int x) {
    int y = x + 1;
    return a;
}

```

使用下方命令运行：

```
./ToyCompile sUniCli -fname:resources/test/easy2.c -o-std
```

得到结果如下：

```
; generated by ToyCompile
; for intel 386 protected mode environment

[bits 32]
section .text

global f
global a
align 4
a:
  db 0, 0, 0, 0

f:
  push ebp
  mov ebp, esp
  sub esp, 4
  mov dword eax, [ebp + 8]
  mov dword edx, eax
  mov dword eax, 1
  add dword edx, eax
  xchg eax, edx
  mov dword [ebp -4], eax
  mov dword eax, [a]
  leave
  ret
```

文本起始，声明代码位数和代码段信息。之后，导出全局符号 f 和 a。

对于全局变量 a，4 字节对齐后，为其保留空间。

对于函数 f，生成汇编指令。我们用人类大脑模拟运行这段代码，可以看出，这段代码的逻辑是正确的。

5.2 错误的样例：引用未定义符号

测试代码如下：

```c
int x;
int q = 3;
int x;

int f() {

    int c = 1;
    y = x + 1 + c;

    return z;

}

```

观察该代码，可以注意到以下问题：

1. 全局区域，变量 x 重复定义；

2. 函数 f 内，引用了未定义的变量 y 和 z。

我们尝试对其进行中间代码生成，得到输出结果如下：

```
flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$
 ./ToyCompile sUniCli -fname:resources/test/easy2.c
warning: symbol redefined: x
  token: x
  loc  : (3, 5)
error: symbol not found: y
  token: y
  loc  : (8, 5)
error: symbol not found: z
  token: z
  loc  : (10, 12)
flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$
```

可见，中间代码生成器成功发现这些错误，并给出提示信息。由于源文件存在致命错误（error），其得到的中间代码不可用。

5.3 全局变量数值编译期计算测试

测试代码如下：

```
1
2  int a = 1, b, c = 2;
3  int d = 2 + 3 * 4 - 5;  // 9
4  int d2 = 2 + 3 * ((4 - 5) * 3); // -7
5  int e = 1 && 2 || 3;  // 非 0
6
7  /*
8      8: 1000
9      6: 0110
10     5: 0101
11
12     0110 & 0101 → 0100
13     1000 | 0100 → 1100
14 */
15 int f = 8 | 6 & 5;  // 12
16
17 /*
18     4: 0100
19     9: 1001
20     0100 ^ 1001 → 1101
21 */
22 int g = 4 ^ 9;  // 13
23
24 int h = 7 && (3 + 4 - 7); // 0
25 int i = (3, 4 + 5, 6 * 7, 8 + 2);  // 10
26 int j = 20 / 7;  // 2
27
```

执行得到结果如下：

```
; generated by ToyCompile
; for intel 386 protected mode environment

[bits 32]
section .text

global a
global b
global c
global d
global d2
global e
global f
global g
global h
global i
global j
align 4
a:
  db 1, 0, 0, 0
align 4
b:
  db 0, 0, 0, 0
align 4
c:
  db 2, 0, 0, 0
align 4
d:
  db 9, 0, 0, 0
align 4
d2:
  db 249, 0, 0, 0
align 4
e:
  db 1, 0, 0, 0
align 4
f:
  db 12, 0, 0, 0
align 4
g:
  db 13, 0, 0, 0
align 4
h:
  db 0, 0, 0, 0
align 4
i:
  db 10, 0, 0, 0
align 4
j:
  db 2, 0, 0, 0
```

不难发现，程序正确地为他们完成空间分配，并完成值初始化。

5.4 设计任务书上的测试样例

我们对任务书上的第一个测试样例进行测试。

由于该样例的第 35 行缺少分号，导致程序报错。

```
parser error: (36, 2) unexpected token: return. at: (36, 2), return.
```

手动修正错误，再次进行编译，得到结果如下：

```
; generated by ToyCompile
; for intel 386 protected mode environment

[bits 32]
section .text

global demo
global main
global program

program:
  push ebp
  mov ebp, esp
  sub esp, 8
  mov dword eax, 0
  mov dword [ebp -4], eax
  mov dword eax, [ebp + 8]
  push eax
  mov dword eax, [ebp + 12]
  mov dword edx, eax
  mov dword eax, [ebp + 16]
  add dword edx, eax
  xchg eax, edx
  pop edx
  cmp dword edx, eax
  je .if_else_2
  mov dword eax, [ebp + 8]
  push eax
  mov dword eax, [ebp + 12]
  mov dword edx, eax
  mov dword eax, [ebp + 16]
  imul dword eax, edx
  mov dword edx, eax
  mov dword eax, 1
  add dword edx, eax
  xchg eax, edx
  pop edx
  add dword edx, eax
  xchg eax, edx
  mov dword [ebp ], eax
  jmp .if_end_1
.if_else_2:
  mov dword eax, [ebp + 8]
  mov dword [ebp ], eax
.if_end_1:
.while_loop_exp_3:
  mov dword eax, [ebp -4]
  mov dword edx, eax
```

