Lab. 2 二代编译器实验说明和要求

一、二代编译器功能描述

二代编译器将一种语法类似 C 语言的语句序列翻译为等价的汇编程序，所输出的汇编程序符合 X86 汇编语言格式要求，可以在 Linux 环境下正常运行。与 lab 1 不同的是，lab 1 提供了一个汇编的框架，用于管理程序执行入口点、函数栈帧等，这次二代编译器需要自行生成相关汇编代码。词法分析部分，可以使用类似 Flex 的工具实现，也可以手工实现。

二、二代编译器文法要求与语句示例

二代编译器能够处理的文法如下所示。

关键字： `int, return, main`

标识符 [1]： 符合 C89 标准的标识符（[A-Za-z_][0-9A-Za-z_]*）

常量： 十进制整型，如 `1`、`223`、`10` 等

赋值操作符： `=`

运算符 [2]： `+ - * / % < <= > >= == != & | ^`

标点符号： `; { } ( )`

语句：

变量声明（单变量且无初始化）	`int a;`
简单表达式赋值语句	`a = b&1;`
复杂表达式[3]赋值语句（仅限等级二）	`a = (d+b&1)/(e!=3^b/c&&d);`
`return` 语句 [4]	`return 0;`
函数调用[4]（只需支持预置函数）	`println_int(a);`

1. 具体标准可参考 C89/C90 standard (ISO/IEC 9899:1990) 中 3.1.2 Identifiers 章节。
2. &为按位与，|为按位或，^为按位异或。<等大小比较操作符，若为真则运算结果为 1，否则 0。这些操作符都是二元操作符。
3. 操作符优先级与 C 语言相同（与 C89 标准相同）。
4. 参数为常数或变量，不需要考虑参数为表达式的情况。

函数定义：只需支持定义 main 函数即可

不带参数	`int main(){...}`
带参数	`int main(int argc, int argv){...}`

预置函数：只需支持对预置函数的调用即可。`println_int(int a)`与 C 语言中 `printf("%d\n", a)`有相同输出。

三、二代编译器输入输出样例

测试用例难度同 lab 1 一样分为两个等级：其中等级一用例里，每个表达式中只含有一个二元操作符，且无括号；等级二用例里，同一个表达式中会有多个二元操作符，可能具有不同优先级，且包含可能有嵌套的括号[5]。测试用例中，第一个等级测试用例占比 90%，第二个等级的测试用例占比 10%。

输入测试用例文件中 Token 之间可能没有分隔的字符，也可能存在多个连续的空格或者回车作为分隔。

评分依据：提交的编译器生成的汇编码，在形成并运行二进制可执行文件后，打印出的值是否符合预期。

输入样例：

```
int main() {
    int a;
    int b;
    int c;
    a = 1;
    println_int(a);
    b = 2;
    println_int(b);
    c = 114514;
    println_int(c);
    return 0;
}
```

5. 与 lab 1 是类似的。

输出样例：

```
    .intel_syntax noprefix # 使用 Intel 语法
    .global main           # 声明 main 函数为全局符号，这使得链接器能够识别程序的入口点。
    .extern printf         # 声明外部函数 printf，表示该函数在其他地方定义，通常是 C 标准库中。

    .data                  # 开始数据段，用于定义程序中的初始化数据。
    format_str:
      .asciz "%d\n"        # 定义一个用于 printf 的格式字符串，输出整数并换行。

    .text                  # 开始代码段，包含程序的实际指令。
    main:
      push ebp             # 将基指针寄存器 ebp 的当前值压入堆栈，保存上一个函数栈帧的基指针
      mov ebp, esp         # 将栈指针 esp 的值复制到基指针 ebp ，设置新的栈帧基指针
      sub esp, 0x100       # 从栈指针 esp 减去 256 字节，为局部变量分配固定大小的栈空间
      # a = 1
      mov eax, 1
      mov DWORD PTR [ebp-4], eax

      push DWORD PTR [ebp-4]    # 准备 printf 的参数
      push offset format_str
      call printf               # 调用 printf 函数
      add esp, 8                # 清理栈

      # b = 2
      mov eax, 2
      mov DWORD PTR [ebp-8], eax

      push DWORD PTR [ebp-8]
      push offset format_str
      call printf
      add esp, 8

      # c = 114514
      mov eax, 114514
      mov DWORD PTR [ebp-12], eax

      push DWORD PTR [ebp-12]
      push offset format_str
      call printf
```

```
add esp, 8

mov eax, 0    # 准备返回值
# 退出 main
leave
ret
```

打印结果样例：

1
2
114514

四、二代编译器实现参考

二代编译器可以使用 Flex 进行词法分析，也可以选择手工生成方式，然后生成 **X86** 代码。

1. 对 println_int 的函数调用

假设有一个预定义的函数 println_int(int)，功能是将整数参数的值打印出来。

X86

利用 C 标准库中的函数 printf 实现。首先声明外部函数 printf，再在数据段定义格式化字符串。

```
.extern printf   # 声明外部函数，表示该函数已在别处定义，通常是 C 标准库
.data            # 开始数据段，用于定义程序中的初始化数据。
format_str:      # 定义一个用于 printf 的格式字符串，输出整数并换行。
  .asciz "%d\n"
```

在需要调用 println_int 函数时，转化为对 printf 的调用。首先对这种情况下的 printf 的两个参数进行准备（参数压栈），然后调用，最终恢复压入参数的栈帧。

```
push DWORD PTR [ebp-8]   # 按顺序将参数压栈
push offset format_str
call printf              # 调用 printf
add esp, 8               # 恢复栈指针
```

