编译原理实验

Table of Contents

第一章 编译过程理解（必须）

1.1 实验目的

理解 clang 或 GCC 的编译过程，加深对编译器的理解

1.2 实验任务

利用 clang 或者 GCC 逐步编译，分析得出结果的正确性。

1.3 实验内容

GCC 构建(Build)可执行程序的过程比较复杂，简单可理解为两步，一步是编译，把源代码编译(Compile)成目标文件，第二步是把多个目标文件链接（Link）成可执行程序。对于简单的程序，可把编译与链接合并成一步得到可执行程序。

假定一个项目（Project）包含有三个文件，头文件 add.h，源文件 add.c 和 main.c，实现一个对两个数进行加法运算，如图 1、图 2 和图 3 所示。

```
#ifndef __ADD_H__
#define __ADD_H__

int add(int a, int b);

#define ADD(a,b) ((a)+(b))

#endif
```

图 1 add.h 对应的头文件

```
#include "add.h"

int add(int a, int b)
{
    return a + b;
}
```

图 2 add.c 对应的源文件

```
#include "add.h"

int g1 = 10;
int g2;

int main()
{
    int a = 10;
    int c;

    c = add(a, g1);

    c = ADD(c, g2);

    return c;
}
```

图 3 main.c 对应的源文件

1.3.1 生成可执行程序（单一命令）

利用如图 4 所示的命令可把上述的源代码生成可执行程序 main。如果在编译的过程中错误或者警告，编译器会输出错误或警告信息。

图 4 gcc 编译 main 程序-单一命令

实际上，上述命令的编译过程包含了两步操作，第一步是编译成目标文件，第二步是通过链接生成可执行程序。可通过命令 gcc -o main main.c add.c –verbose 查看编译的过程，具体的执行过程如图 5 所示，请在报告中验证分析。