```
  mov dword eax, 100
  cmp dword edx, eax
  je .while_loop_end_3
.while_loop_stmt_3:
  mov dword eax, [ebp ]
  mov dword edx, eax
  mov dword eax, 2
  imul dword eax, edx
  mov dword [ebp -4], eax
  jmp .while_loop_exp_3
.while_loop_end_3:
  mov dword eax, [ebp -4]
  leave
  ret


demo:
  push ebp
  mov ebp, esp
  mov dword eax, [ebp + 8]
  mov dword edx, eax
  mov dword eax, 2
  add dword edx, eax
  xchg eax, edx
  mov dword [ebp + 8], eax
  mov dword edx, eax
  mov dword eax, 2
  imul dword eax, edx
  leave
  ret


main:
  push ebp
  mov ebp, esp
  sub esp, 12
  mov dword eax, 3
  mov dword [ebp -4], eax
  mov dword eax, 4
  mov dword [ebp ], eax
  mov dword eax, 2
  mov dword [ebp + 4], eax
  mov dword eax, [ebp -4]
  push eax
  mov dword eax, [ebp ]
  push eax
  mov dword eax, [ebp + 4]
  push eax
  call demo
  add esp, 4
```

```
push eax
call program
add esp, 12
mov dword [ebp -4], eax
leave
ret
```

我们用人类大脑分析以上代码，可以认为，该代码的逻辑是正确的。

我们用 nasm 汇编器对生成的汇编代码进行汇编，设置输出格式为 ELF32。汇编器成功完成任务。可以认为，我们的程序生成的汇编代码在语法上是正确的。

进一步，我们用 readelf 工具解析代码：

```
Symbol table '.symtab' contains 11 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS book.asm
     2: 00000000     0 SECTION LOCAL  DEFAULT    1 .text
     3: 00000044     0 NOTYPE  LOCAL  DEFAULT    1 program.if_else_2
     4: 0000004a     0 NOTYPE  LOCAL  DEFAULT    1 program.if_end_1
     5: 0000004a     0 NOTYPE  LOCAL  DEFAULT    1 program.while_lo[...]
     6: 00000058     0 NOTYPE  LOCAL  DEFAULT    1 program.while_lo[...]
     7: 0000006a     0 NOTYPE  LOCAL  DEFAULT    1 program.while_lo[...]
     8: 00000000     0 NOTYPE  GLOBAL DEFAULT    1 program
     9: 0000006f     0 NOTYPE  GLOBAL DEFAULT    1 demo
    10: 0000008e     0 NOTYPE  GLOBAL DEFAULT    1 main
```

可见，我们的标签导出设置是正确的。

5.5 算法复杂度分析

加载过程中，加载器不断提取输入流中的内容，并作简单处理，复杂度与中间代码长度成线性关系。

优化过程中，若检测到可优化点，会使目标代码规模减小。时间复杂度与目标代码规模呈正相关，实际应乘以一个较大的常数。

生成代码时，执行方法几乎为一一对应，时间复杂度与中间代码长度成线性关系。

5.6 存在的问题

5.6.1 有待改进的中间代码格式

本项目中间代码有较大改进空间。现有的中间代码较为简单，需要用较复杂方式实现部分指令。

我们可以在中间代码内加入不限量的虚拟寄存器（参考 LLVM Clang），在生成汇编代码时动态分配物理寄存器，以充分利用目标机型的物理性能。

5.6.2 对数组的支持

我们的中间代码生成器暂未对数组提供支持。

数组本质是连续存储空间和偏移寻址。该过程仅需要在中间代码生成阶段处理，相关逻辑在生成汇编代码前已经被转换。

5.6.3 对多构建目标的支持

本项目仅支持生成 Intel 386 格式的汇编代码。事实上，我们应该设计一个通用的构造器基础类，在实际构造时，根据目标设备设置，选择不同的构建目标，以体现中间代码机器无关的通用属性，满足不同用户的需求。

六、用户使用说明

6.1 命令行使用说明

为引导用户使用，我们设计简单易懂的使用说明。

使用 -help 标签，即可看到使用说明：

