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。词法分析和语法分析部分，可以使用类似 Flex 和 Bison 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, void,`
`if, else, while, continue, break`

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]：

一元运算符： `- ! ~`

二元运算符[3]： `+ - * / % < <= > >= == != & | ^ && ||`

标点符号： `; { } ( )`

语句：

变量声明 [4]	`int a, b=111, c=1+3;`
表达式赋值语句	`a = (d+b&1)/(e!=3^b/c&d); a = b+c;`
`return` 语句 [5]	`return a+b; return func(a);`
函数调用[5]	`println_int(a+b);`

---

1. 具体标准可参考 C89/C90 standard (ISO/IEC 9899:1990) 中 3.1.2 Identifiers 章节。
2. 操作符优先级与 C 语言相同（与C89 标准相同）。
3. &为按位与，|为按位或，^为按位异或。<等大小比较与逻辑运算符，若为真则运算结果为 1，否则 0。考虑或不考虑运算符短路都可以，我们规定所有**表达式**（除了单独一个函数调用形成的**表达式**）没有副作用， 即不会产生输入输出，也不会修改其它变量、内存的值。这意味着，不会通过运算符短路的方式产生递归调用。请注意**表达式**和**语句**的区别。
4. 可能为单变量或多变量，且可能有初始化。
5. 参数可能为常数、变量、**表达式**。

条件语句	`if ( condition[6] ) { ...[7] }`
	`if ( condition[6] ) { ...[7] } else { ...[7] }`
循环语句	`while ( condition[6] ) { ... [8]}`
循环控制语句	`continue;`
	`break;`

函数定义：

不带参数	`int func(){...}`
	`void func() {...}`
带参数	`int func(int a, int b){...}`
	`void func(int a, int b){...}`

预置函数：在自定义函数外，还需支持对预置函数的调用。

`println_int(int a)` 与 C 语言中 `printf("%d\n", a)`有相同输出

6. 条件是一个**表达式**，若结果为 0 则为假，否则为真。考虑或不考虑运算符短路都可以，我们规定这里的**表达式**（除了单独一个函数调用形成的**表达式**）没有副作用，即不会产生输入输出，也不会修改其它变量、内存的值。这意味着，不会通过运算符短路的方式产生递归调用。
7. 这里可能是 0 个、1 个或多个任意**语句**（不包括循环控制语句 continue、break）；注意多个条件语句、循环语句嵌套的情况。
8. 这里可能是 0 个、1 个或多个任意**语句**（包括循环控制语句 continue、break）；注意多个条件语句、循环语句嵌套的情况。

三、二代编译器输入输出样例

测试用例难度只有一个等级，部分用例会公开在实验平台上。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：

x86 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 i = 0;
    while (i < 5) {
        if (i % 2 == 0) {
            println_int(i);
        }
        i = i + 1;
    }
    return 0;
}
```

X86 汇编输出样例：

```
.intel_syntax noprefix
.global main
.data
format_str:
.asciz "%d\n"
.extern printf
.text
main:
  push ebp
  mov ebp, esp
  sub esp, 8
  push 0
  pop eax
  mov DWORD PTR[ebp-4], eax
.L_while_cond_1:
  mov eax, DWORD PTR[ebp-4]
  push eax
  push 5
  pop ebx
  pop eax
  cmp eax, ebx
  setl al
  movzx eax, al
  push eax
  pop eax
  cmp eax, 0
  je .L_while_end_1
  mov eax, DWORD PTR[ebp-4]
  push eax
  push 2
  pop ebx
  pop eax
  cdq
  idiv ebx
  mov eax, edx
  push eax
  push 0
  pop ebx
  pop eax
  cmp eax, ebx
  sete al
  movzx eax, al
  push eax
  pop eax
  cmp eax, 0
  je .L_if_end_1
  mov eax, DWORD PTR[ebp-4]
  push eax
  push offset format_str
  call printf
  add esp, 4
  add esp, 4
.L_if_end_1:
  mov eax, DWORD PTR[ebp-4]
  push eax
  push 1
  pop ebx
  pop eax
  add eax, ebx
  push eax
  pop eax
  mov DWORD PTR[ebp-4], eax
  jmp .L_while_cond_1
.L_while_end_1:
  push 0
  pop eax
  leave
  ret
```

打印结果样例：

0
2
4

四、二代编译器实现参考

二代编译器可以使用Flex、Bison 进行词法分析和语法分析，也可以选择手工生成方式，然后生成 x86 代码。

本次实验较为简单，条件语句 if-else、循环语句 while 和循环控制语句 continue、break 的代码生成，参考对应章节内容，或者常见编译器的输出即可。

当遇到多层嵌套的 while 循环语句时，循环控制语句 continue、break 需要正确地跳转到所属循环的特定位置。我们可以维护一个跳转 label 的栈，每进入一层循环语句，就将新的 label 入栈，break、continue 时使用栈顶的 label 进行跳转，退出循环语句后 label 出栈，这样就可以确定正确的跳转位置了。

你的二代编译器需要能够正常处理函数递归。

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, .y, CMakeLists.txt 源文件等，整理为压缩包提交。不需要提交 build、.git 等目录。

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。

注：g++用于编译你提交的编译器实验源码。若选择输出 x86 汇编，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(-fsanitize=address)
#  add_link_options(-fsanitize=address)
add_executable(Compilerlab4
    main.cpp
    utils.cpp
    codegen.cpp
)
target_compile_features(Compilerlab2  PRIVATE  cxx_std_14)
```

`add_executable` 的目标必须是 Compilerlab4，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(Compilerlab4
    main.cpp
```

```
    parser.cpp
    utils.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 和Bison 实现词法分析、语法分析，则需要新建一个文件lexer.l，在其中编写 Flex 词法规则，新建一个文件 grammar.y，在其中编写 Bison 词法分析规则；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(lab04)
set(CMAKE_CXX_STANDARD 14)
include(FindFLEX)
include(FindBISON)
if(FLEX_FOUND)
    message("Info: flex found!")
else()
    message("Error: flex not found!")
endif()
if(BISON_FOUND)
    message("Info: bison found!")
else()
    message("Error: bison not found!")
endif()
include_directories(${CMAKE_SOURCE_DIR})
include_directories(${CMAKE_BINARY_DIR})
FLEX_TARGET(MyLexer lexer.l  ${CMAKE_CURRENT_BINARY_DIR}/lexer.cpp)
BISON_TARGET(MyParser grammar.y ${CMAKE_CURRENT_BINARY_DIR}/parser.cpp)
ADD_FLEX_BISON_DEPENDENCY(MyLexer MyParser)
#  add_compile_options(-fsanitize=address)
#  add_link_options(-fsanitize=address)
add_executable(Compilerlab4
    main.cpp
    utils.cpp
    CodeGen.cpp
    ${FLEX_MyLexer_OUTPUTS}
    ${BISON_MyParser_OUTPUTS}
)
target_compile_features(Compilerlab4 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/Compilerlab4`。

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

在之前的实验中，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。

lab4 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 -g)    # 看我
add_link_options(-fsanitize=address)       # 看我
add_executable(Compilerlab4
......
)
```

七、部分用于评测的测试用例

见附件