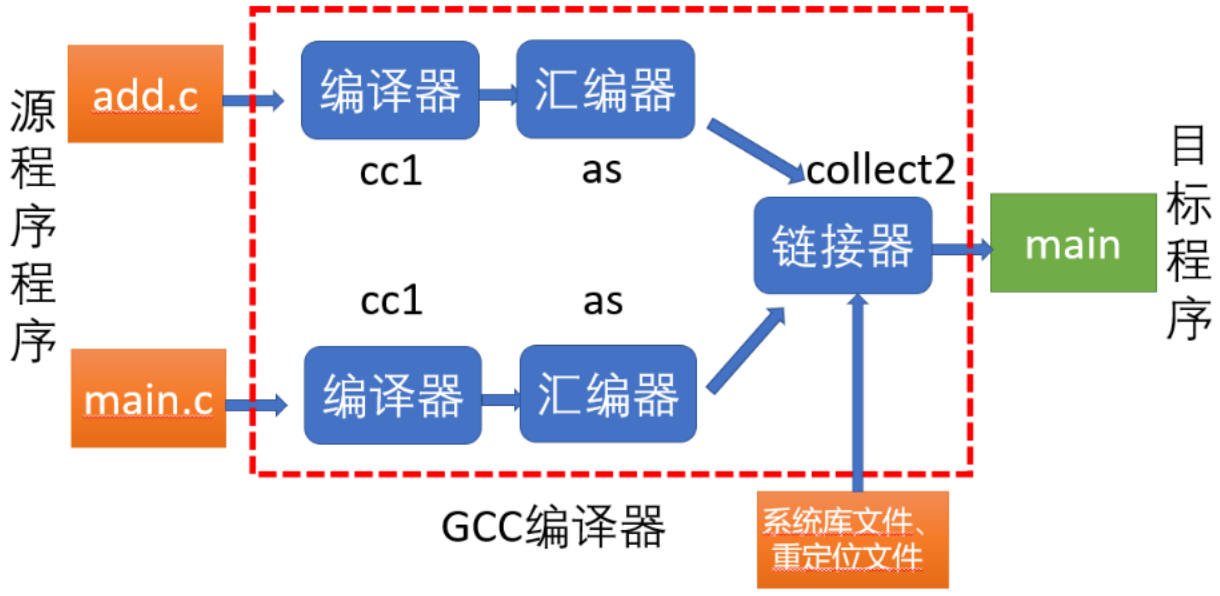


图 5 gcc 内部编译过程

1.3.2 生成可执行程序（多个命令）

利用如下三条命令可把上述的源代码生成可执行程序 main。如果在编译的过程中错误或者警告，编译器会输出错误或警告信息。

```
gcc -c -o add.o add.c
gcc -c -o main.o main.c
```

gcc -o main add.o main.o

gcc -c 命令可把源代码翻译成目标程序，简单来说，先编译成汇编代码，然后经过汇编程序翻译成重定位的目标文件，复杂来说，内部经过了词法分析、语法分析、语义分析、中间代码生成、代码优化、代码生成等生成目标文件。然后在通过 gcc -o 命令把两个目标文件链接成一个可执行程序。

这里有人会问，编译程序既然可一条指令完成，为何这里要分成三条指令完成呢？这主要是为了使得源代码编译成目标代码时可并行处理，主要用在 make 或 cmake 等 make 系统中，请在报告中利用 make 或者 cmake 进行验证，看到底能否加速编译。

图 6 是上述命令的细化，便于大家对编译器的过程理解。

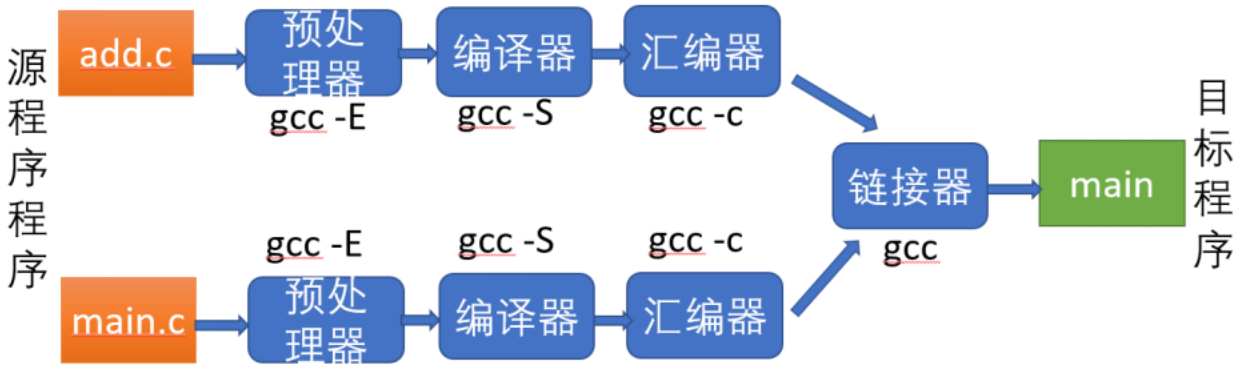


图 6 gcc 编译的详细过程

1.3.2.1 预处理器（Preprocessor）

预处理器的主要功能是处理预处理宏(C 语言中以#开头的宏)，例如把#include 所包含文件拷贝到 C 文件中，#define 宏展开，#ifdef 根据编译指定选择不同的代码等。