```
flowerblack@ubuntu-mibookprox-gty:~/code-repo/ToyCompile/build$ ./ToyCompile sUniCli -help

ToyCompile Unified CommandLine

params:
  fname:[x]       : specify input file 'x'.
  help            : get help.

  dump-tokens     : dump tokens.

  rebuild-table   : reload parser table from tcey file.
                    if not set, parser would try to load cache
                    to improve performance.
  no-store-table  : don't store built table to file.
  cache-table:[x]: specify cache table file.
  tcey:[x]        : set tcey file 'x'.
  dump-ast        : dump parser result.
  dot-file:[x]    : store parser result to file 'x'.

  dump-ir         : dump toycompile ir code.
  ir-to-file:[x]  : store ir code to file.
  disable-color   : disable color to log output stream.

  -o-std           : put asm to stdout.
  -o:[file]       : asm output file.

must have:
  fname:[x]

examples:
  ParserCli -fname:resources/test/easy.c.txt -rebuild-table
```

6.2 输入文件

我们要求从文件输入 C 语言代码。通过设置 fname，指定输入的代码源文件。

6.3 中间过程显示

6.3.1 输出词法分析结果

通过设置 dump-tokens，可以令程序输出词法分析的结果。

6.3.2 输出语法树

通过设置 dump-ast，可以令程序以 dot 格式输出语法树结构。

6.3.3 输出中间代码

通过设置 dump-ir，可以令程序输出 tcir 格式的中间代码。该代码默认以带颜色的文本格式输出到标准输出。

若需要禁用颜色，可以设置 disable-color 属性。

若需要将中间代码输出到文件，可以设置 ir-to-file 属性，并设置目标文件名称。此时，会自动启用 disable-color 属性。

6.4 Action Goto 表缓存控制

由于我们采用的是完整的 C99 语法，规模较大，Action Goto 表构建时间较长，我们引入了缓存机制。当启动时，程序自动检测缓存文件是否存在。若存在，则直接加载，实现极速响应用户请求。否则，会重新构造表，并保存到本地文件，以便后续使用。这些过程可以通过参数控制。

如果我们希望强制重新构造 Action Goto 表，可以通过设置 rebuild-table 属性实现。

如果我们不希望程序保存已经构建的表，可以设置 no-store-table 属性。

如果我们希望指定以某个文件作为表缓存，可以将其设置为 cache-table 的值。

6.5 汇编代码输出

程序的最终目标是输出汇编代码。可以通过设置 o-std 属性以让程序将汇编代码输出到标准输出。通过设置 o 属性的值，可以指定输出的文件。两种模式生成的汇编代码皆不包含颜色控制信息。

o 属性会覆盖 o-std 属性。

如果 o 和 o-std 都没有被设置，程序将不执行汇编代码生成过程。

七、总结与收获

7.1 项目收获

本项目中，我们使用 C++17 语言完成一个完整的类 C 语言编译器。相比 GCC、MSVC 等商用编译器，我们的编译器显得十分简陋。但作为学习项目，它令我们对编译过程得到较深入的理解。研究过程中，我们不断巩固在课堂上学到的知识，并应用到实践。同时，我们也在努力优化项目结构，增强代码可读性，制作能够指引学弟学妹们参考学习的产品。通过本项目的开发，我们亦对 C++语言收获更多理解，并看到无数前人努力的身影。现在的我们拥有很幸福的开发环境。当我们打下一个字符，语言服务器会自动为我们提供输入提示，并进行潜在错误的分析。多文件项目的构建变得几乎无感。这一切都是前人们努力的成果。

