编译实验参考手册

2022.09

2022年编译实验参考手册

前端

〇、概述

0.1 本课程的总体任务

本课程作为《编译原理》实验课程，大家将会在一整个学期中，编写一个完整的编译器。为了更系统地完成这一庞大的任务（大约会有几千行代码量），编译理论上一般将编译器划分为如下五个基本阶段，分别是：

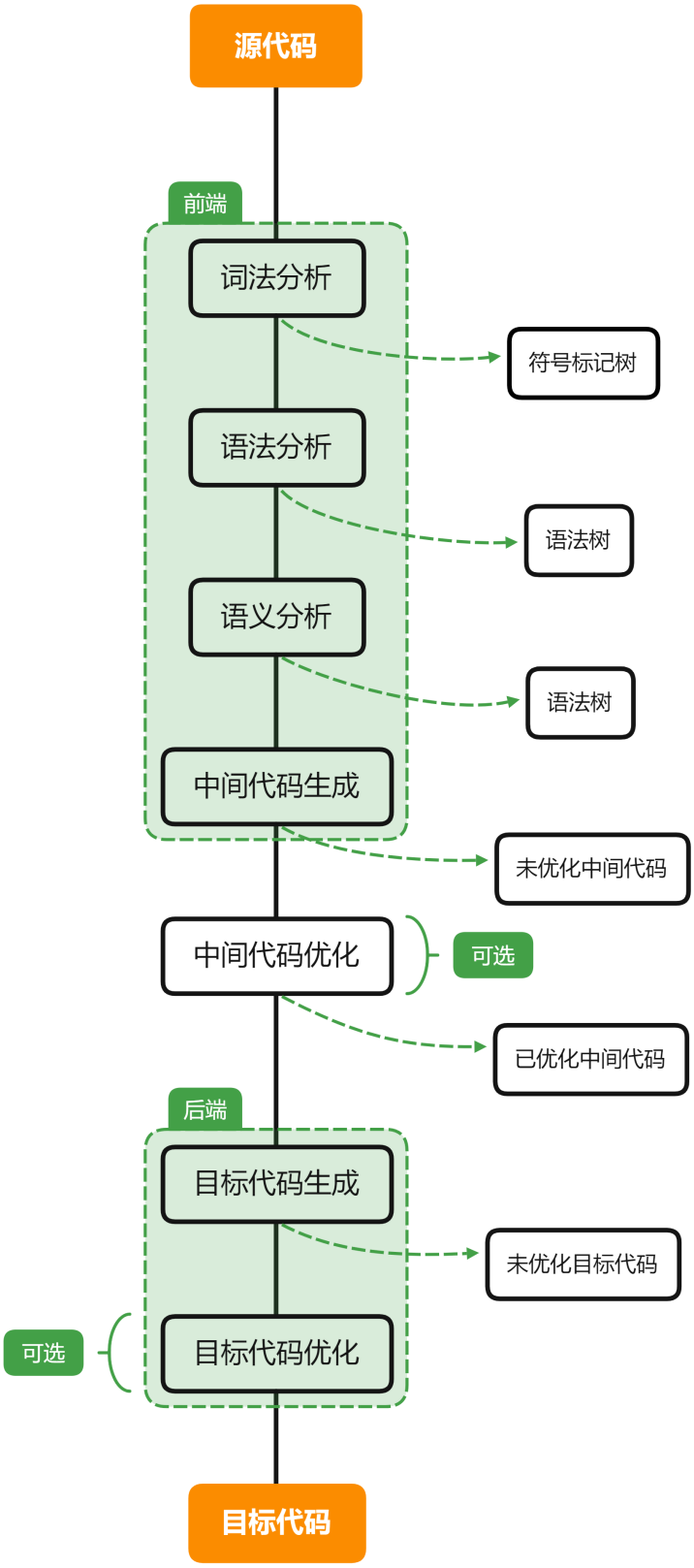
- 词法分析
- 语法分析
- 语义分析和中间代码生成
- 目标代码生成
- 代码优化

与理论划分相近但又稍有不同（如增加了文法解读、错误处理部分，后文会详细介绍），我们的实验中将这一过程拆解为如下的不同子任务：

- 文法解读
- 词法分析
- 语法分析
- 错误处理
- 代码生成1（基础功能）
- 代码生成2（全部功能）
- 竞速排序（仅MIPS分支）

每一个子任务是独立的，但又基于前一个任务的基础上迭代而来，我们对每个子任务均划分了时间期限与设置了相应评测，大家将会从每次的迭代中逐步感受到建构整个编译器大厦的过程。希望每位同学都能通过本门课程得到独属于自己的收获，祝好运！

整体流程的示意图如下（其中**语法树非必须生成**，可以用其他数据结构代替）：



0.2 编译器的语言与包含功能

我们将要编写的编译器的源语言（输入语言、待翻译语言）为SysY语言。SysY语言是一种总体上与C非常相似的语言，但包含的语法成分不及C语言多，仅有其部分功能，且在输入输出语句上的表达与C语言有少许差异。关于文法的详细解释说明参见《编译器文法》手册，不在此处赘述。

SysY 语言支持的功能包括但不限于：

- 常量变量定义与使用
- 一维二维数组的定义与使用
- 条件与循环
- 函数调用
- 整数输入
- 字符串和整数的输出

下面给出一个SysY语言的示例：

```
const int c2 = 1;
int foo(int a, int b) {
  return a + b;
}

int main() {
  int v1 = 0, v2 = 1;
  int res;
  if (v1 == 0 && v2 <= 1) {
    res = foo(v1, v2) + 1;
  }
  printf("%d\n", res);
  return 0;
}
```

我们最终完成的编译器应当能够将接收示例中的代码段作为输入，走完整个编译流程，最终将其编译成不同的目标语言（PCODE、LLVM或MIPS）。当然，对于任意符合 SysY 语言文法的其它代码片段，我们的编译器也应当能够对其进行编译处理。