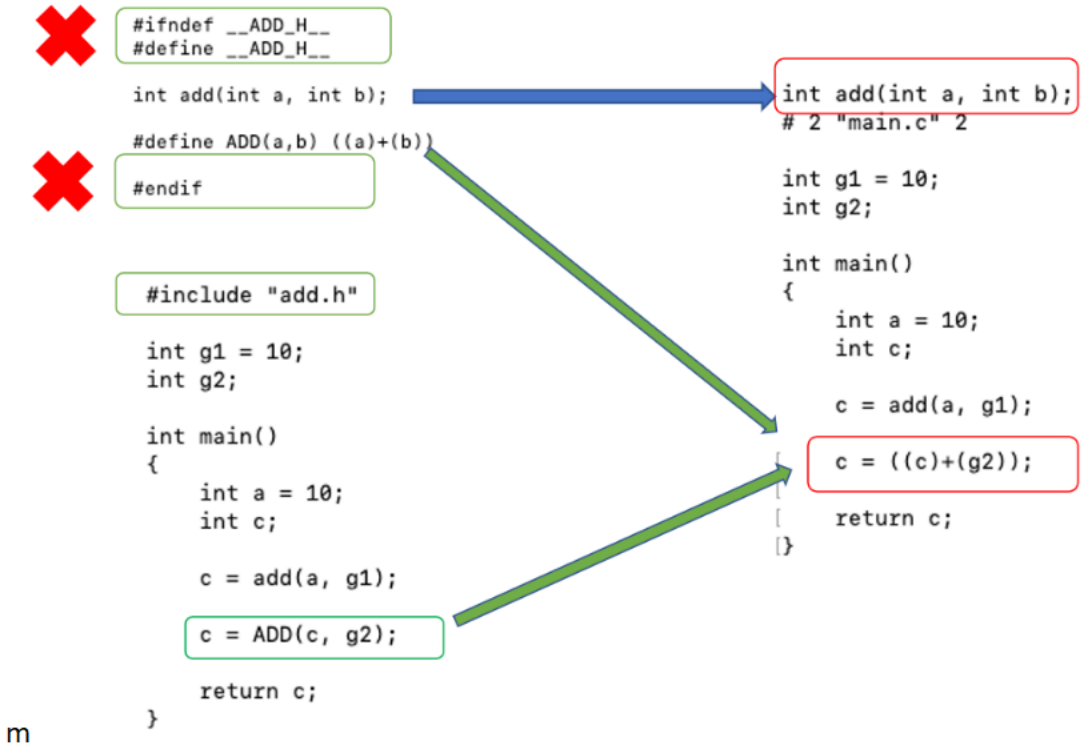
命令：gcc -E

具体命令：

gcc -E -o main.i main.c

gcc -E -o add.i add.c

图 7 gcc -E -o



m ain.i

main.c 后 main.i 与源文件的关系

从上面可以看出，main.c 中经过预处理后没有包含#include “add.h”，把头文件的内容放置到 main.i 中，ADD 宏进行了展开。

1.3.2.2 编译

编译器的主要功能是对源程序进行翻译，输出是汇编语言程序。其翻译主要包含把字符流(文本文件）形成单词流、检查是否满足 C 语言的文法要求、语义转换、代码优化、目标汇编代码生成等。