7.2 遇到的问题及其解决

研发过程中，我们坚持先规划后开发，在逻辑层面未遇到任何问题。然而，由于选用的 C99 语法系统过于庞大，为中间代码的设计带来很大困难。我们成功设计一个满足要求的中间代码，但它显得过于简单，导致生成的程序较为复杂（即使经过优化器的处理）。

我们可以学习 LLVM IR 格式，对自己的中间代码加以改进，以提升生成代码质量，充分利用物理设备的能力。

7.3 课程认识

《编译原理》作为计算机系的一大重要课程，在整个教学体系内扮演不可替代的作用。作为计算机系的学生，学习该课程是研究计算机工作原理时不可少的一部分。

课程上，我们学习到的是相对理论的知识，是对实际应用的抽象。而在课程设计项目中，我们将理论应用到实践，巩固课堂所学的知识，感受前人留下的智慧，收获颇丰。

正如枫铃树在文章《红黑树详解》结尾处所述，上世纪的智者将一个个伟大的火把交到我们手中，我们要牢牢握住，并用它照亮前方的路，点燃远方的灯[12]。上世纪的贤者们开创形式语言相关理论，提出相关方法，制作大量工具。他们的努力为我们带来无比幸福的学习研究环境。我们接过他们的理论，重温经典，力求在经典之上有所创新，将知识的火把一代代传递下去。

7.4 致谢

本项目的诞生与成长离不开卫志华老师的课程教学与答疑帮助。卫老师的课程十分通俗易懂，且很有意思，让我们更愉悦地获取知识。感谢卫老师的辛勤付出。

实现细节部分参考高 QT 学姐和叶 MY 学长的项目，以及 LLVM Clang 项目。学习他们精妙的设计，改进我们的实现。感谢他们的努力与开源的决定。本项目也在 GitHub 开源，以向他们致谢。

项目实现过程中，部分疑难点与曾 SR 同学讨论并解决。感谢她对我的帮助。

八、获取产品

8.1 获取源码

本项目在 GitHub 开源。前往项目仓库以获取源代码：

https://github.com/FlowerBlackG/ToyCompile

8.2 构建运行

项目使用 CMake 管理，依赖 Boost 库。对于 Windows GCC 环境，Boost 库已经内置。对于其他环境，需自行解决依赖问题。

程序运行依赖词法自动机定义文件（tcdf）和语法定义文件（tcey）。这些文件已经放置在 resources 文件夹内，默认情况会自动加载。

编译构建，启动构建产物 ToyCompile.exe，即可体验产品功能。

参考资料

[1] 陈火旺等. 程序设计语言编译原理（第 3 版）. 国防工业出版社, 2000

[2] Maoyao233. ToyCC. https://github.com/Maoyao233/ToyCC

[3] GQT. 词法分析器. 2021

[4] Jutta Degener. ANSI C Yacc grammar. 2012

[5] 王爽. 汇编语言（第 3 版）. 清华大学出版社，2013

[6] LLVM. LLVM Tutorial. https://llvm.org/docs/tutorial/

[7] Evian Zhang. llvm ir tutorial. https://github.com/Evian-Zhang/llvm-ir-tutorial

[8] 踌躇月光. 操作系统实现. bilibili

[9] ZingLix. 8086 汇编指令集整理. ZingLix Blog, 2018

[10] 华为. MAPLE IR Specification. https://gitee.com/openarkcompiler/OpenArkCompiler/blob/master/doc/en/MapleIRDesign.md

[11] 同济大学计算机系. 第七章 语义分析和中间代码产生. 同济大学计算机系

[12] 枫铃树. 红黑树详解（下）（红黑树的删除操作）. CSDN, 2022

[13] intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. December 2022

[14] OSDev. System V ABI. OSDev Wiki, https://wiki.osdev.org/System_V_ABI

附录 1：C99 文法