此外，我们完成的编译器还应当兼具简单的错误处理功能，如对于缺少匹配括号、常量被修改等几种错误情况能够检测到并给予提示。为了降低编译器的开发难度，我们只需考虑课程组列出的几种错误类型，具体说明将会在“错误处理”一节作业中涉及与处理。理论上来说，不存在错误的源代码会被正确编译为目标代码；存在错误的源代码则需要在编译过程中做出识别判断，并输出错误信息。

0.3 课程考核与支线任务的选择

客观上来说，完成编译器开发这一庞大的任务是很有挑战性的，也会牵涉相当多的时间精力。同时，考虑到同学们不同的倾向与意愿，我们设置了多种不同难度的“支线任务”以供选择，根据生成的目标代码，分为以下三种类型：

- 生成Pcode
- 生成LLVM IR中间代码
- 生成mips目标代码+代码优化

以上三种类别的分化从“中间代码生成”一节作业开始，而之前的词法分析、语法分析阶段是共通的，并无区别。总的来说，我们认为三者难度依次递增，考虑到大多数同学应该之前并未接触过LLVM相关的内容，因此选择LLVM的一大挑战在于需要自行查找资料并学习相关语法，我们也撰写了一份关于LLVM语法说明供参考。需要注意的是，选择前两个分支的任务并无“竞速优化”一说，只需保证生成目标代码的正确性即可，因此难度相对低于需要进行优化的mips分

支。~~鲁迅曾经说过，优化是编译器的灵魂~~。我们鼓励学有余力的同学尝试**生成mips目标代码+代码优化**任务。

当然，不同的难度选择也会反映在分数上，具体**请以教学平台上发布的方案与细则为准**。

以下详细举例给出每种分支的最终输入-输出内容，帮助大家有一个宏观上粗浅的了解。需要注意的是，本节给出的代码块仅为**示意**，未包含全部文法内容。每一种输出对应的输入均为以下代码块：

```
const int c1 = 2;
const int c2 = 1;
int foo(int a, int b) {
  return a + b;
}

int func() {
  return c1 * c2;
}

int main() {
  int v1 = 0, v2 = 1;
  int res;
  if (v1 == 0 && v2 <= 1) {
    res = foo(v1, v2) + 1;
  } else {
    res = func();
  }
  printf("%d\n", res);
  return 0;
}
```

0.2.1 PCODE

Pcode似乎得名于Pascal 编译器的中间代码，或者可以理解为“伪代码”的英文全拼pseudo code的缩写。具体来说，它指的是一种栈式抽象机上的代码，与Java字节码类似，其形式可以参考PL/0编译器或PASCAL-S编译器中的目标码进行设计。

或者也可以在上述基础上做一定修改，自行设计定义一套中间代码，用来表示程序的语义逻辑含义，最终能够在虚拟机上正确完成计算，直接输出符合规范的程序运行结果即可。具体的存储、计算等执行的细节部分可参考PL/0编译器或PASCAL-S编译器的解释执行程序，若Pcode有改动，相应的解释执行程序也需要调整。

例如，对上述输入，首先应当能够产生一段自行设计的Pcode代码（但无需输出）：

```
（一段自行设计的Pcode代码，表达原程序含义）
```

最终，需要输出源代码编译产生的Pcode解释执行的结果：

```
2
```

0.2.2 LLVM IR

LLVM项目是一个模块化的、可重用的编译器和工具链集合，目的是提供一个现代的、基于SSA的、能够支持任意静态和动态编译的编程语言的编译策略。而LLVM采用的LLVM IR，作为一种经典的中间代码格式，被大多数现代语言所采用，有一整套工具链体系。

需要注意的是，由于我们没有在之前的课程中系统学习过LLVM IR（相比于MIPS），因此关于LLVM的定义与语法部分需要自行查找资料学习，或者参考我们提供的LLVM IR指导手册，这也是选择这一分支的其中一个难点所在。当然，收获与挑战将同时并存。需要特别注意的是，我们产生的LLVM IR可以直接用LLVM系列工具得到运行结果，我们的平台也提供了结果评测接口，可以提交LLVM IR代码直接进行测试，也就是说我们并不需要完成目标代码生成。

其形式如下（此处可能会有些晦涩难懂，只是给大家一个直观的例子以供了解。**注意！此处IR结果与所给示例源代码之间并不对应**）：

```
; 所有的全局变量都以 @ 为前缀，后面的 global 关键字表明了它是一个全局变量
@a = global i32 5 ; 注意，@a 的类型是 i32* ，后面会详细说明

; 函数定义以 `define` 开头，i32 标明了函数的返回类型
; 其中 `foo`是函数的名字，`@` 是其前缀
; 函数参数（i32 %0, i32 %1）分别标明了其第一、第二个参数的类型以及他们的名字
; 以下函数第一个参数的名字是 %0，类型是 i32；第二个参数的名字是 %1，类型是 i32
define i32 @foo(i32 %0, i32 %1)  {
  ; 以 % 开头的符号表示虚拟寄存器
  ; 可以把它当作一个临时变量（与全局变量相区分），或称之为临时寄存器
  %3 = alloca i32 ; 为 %3 分配空间，其大小与一个 i32 类型的大小相同。%3 类型即为 i32*
  %4 = alloca i32 ; 同理，%4 类型为 i32*

  store i32 %0, i32* %3 ; 将 %0（i32）存入 %3（i32*）
  store i32 %1, i32* %4 ; 将 %1（i32）存入 %4（i32*）

  %5 = load i32, i32* %3 ; 从 %3（i32*）中 load 出一个值（类型为 i32），这个值的名字为 %5
  %6 = load i32, i32* %4 ; 同理，从 %4（i32*） 中 load 出一个值给 %6（i32）

  %7 = add nsw i32 %5, %6 ; 将 %5（i32） 与 %6（i32）相加，其和的名字为 %7。nsw 是 "No Signed Wrap" 的缩写，表示无符号值运算

  ret i32 %7 ; 返回 %7（i32）
}

define i32 @main() {
  ; 注意，下面出现的 %1，%2……与上面的无关，即每个函数的临时寄存器是独立的
  %1 = alloca i32
  %2 = alloca i32

  store i32 0, i32* %1
  store i32 4, i32* %2

  %3 = load i32, i32* @a
  %4 = load i32, i32* %2

  ; 调用函数 @foo ，i32 表示函数的返回值类型
```

```
  ; 第一个参数是 %3（i32），第二个参数是 %4（i32），给函数的返回值命名为 %5
  %5 = call i32 @foo(i32 %3, i32 %4)

  ret i32 %5
}
```