命令：gcc -S

具体命令：

gcc -S -o main.s main.i

gcc -S -o add.s add.i

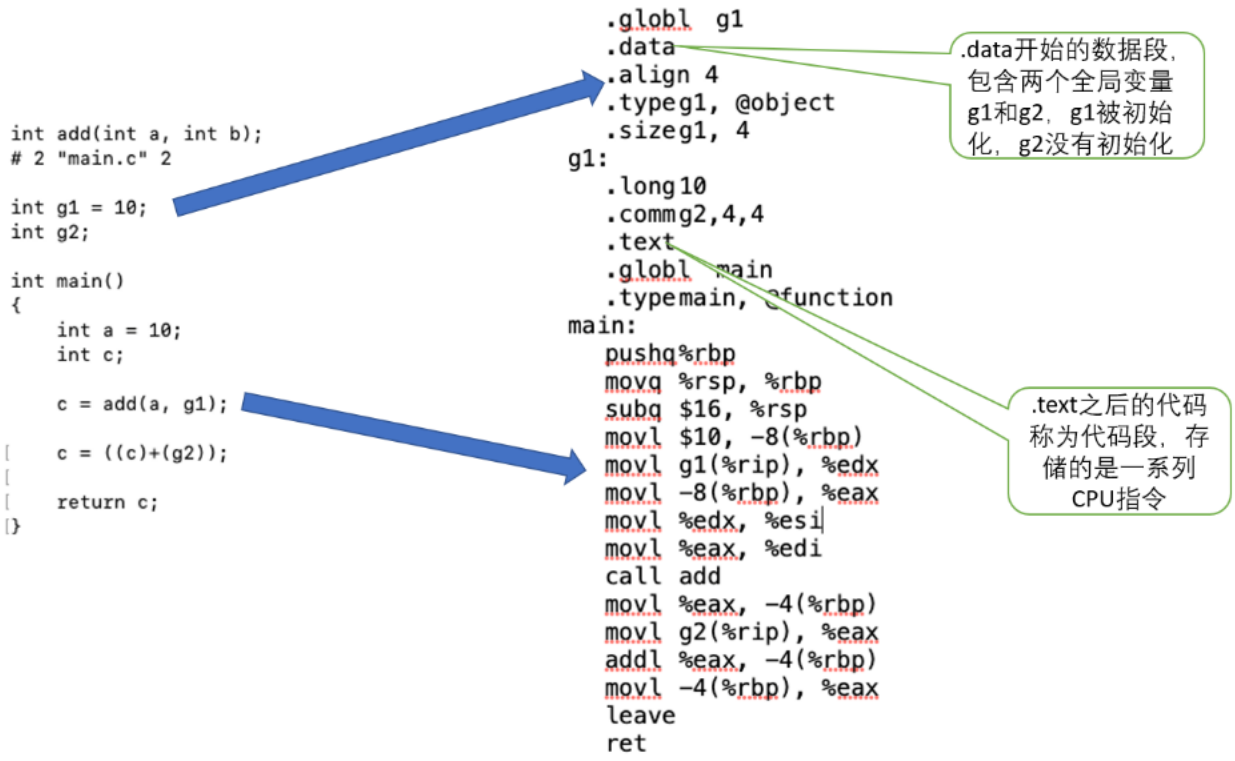


图 8 源代码与汇编的对应关系

有人可能会问，全局变量保存着在.data 中，那局部变量或者形参保存在什么地方呢？

1.3.2.3 汇编

汇编器的主要功能是对汇编源程序进行翻译，目标是可重定位的目标文件。目标文件不能通过文本编辑器查看，不过可以通过 objdump 命令分析它的内容。

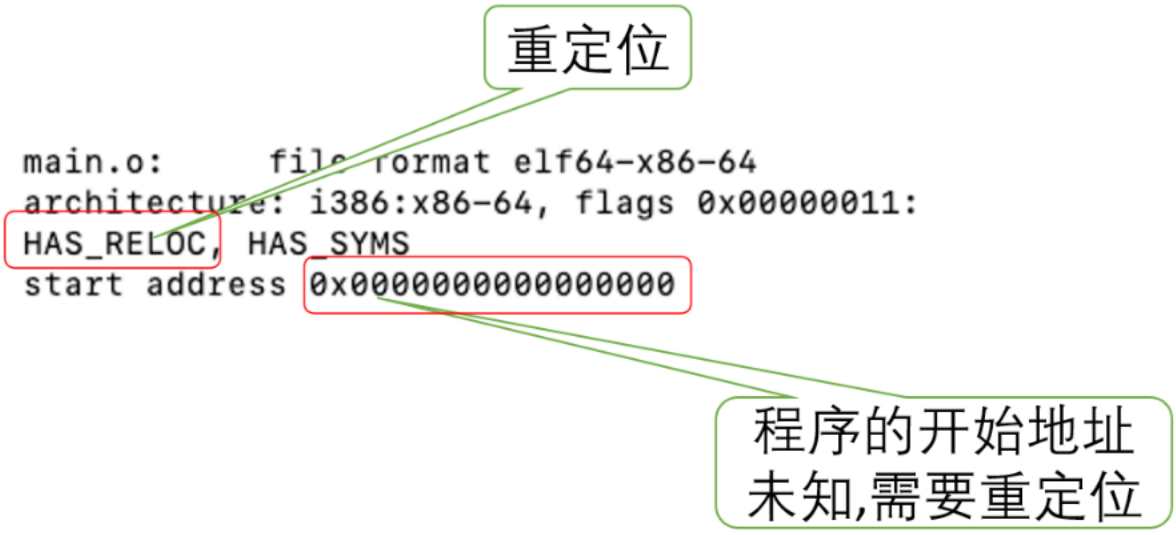
命令：gcc -c 或者 as

gcc -c -o main.o main.s

gcc -c -o add.o add.s

可重定位程序包含了很多 Section，主要包含.text、.data、.bss 等，.text 保存的是 CPU 指令，.data 保存的是初始化的全局变量或者静态变量，.bss 保存的是未初始化的全局变量或者静态变量。

objdump -f main.o 文件头信息



objdump -f main.o 的输出

objdump -x main.o 可查看目标文件中的 Section 信息

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .text         00000032  0000000000000000  0000000000000000  00000040  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000004  0000000000000000  0000000000000000  00000074  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  0000000000000000  0000000000000000  00000078  2**0
                  ALLOC
  3 .comment      00000036  0000000000000000  0000000000000000  00000078  2**0
                  CONTENTS, READONLY
  4 .note.GNU-stack 00000000  0000000000000000  0000000000000000  000000ae  2**0
                  CONTENTS, READONLY
  5 .eh_frame     00000038  0000000000000000  0000000000000000  000000b0  2**3
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, DATA
SYMBOL TABLE:
0000000000000000 l    df *ABS*  0000000000000000 main.c
0000000000000000 l    d  .text  0000000000000000 .text
0000000000000000 l    d  .data  0000000000000000 .data
0000000000000000 l    d  .bss   0000000000000000 .bss
0000000000000000 l    d  .note.GNU-stack        0000000000000000 .note.GNU-stack
0000000000000000 l    d  .eh_frame      0000000000000000 .eh_frame
0000000000000000 l    d  .comment       0000000000000000 .comment