五、二代编译器提交要求

实现语言：C++（语言标准 C++14）

编译环境：g++-11，cmake

测试环境：Ubuntu 22.04，gcc-11

提交内容：所有编译 cmake 工程需要的文件，如.cpp, .h, .l, CMakeLists.txt 源文件等；不需要提交 build 目录。

输入输出：实现的编译器有一个命令行参数，用于指明输入文件路径，编译器从该路径读取源码，并向 stdout 输出编译结果。

希冀平台提交方式：注册希冀平台 GitLab 帐号，创建 git 仓库，以仓库链接的方式提交。基本提交格式如下：

https://gitlab.eduxiji.net/myuser/myproj.git --branch=mybranchname

https://gitlab.eduxiji.net/myuser/myproj.git 为仓库地址

--branch= mybranchname 指定分支

当你提交时，应该将 myuser 替换为你的希冀平台 GitLab 账号名称，myproj 替换为你创建的 git 仓库名称，mybranchname 替换为你创建的分支名称。

注：为防止个人源码泄露，需要将创建的 git 仓库设置为 private（私有）。

详情可查看附件《希冀平台 gitlab 简易使用参考》。

注：g++用于编译你提交的编译器实验源码，gcc 用于将你的编译器实验输出的 x86 汇编码编译成可执行文件，用于测试。

gcc 使用的编译选项为 `-m32 -no-pie`。

CMake 工程文件相关说明

你会收到收到一个含有 CMakeLists.txt 等文件的工程框架。CMakeLists.txt 的文件的内容类似于下列内容：

```
cmake_minimum_required(VERSION 3.16)
project(lab02)
set(CMAKE_CXX_STANDARD 14)
add_compile_options(-pedantic)
# add_compile_options(-fsanitize=address)
# add_link_options(-fsanitize=address)
add_executable(Compilerlab2
    ......
)
target_compile_features(Compilerlab2 PRIVATE cxx_std_14)
```

`add_executable` 的目标必须是 Compilerlab2，这就是编译得到的可执行文件

的名字，评测系统会直接运行这个可执行文件进行评测。每有一个自行编写的.cpp 源文件，都需要将其加入到 add_executable 中省略号所在位置，如下：

```
add_executable(Compilerlab2
    main.cpp
    parser.cpp
    utils.cpp
    asm_writer.cpp
)
```

.h 头文件不需要添加到这里，编译器在编译时会在与.cpp 源文件相同的目录下自动查找头文件。

如果使用 Flex 实现词法分析，则需要新建一个文件 lexer.l，在其中编写 Flex 词法规则，并修改 CMakeLists.txt 文件如下，并同样也在省略号处添加自行编写的.cpp 源文件：

```
cmake_minimum_required(VERSION 3.16)
project(lab02)
set(CMAKE_CXX_STANDARD 14)

# 使用 cmake 的 flex 模块
include(FindFLEX)
if(FLEX_FOUND)
    message("Info: flex found!")
else()
    message("Error: flex not found!")
endif()
# 为了 flex 新增头文件搜索路径
include_directories(${CMAKE_SOURCE_DIR})
# 指定 flex 编译目标
FLEX_TARGET(MyScanner lexer.l  ${CMAKE_CURRENT_BINARY_DIR}/lexer.cpp)

add_compile_options(-pedantic)
# add_compile_options(-fsanitize=address)
# add_link_options(-fsanitize=address)
add_executable(Compilerlab2
    ......
    ${FLEX_MyScanner_OUTPUTS}
)
target_compile_features(Compilerlab2 PRIVATE cxx_std_14)
```

如需使用 cmake 进行编译，在命令行中执行以下命令即可：

```
mkdir build
```

```
cd build
cmake ..
cmake –build .
```

编译产物（你编写的编译器）即为 `build/Compilerlab2`。

VS Code、CLion 等编辑器或 IDE 可以方便地管理 CMake 工程，有需要的同学可以自行搜索相关说明。

若你想自行执行汇编代码并调试

X86:

在自行插入完代码后，在 Linux 终端中执行

```
gcc -m32 -no-pie <输入汇编文件> -o <输出可执行文件>
./<输出可执行文件>
```

即可观察到输出结果。

注：在一些机器上，你可能需要添加 `i386` 架构的包才能正确执行以上操作，参考命令如下

```
sudo dpkg --add-architecture i386
sudo apt-get update
sudo apt-get install libc6:i386 libstdc++6:i386 gcc-multilib
```

六、如何检查自己的代码

在 lab 1 中，很多同学遇到了这些问题：为什么明明在自己本机上运行程序的结果符合预期，但是在评测平台上会出错？为什么程序会奇怪地 segment fault 崩溃？

一个可能的原因是，编写的代码不完全符合 C++语言标准，出现了未定义行为，例如数组越界访问、使用未初始化的变量，导致在不同环境下有不同的运行结果，或有其它细节上的错误。可以给编译器增加 `"-pedantic"` 参数，要求编译器对不标准的代码进行告警。也可以使用 Address Sanitizer（ASAN），快速检测内存错误，用法是给编译器和链接器增加 `"-fsanitize=address"` 参数，编译后正常运行即可。

lab2 提供的 CMake 工程里已经预先启用了 `pedantic` 参数，但因为 ASAN 打印多余的字符，所以没有启用 ASAN。如果需要启用，可以参考以下 CMakeLists.txt 文件内容，添加相关编译器与链接器参数。

```
cmake_minimum_required(VERSION 3.16)
```

```
project(lab02)
......
add_compile_options(-pedantic)
add_compile_options(-fsanitize=address)   # 看我
add_link_options(-fsanitize=address)      # 看我
add_executable(Compilerlab2
......
)
```