0.2.3 MIPS

第三种分支是我们曾经接触过的、较为熟悉的MIPS~~（虽然你现在肯定声称忘了[doge]）~~，我们最终的任务是编译器处理、得到并输出mips文本，我们提供了特定版本的Mars，可以在本地自行将mips代码粘贴其中，执行获取输出结果，用以测试mips编译结果的正确性。我们接下来以一个示例再次简单回顾一下。最开始的SysY代码翻译成MIPS目标代码后可得：

```
.data
str_0: .asciiz "\n"      # str_0 "\n"
.space 4
str_end:
.text
li $fp, 0x10040000

j func_main
nop

func_foo:
lw $s0, 0($fp)       # 加载参数a
lw $s1, 4($fp)       # 加载参数b
add $v0, $s0, $s1    # 计算 a + b，作为函数的返回值
jr $ra

func_func:
li $v0, 2          # 常量c1 * c2 = 2
jr $ra

func_main:
li $s0, 0          # var v1 0
li $s1, 1          # var v2 1

li $t0, 0
bne $s0, $t0, if_end  # if v1 == 0 then ...
li $t1, 1
bgt $s1, $t1, if_end  # if v2 <= 1 then ...

sw $s0, 12($fp)        # 传入参数 v1
sw $s1, 16($fp)        # 传入参数 v2

addi $sp, $sp, -12     # 压栈，保存现场
sw $s0, 0($sp)         # 保存现场，此处保存寄存器$s0
sw $s1, 4($sp)         # 保存现场，此处保存寄存器$s1
sw $ra, 8($sp)         # 保存现场，此处保存寄存器$ra

addi $fp, $fp, 12      # 移动帧指针
```

```
jal func_foo            # 调用函数foo()
addi $fp, $fp, -12      # 移动帧指针

lw $s0, 0($sp)          # 恢复现场
lw $s1, 4($sp)          # 恢复现场
lw $ra, 8($sp)          # 恢复现场
addi $sp, $sp, 12       # 弹栈，恢复现场

move $t2, $v0
addi $t3, $t2, 1        # 计算 res = foo(v1, v2) + 1
move $s2, $t3           # $s2寄存器分配给变量res

j else_end              # 不经过else语句块

if_end:                 # label if_end
addi $sp, $sp, -16      # 压栈，保存现场
sw $s0, 0($sp)          # 保存现场，此处保存寄存器$s0
sw $s1, 4($sp)          # 保存现场，此处保存寄存器$s1
sw $s2, 8($sp)          # 保存现场，此处保存寄存器$s2
sw $ra, 12($sp)         # 保存现场，此处保存寄存器$ra

addi $fp, $fp, 12       # 移动帧指针
jal func_func           # 调用函数func()
addi $fp, $fp, -12      # 移动帧指针

lw $s0, 0($sp)          # 恢复现场
lw $s1, 4($sp)          # 恢复现场
lw $s2, 8($sp)          # 恢复现场
lw $ra, 12($sp)         # 恢复现场
addi $sp, $sp, 16       # 弹栈，恢复现场

move $t4, $v0           # 临时寄存器$t4存储函数func()的返回值
move $s2, $t4           # $s2寄存器分配给变量res

else_end:               # label else_end
li $v0, 1
move $a0, $s2           # printf res
syscall
li $v0, 4
la $a0, str_0           # printf "\n"
syscall
li $v0, 10
syscall
```

0.4 编译器采用语言的抉择

我们的编译器可以使用C++或Java语言进行编写，二者并无优劣高下之分，可随自己喜好进行选择。其中，Java为去年新增加的语言，在更早些时，仅有C++语言一种选择，但当时的文法与现在也有较大差异。现摘录往届学生这两种语言的不同观点如下。

C++派

> 我们毫无犹豫的选择了使用C++ 实现我们的编译器。其原因之一在于我们自认为我们的 Java 水平足够低，使得我们将不得不花费超过必要的时间在使用语言实现上，这对于设计的思路也将成为一种打击。
> 事实上，我们认为对于具有一定水平，这里的一定水平对于大多数计算机学院参与这门课程实验的同学来说应该都完全可以轻易达到，语言的选择并不是一个关键的问题。我们听到了很多关于 Java 会比 C++ 易用的评论，但我们并不以为然。诚然，Java 具有反射等 C++ 还尚未支持的高级功能，但是我们应该看到，此类的高级功能在本实验的正常需求中的需要是完全可以不存在的。我们认为 C++ 提供的标准库中的容器，智能指针等最实用的功能在课程组提供的 C++11 环境中已经基本满足我们的要求，且C++ 具有的对指针进行直接的操作的功能使得我们可以更加轻松的实现一些仿真工作。更进一步的，在 C++11 中已经引入了相当关键的现代 C++的特性，充分的利用 lambda 表达式，函数或类模板等功能能够使得 C++同样相当易用。
> 总之，现代 C++ 岂会是如此不便之物！