0000000000000000 g     O .data  0000000000000004 g1
0000000000000004       O *COM*  0000000000000004 g2
0000000000000000 g     F .text  0000000000000032 main
0000000000000000         *UND*  0000000000000000 add


RELOCATION RECORDS FOR [.text]:
OFFSET           TYPE              VALUE
0000000000000011 R_X86_64_PC32     g1-0x0000000000000004
000000000000001d R_X86_64_PC32     add-0x0000000000000004
0000000000000026 R_X86_64_PC32     g2-0x0000000000000004


RELOCATION RECORDS FOR [.eh_frame]:
OFFSET           TYPE              VALUE
0000000000000020 R_X86_64_PC32     .text
```

objdump -d main.o 可查看目标文件中代码段的反汇编信息

```
Disassembly of section .text:

0000000000000000 <main>:
   0:   55                      push   %rbp
   1:   48 89 e5                mov    %rsp,%rbp
   4:   48 83 ec 10             sub    $0x10,%rsp
   8:   c7 45 f8 0a 00 00 00    movl   $0xa,-0x8(%rbp)
   f:   8b 15 00 00 00 00       mov    0x0(%rip),%edx        # 15 <main+0x15>
  15:   8b 45 f8                mov    -0x8(%rbp),%eax
  18:   89 d6                   mov    %edx,%esi
  1a:   89 c7                   mov    %eax,%edi
  1c:   e8 00 00 00 00          callq  21 <main+0x21>
  21:   89 45 fc                mov    %eax,-0x4(%rbp)
  24:   8b 05 00 00 00 00       mov    0x0(%rip),%eax        # 2a <main+0x2a>
  2a:   01 45 fc                add    %eax,-0x4(%rbp)
  2d:   8b 45 fc                mov    -0x4(%rbp),%eax
  30:   c9                      leaveq
  31:   c3                      retq
```

请注意上面的 callq 21 <main+0x21>以及 mov 0x0(%rip),%edx 指令中包含有四字节的 0，这主要是因为这些位置未定，需要在链接时重定位。

1.3.2.4 链接（Linker）

链接器的主要功能是把多个可重定位目标文件或者动态库或静态库进行符号地址值确定、多个目标文件合并成可执行程序。

命令：gcc 或者 ld 命令执行链接，若静态链接则需指定-static 选项

gcc -o main main.o add.o

objdump -f main