```
/*

ANSI C Yacc grammar (This yacc file is accompanied by a matching lex file.)

In 1985, Jeff Lee published his Yacc grammar for the April 30, 1985 draft version of the ANSI C standard.  Tom Stockfisch reposted
it to net.sources in 1987; that original, as mentioned in the answer to question 17.25 of the comp.lang.c FAQ, used to be available
via ftp from ftp.uu.net as usenet/net.sources/ansi.c.grammar.Z

The version you see here has been updated based on an 1999 draft of the standards document. It allows for restricted pointers,
variable arrays, "inline", and designated initializers. The previous version's lex and yacc files (ANSI C as of ca 1995) are still
around as archived copies.

I want to keep this version as close to the 1999 Standard C grammar as possible; please let me know if you discover discrepancies.
(If you feel like it, read the FAQ first.)

Jutta Degener, 2012

*/

%token IDENTIFIER CONSTANT STRING_LITERAL SIZEOF
%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%token TYPEDEF EXTERN STATIC AUTO REGISTER INLINE RESTRICT
%token CHAR SHORT INT LONG SIGNED UNSIGNED FLOAT DOUBLE CONST VOLATILE VOID
%token BOOL COMPLEX IMAGINARY
%token STRUCT UNION ENUM ELLIPSIS

%token CASE DEFAULT IF ELSE SWITCH WHILE DO FOR GOTO CONTINUE BREAK RETURN

%start translation_unit
%%

primary_expression
    : IDENTIFIER
    | CONSTANT
    | STRING_LITERAL
    | '(' expression ')'
    ;

postfix_expression
    : primary_expression
    | postfix_expression '[' expression ']'
    | postfix_expression '(' ')'
    | postfix_expression '(' argument_expression_list ')'
    | postfix_expression '.' IDENTIFIER
    | postfix_expression PTR_OP IDENTIFIER
    | postfix_expression INC_OP
    | postfix_expression DEC_OP
    | '(' type_name ')' '{' initializer_list '}'
    | '(' type_name ')' '{' initializer_list ',' '}'
    ;

argument_expression_list
    : assignment_expression
    | argument_expression_list ',' assignment_expression
    ;

unary_expression
    : postfix_expression
    | INC_OP unary_expression
    | DEC_OP unary_expression
    | unary_operator cast_expression
    | SIZEOF unary_expression
    | SIZEOF '(' type_name ')'
    ;
```

```
unary_operator
    : '&'
    | '*'
    | '+'
    | '-'
    | '~'
    | '!'
    ;

cast_expression
    : unary_expression
    | '(' type_name ')' cast_expression
    ;

multiplicative_expression
    : cast_expression
    | multiplicative_expression '*' cast_expression
    | multiplicative_expression '/' cast_expression
    | multiplicative_expression '%' cast_expression
    ;

additive_expression
    : multiplicative_expression
    | additive_expression '+' multiplicative_expression
    | additive_expression '-' multiplicative_expression
    ;

shift_expression
    : additive_expression
    | shift_expression LEFT_OP additive_expression
    | shift_expression RIGHT_OP additive_expression
    ;

relational_expression
    : shift_expression
    | relational_expression '<' shift_expression
    | relational_expression '>' shift_expression
    | relational_expression LE_OP shift_expression
    | relational_expression GE_OP shift_expression
    ;

equality_expression
    : relational_expression
    | equality_expression EQ_OP relational_expression
    | equality_expression NE_OP relational_expression
    ;

and_expression
    : equality_expression
    | and_expression '&' equality_expression
    ;

exclusive_or_expression
    : and_expression
    | exclusive_or_expression '^' and_expression
    ;

inclusive_or_expression
    : exclusive_or_expression
    | inclusive_or_expression '|' exclusive_or_expression
    ;

logical_and_expression
    : inclusive_or_expression
    | logical_and_expression AND_OP inclusive_or_expression
    ;

logical_or_expression
    : logical_and_expression
    | logical_or_expression OR_OP logical_and_expression
    ;
```

```
conditional_expression
    : logical_or_expression
    | logical_or_expression '?' expression ':' conditional_expression
    ;

assignment_expression
    : conditional_expression
    | unary_expression assignment_operator assignment_expression
    ;

assignment_operator
    : '='
    | MUL_ASSIGN
    | DIV_ASSIGN
    | MOD_ASSIGN
    | ADD_ASSIGN
    | SUB_ASSIGN
    | LEFT_ASSIGN
    | RIGHT_ASSIGN
    | AND_ASSIGN
    | XOR_ASSIGN
    | OR_ASSIGN
    ;

expression
    : assignment_expression
    | expression ',' assignment_expression
    ;

constant_expression
    : conditional_expression
    ;

declaration
    : declaration_specifiers ';'
    | declaration_specifiers init_declarator_list ';'
    ;

declaration_specifiers
    : storage_class_specifier
    | storage_class_specifier declaration_specifiers
    | type_specifier
    | type_specifier declaration_specifiers
    | type_qualifier
    | type_qualifier declaration_specifiers
    | function_specifier
    | function_specifier declaration_specifiers
    ;

init_declarator_list
    : init_declarator
    | init_declarator_list ',' init_declarator
    ;

init_declarator
    : declarator
    | declarator '=' initializer
    ;

storage_class_specifier
    : TYPEDEF
    | EXTERN
    | STATIC
    | AUTO
    | REGISTER
    ;

type_specifier
    : VOID
    | CHAR
    | SHORT
    | INT
```

```
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    | BOOL
    | COMPLEX
    | IMAGINARY
    | struct_or_union_specifier
    | enum_specifier
    | TYPE_NAME
    ;

struct_or_union_specifier
    : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
    | struct_or_union '{' struct_declaration_list '}'
    | struct_or_union IDENTIFIER
    ;

struct_or_union
    : STRUCT
    | UNION
    ;

struct_declaration_list
    : struct_declaration
    | struct_declaration_list struct_declaration
    ;

struct_declaration
    : specifier_qualifier_list struct_declarator_list ';'
    ;