Java派

- 之前OO课程接触过，学习成本低，但相应也缺乏挑战
- IDEA等编辑器好用、免费，维护调试方便
- Java是世界上最棒的语言！

0.5 关于上机时间与期中/期末上机考核

从第二周开始，每周有3个机时的上机时间。

编译实验课的“上机”并非OS的考核形式、OO的课上作业形式，而是自由的“答疑”和分享交流环节，会有助教给大家提供帮助。上机时间不需要签到，根据自身需要前往指定机房即可。

至于期中/期末上机考核，形式为在限定时间内前往机房上机，下载自己之前提交的编译器代码，根据题目要求，或修改编译器以适应新的文法，或根据自己的实现回答问题，这与《计算机组成》课程较为类似。

0.6 实验平台与评测环境

评测机的操作系统为Linux、软件及其版本为：Mars 4.5、Clion 2018.3.4、Codeblocks 20.03、IDEA 2018.3.6。若采用C++编写，对应的编译器版本为gcc 8.1.0，若采用Java，对应编译器版本为jdk1.8。

采用judge平台，全部任务的评测也在其上进行，示例图片为代码生成作业的评测结果：

	status	statistics
testfile1	Accepted	ALU:131, Jump:9, Branch:0, Memory:86, Other:9, rank:334.5
testfile10	Accepted	ALU:265, Jump:16, Branch:16, Memory:114, Other:53, rank:618.0
testfile11	Accepted	ALU:105, Jump:1, Branch:0, Memory:29, Other:9, rank:174.5
testfile12	Accepted	ALU:60, Jump:7, Branch:10, Memory:14, Other:7, rank:132.5
testfile13	Accepted	ALU:103, Jump:7, Branch:3, Memory:53, Other:7, rank:239.5

0.7 答疑与查重

答疑共有三种途径：在教学平台提问、微信群讨论交流或上机时间机房现场交流。我们鼓励大家优先选择在教学平台提问和交流。微信群的答疑虽然方便了同学，但也对助教的工作时间带来了很大的要求，答疑的结果也不利于分享给其他同学。

judge平台提供了讨论区论坛板块，对于不会的问题，鼓励大家互相讨论交流思路（还可以考古10年前的帖子），或与老师助教进行答疑。但**严禁抄袭！互相抄袭或对往届代码、编译大赛代码的复用均在查重范围内。**

0.8 编译器整体设计

在每一部分内容中的具体架构上，鼓励使用面向对象的设计架构，使代码耦合性更好、更加清晰易读，但同样只要能够完成特定功能，使用任意架构形式都是自由的，并不强制要求。以下给出了一个参考的架构设计方案，仅供参考。注意！其中一些步骤如SSA（静态单赋值形式，Static Single Assignment form）、消除 ϕ 节点、抽象语法树（Abstract Syntax Tree，AST）等并非必需。

编译器整体分为四个层次结构。

- 第一层的主要作用是从源程序中提取信息，主要有词法分析、语法分析两部分，在进入词法分析前进行对注释部分预处理，在语法分析的过程中对错误进行处理。
- 第二层的主要作用是将源程序信息转化成便于处理的四元式，先从语法分析中抽象出AST树，然后进行一定的常数传递与消解，并生成四元式形式的中间代码。
- 第三层的主要作用是进行代码优化，可尝试利用中间代码建立SSA，然后进行一系列优化（例如常数传播、死代码删除），对优化重构后的代码进行寄存器分配，最后生成优化后的代码。
- 第四层的主要作用是将四元式中间代码转化成mips目标代码，经过乘除优化、指令合并和缩减等，生成最终的目标代码。

0.9 关于建议

后续各章节内容除输出约定等评测时的要求外，只是笔者自己之前实现或收集到的一些同学们的想法，当然在许多方案上，我们非常鼓励自行设计实现，而不被以下方案涉及到的思路所

限。如果在实现中遇到实在无法解决的问题，希望能够在设计上有所帮助。

以下是往届同学们对本课程的建议与评价，希望对大家有所启发。

2019版指导手册：

> 设计永远大于实现，没有做好设计之前千万不要着急去码代码。在正式开始编译器的实现之前，提前做好设计是很有必要的。建议考虑好总体架构和各部分间的接口之后，再开始码代码。同时，代码中注意添加适当的注释。

往届学长1：

> 我的设计未参考任何已公开项目代码，全部架构设计来源于相关书籍和本人设计，因此在设计实现过程中也耗费了很大精力。在工程中，我深刻体会到软件模块化的优势，通过将完整的编译器功能拆解为各个模块，统一各个模块间接口，可以使得工程结构清晰，便于开发调试。在调试阶段，对中间代码、目标代码的反复阅读理解也提升了我对程序执行底层原理的理解水平。虽然最后没能实现一些高级的优化，但整体而言还是收获颇丰。

往届学长2：

> 代码优化是整个编译器设计中最有挑战性的部分。这一阶段的优化部分是在中间代码上实现的。经过反复重构中间代码，让我更深信了设计优先的原则：**用于设计的精力一定要大于编码的精力！！！**优化部分往往需要多个优化方案结合起来才能发挥出威力；寄存器分配方案无最优解

往届学长3：

> 在学期开始之初，我无论如何也想象不到自己可以用并（没）不（有）熟（用）练（过）的 c++ 写出上万行的程序，并且取得还不错的性能。因此，编译对我的代码能力和设计能力都有着极大的提升。看到编译器正确运行并且输出目标代码，还是很有成就感的。
>
> 在编写大型系统时，一个清晰的设计尤其重要。因此，在编写代码前我都会给自己列出一个步骤清单，先把架构设计想清楚，从整体结构再到每步实现时的注意事项，然后严格按照之前书写的步骤进行编译器的书写。这样做虽然一开始进度可能会慢于周围的同学，但一个清晰的思路会让后面的工作轻松不少。
>
> 除此之外，代码风格和命名也十分重要。在编译器的规模逐渐增加时，一些具体的函数实现方法可能已经变得模糊，如果能有一个清晰的命名和辅助的注释，则可以让自己迅速理解函数的作用，并决定是否调用。
>
> 总的来说，虽然中间有过很痛苦的时候，但能自己写出一个基本的编译器还是很有趣的经历 :)