```
main:     file format elf64-x86-64
architecture: i386:x86-64, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00000000004003e0
```

请与 objdump -f main.o 的输出进行比对，有何不同。

objdump -x main

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
 10 .init         0000001a  0000000000400390  0000000000400390  00000390  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .text         000001b2  00000000004003e0  00000000004003e0  000003e0  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .rodata       00000004  00000000004005a0  00000000004005a0  000005a0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 24 .data         00000014  0000000000601020  0000000000601020  00001020  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 25 .bss          0000000c  0000000000601034  0000000000601034  00001034  2**2
                  ALLOC

SYMBOL TABLE:
00000000004003e0 l    d  .text  0000000000000000              .text
0000000000601030 g     O .data  0000000000000004              g1
00000000004003e0 g     F .text  000000000000002a              _start
0000000000601038 g     O .bss   0000000000000004              g2
00000000004004d6 g     F .text  0000000000000032              main
```

objdump -d main 的输出

```
00000000004004d6 <main>:
  4004d6:       55                      push   %rbp
  4004d7:       48 89 e5                mov    %rsp,%rbp
  4004da:       48 83 ec 10             sub    $0x10,%rsp
  4004de:       c7 45 f8 0a 00 00 00    movl   $0xa,-0x8(%rbp)
  4004e5:       8b 15 45 0b 20 00       mov    0x200b45(%rip),%edx        # 601030 <g1>
  4004eb:       8b 45 f8                mov    -0x8(%rbp),%eax
  4004ee:       89 d6                   mov    %edx,%esi
  4004f0:       89 c7                   mov    %eax,%edi
  4004f2:       e8 11 00 00 00          callq  400508 <add>
  4004f7:       89 45 fc                mov    %eax,-0x4(%rbp)
  4004fa:       8b 05 38 0b 20 00       mov    0x200b38(%rip),%eax        # 601038 <__TMC_END__>
  400500:       01 45 fc                add    %eax,-0x4(%rbp)
  400503:       8b 45 fc                mov    -0x4(%rbp),%eax
  400506:       c9                      leaveq
  400507:       c3                      retq

0000000000400508 <add>:
  400508:       55                      push   %rbp
  400509:       48 89 e5                mov    %rsp,%rbp
  40050c:       89 7d fc                mov    %edi,-0x4(%rbp)
  40050f:       89 75 f8                mov    %esi,-0x8(%rbp)
  400512:       8b 55 fc                mov    -0x4(%rbp),%edx
  400515:       8b 45 f8                mov    -0x8(%rbp),%eax
  400518:       01 d0                   add    %edx,%eax
  40051a:       5d                      pop    %rbp
  40051b:       c3                      retq
  40051c:       0f 1f 40 00             nopl   0x0(%rax)
```

请注意 5004e5 和 4004f2 指令相对目标文件有何不同。

1.3.3 For 循环与汇编语言对应分析

请编写含有 for 循环的 C 程序，分析 C 语言的每个句子与汇编语言的对应关系，理解编译器的语义分析功能。

1.3.4 函数调用

请编写含有函数调用（要有参数传递）的 C 程序，理解编译器在函数调用时所做的动作，并通过汇编试图去分析，验证函数的动态执行过程。

1.4 课后作业

1）函数的形式参数和局部变量保存在什么地方，编译器是如何动态分配？请在 32 位编译器和 64 位编译器上分别验证，并试图给出为何要这样做？

2）编写 Makefile 实现对多文件项目的编译和链接，并验证加速编译。若在 Windows 不可行，请在 Linux 系统上进行。

第二章 词法分析与语法分析理解与学习（必须）

2.1 实验目的

1) 学习使用词法分析程序自动构造工具 Flex
2) 学习使用语法分析程序自动构造工具 Bison/YACC
3) 熟悉 LEX 源程序语法
4) 熟悉 YACC 源程序语法
5) 熟悉递归下降分析法