specifier_qualifier_list
    : type_specifier specifier_qualifier_list
    | type_specifier
    | type_qualifier specifier_qualifier_list
    | type_qualifier
    ;

struct_declarator_list
    : struct_declarator
    | struct_declarator_list ',' struct_declarator
    ;

struct_declarator
    : declarator
    | ':' constant_expression
    | declarator ':' constant_expression
    ;

enum_specifier
    : ENUM '{' enumerator_list '}'
    | ENUM IDENTIFIER '{' enumerator_list '}'
    | ENUM '{' enumerator_list ',' '}'
    | ENUM IDENTIFIER '{' enumerator_list ',' '}'
    | ENUM IDENTIFIER
    ;

enumerator_list
    : enumerator
    | enumerator_list ',' enumerator
    ;

enumerator
    : IDENTIFIER
    | IDENTIFIER '=' constant_expression
    ;

type_qualifier
    : CONST
    | RESTRICT
```

```
    | VOLATILE
    ;

function_specifier
    : INLINE
    ;

declarator
    : pointer direct_declarator
    | direct_declarator
    ;

direct_declarator
    : IDENTIFIER
    | '(' declarator ')'
    | direct_declarator '[' type_qualifier_list assignment_expression ']'
    | direct_declarator '[' type_qualifier_list ']'
    | direct_declarator '[' assignment_expression ']'
    | direct_declarator '[' STATIC type_qualifier_list assignment_expression ']'
    | direct_declarator '[' type_qualifier_list STATIC assignment_expression ']'
    | direct_declarator '[' type_qualifier_list '*' ']'
    | direct_declarator '[' '*' ']'
    | direct_declarator '[' ']'
    | direct_declarator '(' parameter_type_list ')'
    | direct_declarator '(' identifier_list ')'
    | direct_declarator '(' ')'
    ;

pointer
    : '*'
    | '*' type_qualifier_list
    | '*' pointer
    | '*' type_qualifier_list pointer
    ;

type_qualifier_list
    : type_qualifier
    | type_qualifier_list type_qualifier
    ;

parameter_type_list
    : parameter_list
    | parameter_list ',' ELLIPSIS
    ;

parameter_list
    : parameter_declaration
    | parameter_list ',' parameter_declaration
    ;

parameter_declaration
    : declaration_specifiers declarator
    | declaration_specifiers abstract_declarator
    | declaration_specifiers
    ;

identifier_list
    : IDENTIFIER
    | identifier_list ',' IDENTIFIER
    ;

type_name
    : specifier_qualifier_list
    | specifier_qualifier_list abstract_declarator
    ;

abstract_declarator
    : pointer
    | direct_abstract_declarator
    | pointer direct_abstract_declarator
    ;
```

```
direct_abstract_declarator
    : '(' abstract_declarator ')'
    | '[' ']'
    | '[' assignment_expression ']'
    | direct_abstract_declarator '[' ']'
    | direct_abstract_declarator '[' assignment_expression ']'
    | '[' '*' ']'
    | direct_abstract_declarator '[' '*' ']'
    | '(' ')'
    | '(' parameter_type_list ')'
    | direct_abstract_declarator '(' ')'
    | direct_abstract_declarator '(' parameter_type_list ')'
    ;

initializer
    : assignment_expression
    | '{' initializer_list '}'
    | '{' initializer_list ',' '}'
    ;

initializer_list
    : initializer
    | designation initializer
    | initializer_list ',' initializer
    | initializer_list ',' designation initializer
    ;

designation
    : designator_list '='
    ;

designator_list
    : designator
    | designator_list designator
    ;

designator
    : '[' constant_expression ']'
    | '.' IDENTIFIER
    ;

statement
    : labeled_statement
    | compound_statement
    | expression_statement
    | selection_statement
    | iteration_statement
    | jump_statement
    ;

labeled_statement
    : IDENTIFIER ':' statement
    | CASE constant_expression ':' statement
    | DEFAULT ':' statement
    ;

compound_statement
    : '{' '}'
    | '{' block_item_list '}'
    ;

block_item_list
    : block_item
    | block_item_list block_item
    ;

block_item
    : declaration
    | statement
    ;
```

```
expression_statement
    : ';'
    | expression ';'
    ;

selection_statement
    : IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
    | SWITCH '(' expression ')' statement
    ;

iteration_statement
    : WHILE '(' expression ')' statement
    | DO statement WHILE '(' expression ')' ';'
    | FOR '(' expression_statement expression_statement ')' statement
    | FOR '(' expression_statement expression_statement expression ')' statement
    | FOR '(' declaration expression_statement ')' statement
    | FOR '(' declaration expression_statement expression ')' statement
    ;