往届学长4：

> 我感觉最重要的为有一点：做好设计！做好设计可以让你码代码的时间大幅缩短， BUG 出现率也会显著减少，也基本不会出现重构了

往届学长5：

> “对于编译技术课程设计，本身课程内容上，实际并不困难，仅仅只是需要时间，来应对整体工程所需的复杂性，即需要时间调试、测试。不存在‘做不到’的情况”，编译器课设并不困难。要求的文法是在 C 语言的基础上简化了很多的类 C 文法，这就极大地降低了实现的难度。

0.10 开始前–文法解读作业

在正式开始完成编译器课程设计之前，需要先对文法进行分析。对文法中每条规则所定义的语法成分进行分析，考虑其作用、限定条件，思考语法成分中各种情况的组合。在测试自己编写的编译器时，除了满足评测所需的对每条规则的覆盖、对规则内不同组合情况的覆盖外，还要考虑对各种不同语法成分之间的常见组合进行测试。

内容说明

文法解读的作业内容为：对文法中每条规则所定义的语法成分进行分析，了解其作用、限定条件、组合情况和可能产生出的句子，在此基础上，编写4-6个符合文法定义的测试程序，要求所有测试程序能覆盖所有语法规则及每条规则内的常见组合情况。

语句需尽量反映出程序定义的数据及其运算结果，以便在后续阶段测试各种语法成分翻译的正确性。请提供每个测试程序的输入数据(有<读语句>则提供，否则无需提供)、**输出数据（必须提供）**。若输入输出数据没有正确提供，评测时会报错），放到文件中。

在后续对编译器进行正确性测试阶段，可以使用本次作业中撰写的测试输入、输出与程序，因此请认真对待本次作业。当然，自己编写的样例可能不一定能够覆盖全部文法情况，或者一些较为特殊的情形难以想到，因此课程组会从所有同学提交的测试程序中筛选出一些用来建立公共测试库，便于同学们进行调试。

评测与覆盖率

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef
```

如上这条规则说明了程序可以存在（变量/常量）声明、函数定义，也可以不存在。那么就意味着在设计编译器时需要考虑不同情况在处理流程上的差别，应设计出相应的测试程序对不同情况的处理流程进行测试；如果语法成分存在，则有出现1次和出现多次的可能，这也是在设计测试程序时均需要考虑到并进行覆盖的地方。

```
语句 Stmt → ...
          | 'if' '(' Cond ')' Stmt [ 'else' Stmt ]
          ...
```

同样，如上这条规则说明了条件语句可能存在 else 分支也可能不存在，这两种情况都需要覆盖到。

这里只是列举了一些例子，其他情况还请自己仔细思考。**评测时主要根据对每条规则、对规则内不同组合情况的覆盖率进行打分**。需要注意这只是对测试程序的基本要求，建议同学们进一步考察各种语法成分的不同组合情况，以期编写出能对编译器进行更全面测试的测试程序。

文法解读样例

这里给出了一个完整的测试程序样例应当具备的格式。**注意！该样例远未覆盖文法的全部情况**，大家需要自行构造能够覆盖全部文法情况的文法解读作业样例。

- testfile.txt文件

```
const int N = 6;
int partition(int arr[], int low, int high){
    int key;
    key = arr[low];
    while (low < high) {
        while (low < high && arr[high] >= key ){
            high = high - 1;
```

```
        }
        if (low < high) {
            arr[low] = arr[high];
            low = low + 1;
        }
        while (low < high && arr[low] <= key) {
            low = low + 1;
        }
        if (low < high) {
            arr[high] = arr[low];
            high = high - 1;
        }
    }
    arr[low] = key;
    return low;
}
void quick_sort(int arr[], int start, int end){
    int pos;
    if (start < end) {
        pos = partition(arr, start, end);
        quick_sort(arr, start, pos - 1);
        quick_sort(arr, pos + 1, end);
    }
    return;
}
int main(){
    int i;
    int arr[N] = {32, 12, 7, 78, 23, 45};
    printf("before \n");
    i = 0;
    while (i < N) {
        printf("%d\t", arr[i]);
        i = i + 1;
    }
    quick_sort(arr, 0, N - 1);
    printf("\n after \n");
    i = 0;
    while (i < N) {
        printf("%d\t", arr[i]);
        i = i + 1;
    }
    printf("\n");
    return 0;
}
```

- input.txt（程序输入，空文件）

