

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 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、三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

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。

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
```

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 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
```

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
...	
...	

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

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。gcc 使用的编译选项为 `-m32 -no-pie`。