Lab. 3 三代编译器实验说明和要求

一、三代编译器功能描述

三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 或 MIPS 汇编语言格式要求。若为 X86，可以在 Linux 环境下正常运行；若为 MIPS，可以在 spim 模拟器内正常运行。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、三代编译器文法要求与语句示例

三代编译器能够处理的文法如下所示。

关键字： `int, return, main, void`

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。
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。

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

三、三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。

MIPS 提交的编译器生成的汇编码，在 spim 模拟器里执行后，打印出的值是否符合预期。

输入样例：

```
void myprint(int a, int b, int c) {
  println_int(a);
  println_int(b);
  println_int(c);
}

int main() {
  int a = 1, b = 2;
  int c = a + b;
  myprint(a, b, c);
  return 0;
}
```

X86 汇编输出样例：

```
.intel_syntax noprefix
.global main
.global myprint
.data
format_str:
.asciz "%d\n"
.extern printf
.text

myprint:
  push ebp
  mov ebp, esp
  sub esp, 4
  mov eax, DWORD PTR[ebp+8]
  push eax
  push offset format_str
  call printf
  add esp, 4
  add esp, 4
  mov eax, DWORD PTR[ebp+12]
  push eax
  push offset format_str
  call printf
  add esp, 4
  add esp, 4
  mov eax, DWORD PTR[ebp+16]
  push eax
  push offset format_str
  call printf
  add esp, 4
  add esp, 4
  leave
  ret

main:
  push ebp
  mov ebp, esp
  sub esp, 16
  push 1
  pop eax
  mov DWORD PTR[ebp-4], eax
  push 2
  pop eax
  mov DWORD PTR[ebp-8], eax
  mov eax, DWORD PTR[ebp-4]
  push eax
  mov eax, DWORD PTR[ebp-8]
  push eax
  pop ebx
  pop eax
  add eax, ebx
  push eax
  pop eax
  mov DWORD PTR[ebp-12], eax
  mov eax, DWORD PTR[ebp-12]
  push eax
  mov eax, DWORD PTR[ebp-8]
  push eax
  mov eax, DWORD PTR[ebp-4]
  push eax
  call myprint
  add esp, 12
  push 0
  pop eax
  leave
  ret
```

MIPS 汇编输出样例：

```
.globl main
.globl myprint
.data
.text

myprint:
  addiu $sp, $sp, -4
  sw $ra, 0($sp)
  sw $fp, 4($sp)
  move $fp, $sp
  addiu $sp, $sp, -4
  lw $v0, 8($fp)
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $a0, 4($sp)
  li $v0, 1
  syscall
  li $v0, 11
  li $a0, 0x0A
  syscall
  addiu $sp, $sp, 4
  lw $v0, 12($fp)
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $a0, 4($sp)
  li $v0, 1
  syscall
  li $v0, 11
  li $a0, 0x0A
  syscall
  addiu $sp, $sp, 4
  lw $v0, 16($fp)
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $a0, 4($sp)
  li $v0, 1
  syscall
  li $v0, 11
  li $a0, 0x0A
  syscall
  addiu $sp, $sp, 4
  move $sp, $fp
  lw $fp, 4($sp)
  lw $ra, 0($sp)
  addiu $sp, $sp, 4
  jr $ra

main:
  move $fp, $sp
  addiu $sp, $sp, -16
  li $v0, 1
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $t0, 4($sp)
  sw $t0, -4($fp)
  addiu $sp, $sp, 4
  li $v0, 2
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $t0, 4($sp)
  sw $t0, -8($fp)
  addiu $sp, $sp, 4
  lw $v0, -4($fp)
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $v0, -8($fp)
  sw $v0, 0($sp)
```

```
addiu $sp, $sp, -4
lw $t1, 4($sp)
lw $t0, 8($sp)
addiu $sp, $sp, 8
add $t0, $t0, $t1
sw $t0, 0($sp)
addiu $sp, $sp, -4
lw $t0, 4($sp)
sw $t0, -12($fp)
addiu $sp, $sp, 4
lw $v0, -12($fp)
sw $v0, 0($sp)
addiu $sp, $sp, -4
lw $v0, -8($fp)
sw $v0, 0($sp)
addiu $sp, $sp, -4
lw $v0, -4($fp)
sw $v0, 0($sp)
addiu $sp, $sp, -4
jal myprint
addiu $sp, $sp, 12
li $v0, 0
sw $v0, 0($sp)
addiu $sp, $sp, -4
lw $v0, 4($sp)
addiu $sp, $sp, 4
li $v0, 10
syscall
```

打印结果样例：

1
2
3

四、三代编译器实现参考

三代编译器可以使用 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 或 MIPS 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
```

MIPS

服务	Syscall 代码	参数
print_int	1	$a0 = integer
print_string	4	$a0 = string

利用 syscall 的组合实现。可翻译为：

```
lw $a0, x   # 将值从 x 表示的某处放到$a0
li $v0, 1   # 1 表示打印 int，打印$a0 的值
syscall
# 换行需要再将“\n”打印出来：
li $v0,4
la $a0, newline
syscall
```

注意：需要在生成的 MIPS 代码前面数据段“.data”处使用“.asciiz”定义表示“\n”的 newline。

```
.data
newline: .asciiz "\n"  # 定义一个字符串，用于输出换行。
```

2. 调用规约与栈帧

调用规约是一系列关于函数如何接收参数、返回结果及管理栈帧的规则集，它确保不同代码片段能有效。其实你不必严格按照某一种现有约定进行

X86

调用方从右至左将参数入栈，然后使用 call 指令调用 被调用函数。

被调用函数保存寄存器，然后在栈上分配空间保存局部变量，最后返回时恢复寄存器。函数返回值保存在 eax 寄存器内。

一个简单的例子：