```

```

- output.txt（程序输出）

```
before
```

```
32      12      7       78      23      45
after
7       12      23      32      45      78
```

提交以上文件后，评测机会返回评测结果，告知每一个testfile样例覆盖了多少百分比的文法规则，以及全部的测试用例合计的覆盖结果。如果全部正确，你应当看到如下结果：

	Status	Coverage	Info
testfile1	RIGHT	67.0%	
testfile2	RIGHT	76.0%	
testfile3	RIGHT	63.0%	
testfile4	RIGHT	77.0%	
testfile5	RIGHT	43.0%	
Total	RIGHT	100%	
Score	100%		
Detail	恭喜你！样例已覆盖所有语法成分！		

一、词法分析

1.1 概述

词法分析部分主要任务是读取源程序代码，并将其划分成若干单词，单词为规定的保留字、字符串常量、标识符、数字、符号等。此外，还需要对注释进行处理，最终的输出结果中，将单行注释与多行注释过滤掉，仅输出有效语法成分。

词法分析总体上可分为**单词识别**和**存储**两个部分。单词识别分为单字符符号识别、双字符符号识别、保留字识别、标识符识别、字符串常量识别。按照逻辑顺序逐一进行匹配和识别即可。识别后将单词序列按照顺序存储，注意需要同时保存所在行的信息，以备后续输出和错误处理使用（或者后续再对现有架构进行修改）。

1.2 总体思路与难点分析

关于题目的具体要求与输入输出形式请以教学平台发布的作业为准，本节主要从宏观的层次介绍下总体的内容与思路，并对难点做出剖析。

由于我们的输入需要从文件中进行读取，因此如何处理IO并将输入的字符串等格式转化为词法分析所需要的格式是本次任务一个较大的难点。在这里我们给出如下的一种有限自动机思路作为参考（见1.3节），其核心做法是每次读取单个字符，并标记、设定当前状态，根据接下来读入的不同字符产生不同的结果，并跳转到新的状态，直到读取到输入文件结尾，这一方案与教材上的处理方式较为接近，但丰富了一些实现上的细节。

在难点上，主要包括以下几个问题需要处理，此处交给读者先思考一下其解决方案。

注释的处理

词法分析分的难点之一在于注释的处理，而注释又分为单行注释和多行注释，注释的命名规则与形式与C语言相同（ `//` 表示单行注释， `/* ... */` 表示多行注释）。需要注意的是，不是所有的 // 和 /*...*/ 都应该被识别为注释。若存在同一行有多个注释起始符的情况，只能有一个生效。比如以下情况中，后面的多行注释起始符应当被忽略：