jump_statement
    : GOTO IDENTIFIER ';'
    | CONTINUE ';'
    | BREAK ';'
    | RETURN ';'
    | RETURN expression ';'
    ;

translation_unit
    : external_declaration
    | translation_unit external_declaration
    ;

external_declaration
    : function_definition
    | declaration
    ;

function_definition
    : declaration_specifiers declarator declaration_list compound_statement
    | declaration_specifiers declarator compound_statement
    ;

declaration_list
    : declaration
    | declaration_list declaration
    ;
%%
```

附录 2：TCIR 中间代码设计

位置

ToyCompile IR（tcir）处于语法树和汇编代码之间。

我们将基于 C99 文法生成的语法树转换成 tcir，再将后者转换成汇编语言，再由汇编编译器转换为可重定向二进制文件。

文件后缀（推荐）

`tcir`

TCIR 整体结构

存储形式

使用字符形式存储，存放于文件、字符串或字符流中。文件编码必须是纯 ascii，暂不支持中文等多字节符号（特殊说明除外）。

通过换行符号标记每条指令的结尾，即：每个指令一行，每条信息一行。

每行的前缀和后缀空白符号会被过滤。

通过空格或 tab 隔离符号，即每个符号之间都要有空格，符号之间不能有逗号等奇怪的东西。

功能块设定

IR 包含几大部分，每部分有不同功能。通过以下方式标记一个区域的开始和结尾：

@ begin of ${name}

${body}

@ end of ${name}

其中，${name} 是功能区名，要按规定设置。后续会有描述。

${body} 内是该功能区的内容。具体解析方式取决于功能区名。

功能块不允许重复出现，不允许嵌套出现。

单行注释

为方便加入说明信息，tcir 支持内嵌注释。

注释格式：

// ${comment}

当检测到 // 其后（包括其本身）内容将被忽略。当然，单双引号区间内（字符串内）的双斜线不遵循此规则。

虚拟寄存器

为便于表达式计算，设计 2 个 32 位虚拟寄存器：

- t0
- t1

其中，所有表达式计算结果（含函数返回值）使用 t0 存储。

转成 x86 汇编时，可以以 eax 和 edx 表示它们。

栈增长方向

从高往低增长。例，某变量地址为 a，下一变量地址可能是 a - n。

符号关联

```
@ begin of extlink
${body}
@ end of extlink
```

该区域用于导入导出符号，相当于 8086 汇编里的 extern。

import 从外部导入

```
import ${symbol}
```

相当于 8086 汇编的 extern。

export 导出

```
export ${symbol} ${"fun" : "var"}
```

相当于 8086 汇编的 global。

注意，这里会登记全局变量，但不会登记函数。

全局数据

```
@ begin of static-data
${body}
@ end of static-data
```

字符串

```
str ${access} ${value name} "${...}"
```

其中，转义符号保留。不显式表示尾 0。value name 不含引号，需遵循变量命名规则。

access：

- val：不可变
- var：可变

例：

```
str val sayhi "hello world!\n" // const char* sayhi = "hello world\n";
```

整形

```
int ${access} ${value name} ${length} ${value}
```

其中，length 可选值如下：

内容	备注
s8	signed int 8
u8	unsigned int 8
s16	signed int 16
u16	unsigned int 16
s32	signed int 32
u32	unsigned int 32

value 为值，以 10 进制形式表示。允许有前缀负号。

例：

```
int var year s32 2022 // int year = 2022;
```

全文符号表

该表为后续代码区域提供辅助说明信息。全文不是全局的意思，该表记录所有可能用到的符号，包含那些并没有暴露给全局的符号。

```
@ begin of global-symtab
${body}
@ end of global-symtab
```

函数定义

```
fun ${visibility} ${name} ${argc} ${return-type}
    ${...args}
```

`${name}` 指定函数名。`${argc}` 指定参数个数。`${...args}` 表示参数表。

`${visibility}`:

- internal
- visible

参数表由多个三元式组成，可以写在多行。

三元式结构：

```
${type} ${ "value" | "ptr" } ${name}
```

${type} 和函数返回类型 ${return-type} 可选的值：

- u8
- s8
- u16
- s16
- u32
- s32

结构体定义

暂不支持

变量符号说明表

代码指令中的栈上变量以一个 id 记录。通过 id 在**变量符号说明表**内寻找其详细信息。

详细信息包括：

- 符号 id
- 符号名
- 符号类型
- 字节宽度

描述格式：