2.2 实验平台

Windows 或 Linux+ Flex + Bison

2.3 实验任务

1) 以计算器为例，掌握两个工具的使用以及学会调试解决问题；
2) 强化计算器，增加对实数的支持，可进行加减乘除；
3) 采用递归下降分析法实现计算器的语法识别和 2）中所要求的功能

2.4 实验内容

2.4.1 利用两个工具与理解词法和语法功能

通过 VSCode 打开计算器的例子，掌握 flex 和 bison 工具生成 C 语言代码的方法，并通过调试理解词法分析和语法分析的过程。

2.4.2 强化计算器

2.4.2.1 词法对实数支持

在.l 文件中追加对实数的支持，如小数、科学计数法表示的实数等

2.4.2.2 语法对实数运算支持

然后在.y 文件中修改或增加对实数的加减乘除运算的支持。

2.4.2.3 整数和实数混合支持

在运算中若操作数全部都是整数，则按整数运算输出结果，其它情况则按实数运算输出实数，例如 2+2/4 等于 2，2+2.0/4 则等于 2.5。

2.4.3 递归下降分析法实现

采用递归下降分析法实现对上述功能的语法识别和语义处理，实现如上计算器的功能。该部分的功能，课后实现，提交源代码和测试报告文档。

2.5 课后作业

通过递归下降分析法实现如上要求的计算器功能。

第三章 编译前端实现（必须）

3.1 实验目的

对输入的语言进行分析分析、语法分析以及语义分析，生成中间代码语言，并产生抽象语法树。

3.2 实现任务

可选择不同的实现方式进行语法制导翻译
A）借助 flex 与 bison 工具实现，词法采用 flex，语法与语义借助 bison
B）词法人工实现，语法与语义分析借助 bison 工具实现
C）词法人工实现，语法借助递归下降分析法实现。

语义分析及中间代码生成可在生成的抽象语法树 AST 上遍历进行中间代码生成，也可边进行语法识别边进行语义分析与中间代码生成。

3.3 实验平台

Windows + Visual Studio/Visual Studio Code + Flex + Bison

3.4 实现内容

3.4.1 实验要求

1） 识别程序是否符合 MiniC 的语法要求
2） 输入 MiniC 的源文件
3） 输出程序中的各种符号定义（符号表的形式）
4） 输出程序的四元式表示
5） 输出抽象语法树，通过 Graphviz 显示

3.4.2 安装 Graphviz 工具以及对应的库

3.4.3 MiniC 编译器的语法

该文法是描述的是一个 C 语言的子集，详细见 MiniC 文法.md，或者自行定义文法产生式。

3.4.4 实现小型编译器

小型编译器程序的名字为 minic，Windows 系统下要带后缀.exe。这里假定编译器的名字为 minic，那么要实现的功能主要有：

1） 输出符号表

minic -sym MiniC 的源文件

2） 输出 IR

minic -ir MiniC 的源文件

3） 输出抽象语法树

minic -ast MiniC 的源文件

3.4.5 测试验证

根据检查点要求，为每个检查点准备至少 2 个测试用例。通过实验的小型编译器根据要求输出符号表，中间 IR 以及抽象语法树，验证编译器的正确性。

第四章 代码优化-划分基本块（必须）

4.1 实验目的

基本块是代码优化的基础，为后续的代码优化做准备。

4.2 实验任务

划分基本块，图形化显示基本块，同时可删除一些死代码
通过 Graphviz 显示控制流图

4.3 实验内容

给定 IR，划分基本块，生成控制流图并显示。