```
// /* ......
```

除此之外，注释符号中的 `/` 和 `*` 分别也与除号和乘号相冲突。因此，我们设计的注释处理方案需要能够对当前符号的类别作出准确区分。可以通过对状态进行标记，或者提前“偷看”下一个字符等方式来实现。

字符串的存在也会影响注释的识别。若在字符串内出现了注释的起始符号，需要忽略它，仅仅当作字符串的一部分。例如， `"//"` 及 `"/*"` 应当被正确识别为字符串，而非注释的标志符。

以上就是一些需要注意并考虑到的常见情形。在注释的处理上，可以选择用独立的一步对注释内容进行处理：扫描一遍整个代码，对所有注释统一处理、替换。也可以只用一遍扫描同时处理注释与非注释内容，具体的实现上就看大家设计发挥了~

双字符符号识别

词法分析过程中，需要遵循最长匹配原则，因此不能每读取到一个字符就进行识别，需要考虑到各标识符的结构，尤其对于如下列出的双字符构成的词法成分而言。

- 双字符及其前缀类

!	>	<	=	&	|
!=	>=	<=	==	&&	||

可以看出，首行每个字符都是次行相应字符的前缀，因此在读取到首行的字符时，无法立即判断当前的单词种类，需要读取完后面紧接着的一个字符后，才能确定它是单个出现还是和后面紧跟着的符号一起构成一个单词，从而做到不漏读、不复读字符。

标识符处理

根据文法定义，标识符（IDENFR）指的是一个字符串，可以为程序保留关键字（如int、const）或命名标识符（例如函数名、变量名），因此在确定一个字符串的内容后需要区分开两种不同的情形，尝试匹配各个英文保留字，若均无法匹配，则认为是一个标识符。

需要注意的是，标识符中除大小写字母外，也是允许包含数字或下划线的，例如 `a0` 、`a_0` 均是一个合法的标识符类别，因此，需要将其能够与全为数字的整数字符串，或运算符字符区分开。

换行

在所有语句中，一部分如赋值语句会以分号 `;` 作为结尾，但也存在诸如大括号作为一行结尾的情况，当进入到新的一行时，需要对单行注释等状态进行重置，并结算当前读取输入的情况。

需要额外注意的是，标志一行结尾的换行符可能有 `\n` 与 `\r` 两种情况，在Windows、MacOS或Linux等不同操作系统下也会有所区别，这可能会导致在本地测试时正常运行，但提交到评测机后出现运行时的错误，因此我们建议在设计时为代码赋能对不同换行符的处理能力，避免因平台问题导致出错。

根据单行读取或多行读取等不同的文件读取方式，对应的处理方式也会存在一定区别。这里的一个小建议是可以保存好每个单词或语句的行号，后续的错误处理作业中会有要求输出错误片段对应行号的要求，为之后的任务做出一定的铺垫。

1.3 有限自动机实现

注意！以下思路仅供参考，仍需自行独立设计方案。

方案1

词法分析过程本质是根据文法所用到的终结符构建一个 **DFA（有限状态自动机）**，识别出所有的关键字及标识符。由于实现的文法关键词较少，文法相对简单，关键词长度不超过 10，实现过程中并不需要真正构建出一个 DFA，只需模拟 DFA 进行贪心匹配即可。具体来说，每个关键字 / 标识符之间可以用如下的方式来进行分隔：

- 空格、制表符、换行符。
- 字母数字下划线和其它字符相邻。

对于每个分隔出来的部分去判断其是属于哪一类，如果以数字开头则是 IntConst，否则如果和某个关键字相同则是对应关键字，否则是标识符。因为文法中支持单行注释、多行注释和字符串输出，要进行三个特判：

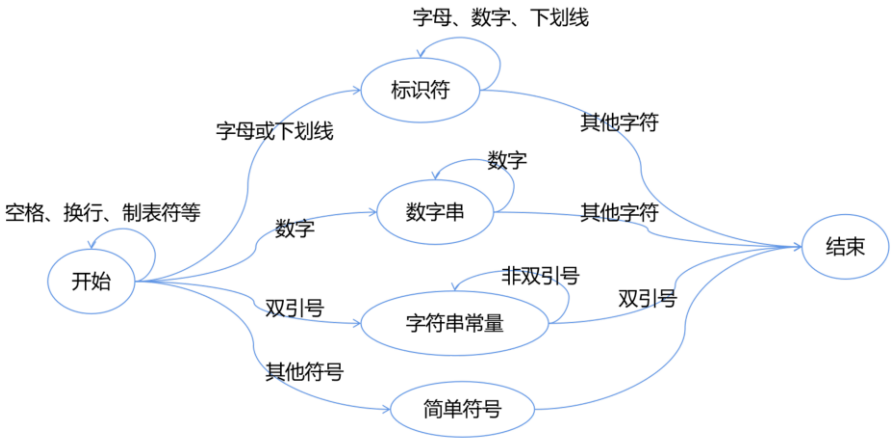
- 单行注释，从 // 开始，到换行符结束。
- 多行注释，从 /* 开始，到 */ 结束。
- FormatString，从 ” 开始，到 ” 结束。

使用三个变量标记当前是否处于上述三者内，若是则跳过即可。

此外，还有两个细节需要注意，不过无伤大雅：

- 由于很多数据是在 Windows 上生成的，需要特判 \r 字符。
- 像 = 和 == 这样的关键字，不能只读一个就判断，需要读入第二个字符后再进行判断。

总体读取状态转移图如下：



方案2

关于词法分析程序的实现，可采用**有限自动机DFA**的方式。该状态机的状态和转移规则设定如下：

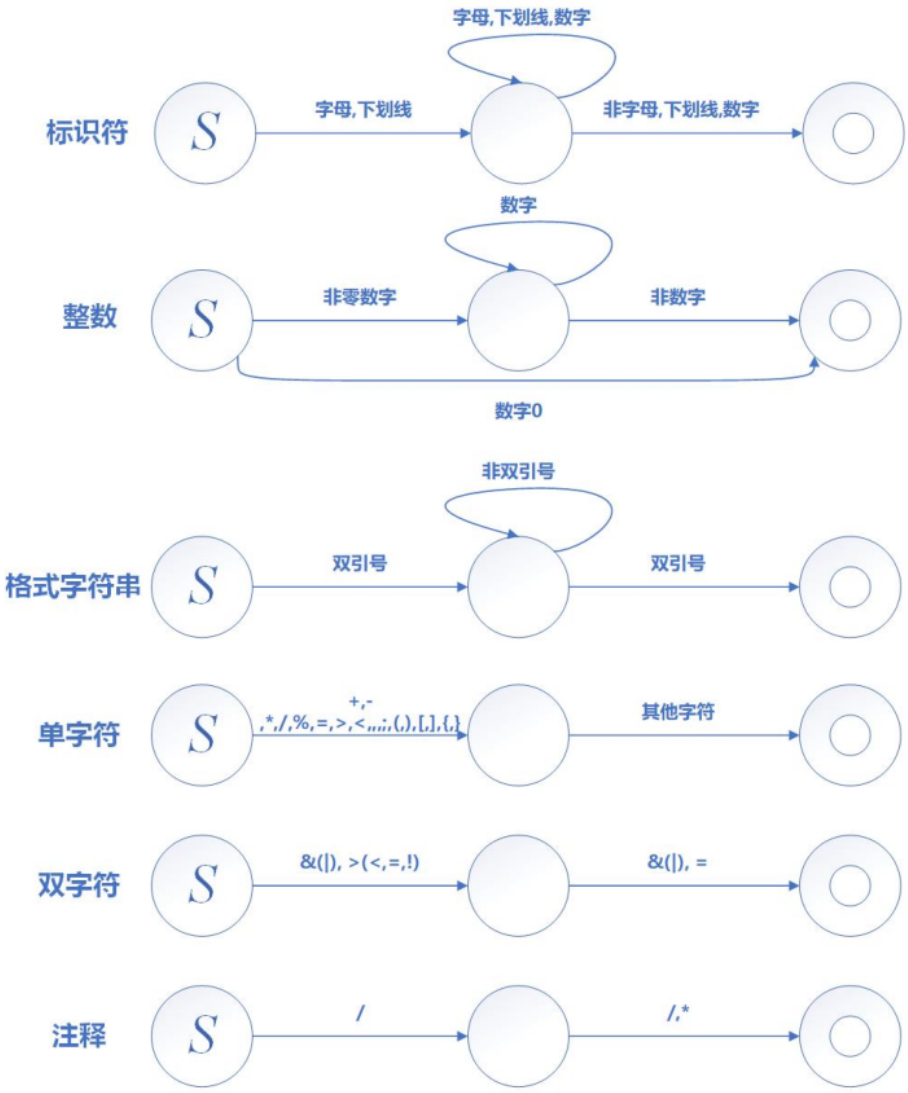
1. INITIAL：初始状态
2. IDENFR：标识符状态，当前读入字符串为标识符或关键字
3. INTCON: 整数状态，当前读入字符串为整数（INT）类型
4. STRCON: 格式字符串状态，当前读入字符串为格式字符串
5. NOTE_ONE：单行注释状态，当前读入字符串为单行注释（即’//’所在行）
6. NOTE_MULT: 多行注释状态，当前读入字符串为多行注释（即’/ *... */’ 所包含内容）
7. END：结束状态，当前读入指针移动到文件末尾

再根据每种情况设定好相应的状态转移规则，做到不重不漏，即可完成全部字符的处理。

补充说明：对于单字符分界符（即’+、−、*、/、%、>、<、=、!、 ,、 ;、(、)、[、]、{、}’）和双字符分界符（即’&&、‖、<=、>=、==、!=’），读入后直接解析处理生成类别码，不

进行状态转移；而对于标识符和关键字，可以认为关键字是标识符的一种特殊情况而把它们归为一个状态处理，由于标识符不能使用关键字，可以穷举关键字进行解析生成类别码。

总体状态转移图如下所示：



方案3

根据文档中提到的词法分析类别，将词法类型分为如下五类：