```
var ${id} ${name} ${type} ${bytes}
```

块符号表

块符号表可以用来辅助栈上内存分配，和判断变量作用范围（虽然这个范围在 ir 内可能很难判断）。

```
@ begin of block-symtab

% begin
${tab}
% end

% begin
```

```
${tab}
% end
...
@ end of block-symtab
```

表 id

```
tab-id ${id}
```

标明本符号表的 id。

父表 id

```
parent-tab-id ${id}
```

标明本符号表的父级表的 id。如果该表已经是祖先，则父 id 与自己的 id 相同。

符号定义

```
var ${id} ${name} ${type} ${bytes}
```

指令概述

指令区通过以下形式划出：

```
@ begin of instructions
${instructions}
@ end of instructions
```

值表示

对于常量，直接书写值。允许写负数。

对于变量，写其在**变量符号说明表**内的编号。

对于虚拟寄存器，写其后缀编号。

值种类：

- imm: 数字（立即数）
- val: 变量。数字表示变量 id，名字表示全局变量。
- vreg: 虚拟寄存器
- fval: 函数参数

访存方式：

- 不加任何标记：直接
- **暂不支持偏移寻址**

后续所有 ${value} 都遵循此规则。

例：

```
mov vreg 0 imd 2  // t0 = 2
```

指令

标签与跳转

标签定义

```
label ${name}
```

建议当成汇编里的标签看。

如果这个标签表示一个函数，在 ir 内不用考虑寄存器保护问题。后续转汇编时，结合全局函数表，判断标签是否为函数，再决定是否要保护寄存器。

直接跳转

```
jmp ${label-name}
```

条件跳转

```
j${condition} ${label-name}
```

condition 为跳转条件，根据 t0 中的值判断。

可用的条件：

```
jge: x >= 0
jg: x > 0
je: x == 0
jl: x < 0
jle: x <= 0
jne: x != 0
```

函数调用跳转

```
call ${label-name}
```

不负责处理调用栈创建与清理等事情。

需要负责完成部分寄存器保护等工作。

函数调用返回

```
ret
```

不负责处理调用栈创建与清理等事情。

需要完成部分寄存器恢复等工作。

栈操作

push 压栈

push ${bytes} ${value}

bytes: 1, 2, 4

value：常数、虚拟寄存器、变量、全局常量

pushfc 函数调用压栈

pushfc ${bytes} ${value}

pop 弹出

pop ${bytes} ${value}

pop ${bytes} // 空弹出

popfc 函数调用弹出

popfc ${bytes}

存取

mov

```
mov ${container} ${value}
```

注意，这与汇编中的 mov 是很不同的。汇编中很难直接在内存之间移动，但 tcir 允许这么做，毕竟这只是中间代码。

lea

```
lea ${container} ${value}
```

加载地址。

算术

add

```
add ${value1} ${value2}  // value1 = value1 + value2
```

sub

```
sub ${value1} ${value2}  // value1 = value1 - value2
```

neg

```
neg ${value1}  // value1 = -value1
```

乘法和除法

暂不支持

and

```
and ${value1} ${value2}  // value1 = value1 & value2
or
or ${value1} ${value2}  // value1 = value1 | value2
xor
xor ${value1} ${value2}  // value1 = value1 ^ value2
not
not ${value}  // value = !value
```

比较

```
cmp
cmp ${value1} ${value2} ${true-condition}
${true-condition}:
ge - greater or equal
le - lesser or equal
eq - equal
ne - not equal
g
l
```

比较 value1 和 value2，t0 被设为是否满足 true-condition。

其他

```
xchg
xchg ${value1} ${value2}  // swap value1, value2
```

参考

- [1] 王爽. 汇编语言（第 3 版）. 清华大学出版社，2013
- [2] LLVM. LLVM Tutorial. https://llvm.org/docs/tutorial/
- [3] Evian Zhang. llvm ir tutorial. https://github.com/Evian-Zhang/llvm-ir-tutorial
- [4] 踌躇月光. 操作系统实现. bilibili
- [5] ZingLix. 8086 汇编指令集整理. ZingLix Blog, 2018
- [6] 华为. MAPLE IR Specification. https://gitee.com/openarkcompiler/OpenArkCompiler/blob/master/doc/en/MapleIRDesign.md
- [7] 陈火旺等. 程序设计语言编译原理（第 3 版）. 国防工业出版社，2000
- [8] 同济大学计算机系. 第七章 语义分析和中间代码产生. 同济大学计算机系