```
int add10(int a){
  int c = 10;
  c = c+a;
  return c;
}
int main(){
  int a;
  a = add10(1);
  return 0;
}
```

对应的 x86 汇编可能是

```
add10:
  push ebp          # 保存 ebp
  mov ebp, esp      # 设置 ebp
  sub esp, 4        # 为局部变量分配空间
  mov DWORD PTR [ebp-4], 10    # 初始化局部变量
  mov eax, DWORD PTR [ebp+8]   # 将参数 a 放入 eax
  mov DWORD PTR [ebp-4], ebx   # 将 a 加到 c 上
  add eax, ebx                 # c = c+a
  mov DWORD PTR [ebp-4], eax   #存储结果
  mov eax, DWORD PTR [ebp-4]   # 返回值放入 eax
  leave     # 恢复栈帧
  ret       # 返回

main:
  push ebp
  mov ebp, esp
  sub esp, 4                # 为局部变量分配空间
  mov DWORD PTR [ebp-4], 1 # 设置局部变量
  push DWORD PTR [ebp-4]   # 参数压栈
  call add10                # 调用 add10
  add esp, 4                # 清理之前压入栈的参数，恢复栈指针
  mov DWORD PTR [ebp-4], eax # 将返回值放入局部变量
  leave                     # 恢复栈帧
  ret
```

栈帧的结构如下

```
地址     内容
...
ebp+12  参数
ebp+8   参数
ebp+4   返回地址
ebp     旧 ebp
ebp-4   局部变量
ebp-8   局部变量
ebp-12  局部变量
...
```

MIPS

为简单起见，这里我们使用栈传递函数参数，而非标准做法的寄存器。调用方从右至左将参数入栈，然后使用 jal 指令调用 被调用函数。

被调用函数保存寄存器（主要是 ra 和 fp），然后在栈上分配空间保存局部变量，最后返回时恢复寄存器。函数返回值保存在 v0 寄存器内。

一个简单的例子：

```
int add10(int a){
  int c = 10;
  c = c+a;
  return c;
}
int main(){
  int a;
  a = add10(1);
  return 0;
}
```

对应的 MIPS 汇编可能是

```
add10:
  addiu $sp, $sp, -4  # 为保存寄存器分配空间
  sw $ra, 0($sp)      # 保存 ra
  sw $fp, 4($sp)      # 保存 fp
  move $fp, $sp       # 设置 fp，便于访问局部变量
  addiu $sp, $sp, -8  # 为局部变量分配空间
  li $t0, 10          # 初始化局部变量
  sw $t0, -4($fp)
  lw $v0, -4($fp)     # 将局部变量 c 入栈
```

```
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $v0, 8($fp)      # 将参数 a 入栈
  sw $v0, 0($sp)
  addiu $sp, $sp, -4
  lw $t1, 4($sp)      # 从栈中取出参数 a
  lw $t0, 8($sp)      # 从栈中取出局部变量 c
  addiu $sp, $sp, 8   # 清理栈
  add $t0, $t0, $t1   # c = c+a
  sw $t0, -4($fp)     # 存储结果
  lw $v0, -4($fp)     # 返回值放入 v0
  move $sp, $fp       # 清理局部变量
  lw $fp, 4($sp)      # 恢复 fp
  lw $ra, 0($sp)      # 恢复 ra
  addiu $sp, $sp, 4   # 清理为保存寄存器分配的空间，恢复栈指针
  jr $ra             # 返回

main:
  move $fp, $sp       # 设置 fp
  addiu $sp, $sp, -8  # 为局部变量分配空间
  li $v0, 1          # 设置参数
  sw $v0, 0($sp)      # 参数入栈
  addiu $sp, $sp, -4  # 为参数分配空间
  jal add10          # 调用 add10
  addiu $sp, $sp, 4   # 清理栈
  sw $v0, 0($sp)      # 将返回值压入栈
  addiu $sp, $sp, -4  # 为返回值分配空间
  lw $t0, 4($sp)      # 从栈中取出返回值
  sw $t0, -4($fp)     # 将返回值存入局部变量
  addiu $sp, $sp, 4   # 清理栈
  li $v0, 0          # 返回值
  li $v0, 10         # 退出
  syscall
```

栈帧的结构如下

```
...
fp+12  参数
fp+8     参数
fp+4     ra
fp       旧 fp
fp-4     局部变量
fp-8     局部变量
fp-12    局部变量
...
```

五、三代编译器提交要求

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，spim

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。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。若选择输出 MIPS 汇编，spim 用于模拟执行你的编译器实验输出的 MIPS 汇编码，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(lab03)
set(CMAKE_CXX_STANDARD 14)
# add_compile_options(-fsanitize=address)
# add_link_options(-fsanitize=address)
```

```
add_executable(Compilerlab3
    main.cpp
    utils.cpp
    CodeGen.cpp
)
target_compile_features(Compilerlab3 PRIVATE cxx_std_14)
```

add_executable 的目标必须是 Compilerlab3，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(Compilerlab3
    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(lab03)
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(Compilerlab3
    main.cpp
    utils.cpp
    CodeGen.cpp
    ${FLEX_MyLexer_OUTPUTS}
    ${BISON_MyParser_OUTPUTS}
)
target_compile_features(Compilerlab3 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/Compilerlab3`。

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
```

MIPS:

在自行插入完代码后，在终端中执行以下命令即可观察到输出结果。

```
spim -file <输入汇编文件>
```

或者使用跨平台的可视化 `mips` 模拟器 `QtSpim`。

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。

lab3 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(Compilerlab3
......
)
```

七、部分用于评测的测试用例

见附件