```
Ident, Num, FormatString, SeparationChar, Note
```

判断和读取方法如下：

- 如果当前字符为大小写字母或'_'，则接下来的词法成分是Ident(此时的Ident包含了保留字，后面会做出区分)，开始读取字符直到读到的字符不是大小写字母，'_'，数字中的任意一个，随后判断是否为保留字
- 如果当前字符为数字则接下来的词法成分是数值常量，开始读取字符直到读到的字符不是数字
- 如果当前字符为"则接下来的词法成分是FormatString，开始读取字符直到读到"
- 判断当前字符或者当前和下一字符构成的串，是否在自己定义的分隔符表中，如果在，则接下来的词法成分为分隔符（此时包含了注释的情况，后面会做出区分），注意到分隔符包含了除号/，而单行注释和多行注释的开头也是/，因此在读取分隔符前应判断当前和下一字符构成的字符串是否为//或/*，如果是则读取一个注释，否则读取上述分隔符

- 重复上述过程，直到结束，即可得到源代码的所有词法成分。

方案4

关于字符串的遍历与扫描，有如下方案可供参考。

①读入所有字符，得到一整个字符串。

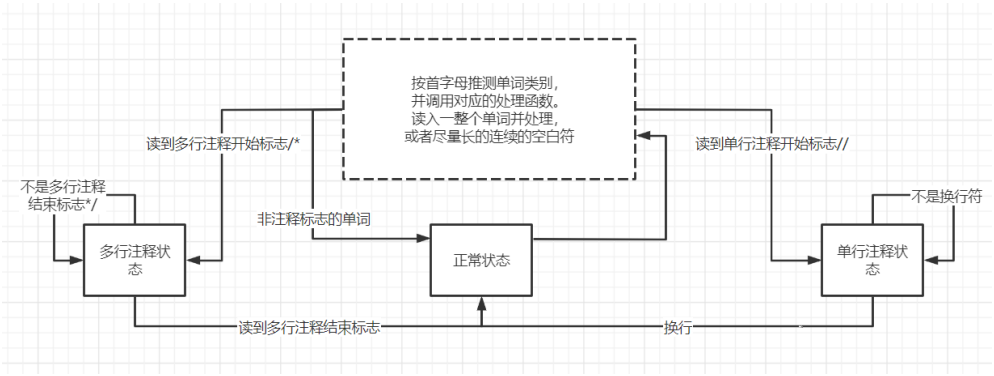
扫描输入文件中的所有字符，如果不是'\r'，就把它存入全局字符流，最后转成全局字符串。

去除\r的做法是编码完成后的更改。这样做是因为本程序要通过单一的\n来判断换行，去除\r避免判断换行时出错。windows中的换行符是\r+\n，但是在windows下运行的程序会把\r和\n合并处理，本身不会出错。但测评机的环境是Linux，此时如果测评样例是在windows中编辑得到的，那么\r+\n就会被拆成两个字符，而后续词法分析代码没有处理\r的能力，从而导致出错。

②状态+贪心扫描结合遍历字符串

词法分析本身可以写成一个大状态机，但是为了编程的简便和代码的可读性，采用了状态+贪心扫描结合的方式。

设置三个状态：正常状态，单行注释状态，多行注释状态。不同状态之间依照下图规则进行转移



1.4 输入–输出的评测约定

提交词法分析程序的源代码，要求将词法分析的结果输出到命名为 `output.txt` 的结果文件中。例如有如下程序段：

```
const int array[2] = {1,2};

int main(){
    int c;
    c = getint();
    printf("output is %d",c);
    return c;
}
```

则输出的结果文件 `output.txt` 中具有如下内容：

```
CONSTTK const
INTTK int
IDENFR array
LBRACK [
INTCON 2
```

```
RBRACK ]
ASSIGN =
LBRACE {
INTCON 1
COMMA ,
INTCON 2
RBRACE }
SEMICN ;
INTTK int
MAINTK main
LPARENT (
RPARENT )
LBRACE {
INTTK int
IDENFR c
SEMICN ;
IDENFR c
ASSIGN =
GETINTTK getint
LPARENT (
RPARENT )
SEMICN ;
PRINTFTK printf
LPARENT (
STRCON "output is %d"
COMMA ,
IDENFR c
RPARENT )
SEMICN ;
RETURNTK return
IDENFR c
SEMICN ;
RBRACE }
```

1.5 建议与感想

往届学长1：

> 词法分析可以为后面的内容提前做好准备，比如在我的设计中，词法分析同时完成了格式字符串的格式识别，方便格式字符串的错误处理和代码生成时的翻译。
> 在我的设计中，词法分析作为单独的一个过程，产生一列Token流供后续处理，有利于后面语法分析的回溯，且解耦了词法分析和语法分析，方便分别进行修改。

往届学长2：

> 在遍历上述testfile.txt拼接成的String时，随时记录当前的行数，读取到\n则当前行数加1，并在词法成分的类LexicalData.java中记录该词法成分处在的行数，方便后续错误处理时输出行数。
> 后续发现注释对后面的工作没有任何作用，可直接丢弃不予保存

二、语法分析

2.1 概述

语法分析（英语：syntactic analysis，也叫 parsing）是根据某种给定的形式文法对由单词序列（如英语单词序列）构成的输入文本进行分析并确定其语法结构的过程。它是编译过程中的核心部分，这一阶段的任务就是检查源程序是否符合规定的文法，并将源程序识别为不同的语法成分，为后续的语义分析和生成中间代码做准备。

语法分析器（parser）通常是作为编译器或解释器的组件出现的，它的作用是进行语法检查、并构建由输入的单词组成的数据结构（一般是语法分析树、抽象语法树等层次化的数据结构）。语法分析器通常使用一个独立的词法分析器从输入字符流中分离出一个个的“单词”，并将单词流作为其输入。实际开发中，语法分析器可以手工编写，也可以使用工具（半）自动生成，在本实验课程中，要求大家手工进行编写。

2.2 总体思路与难点分析

语法分析部分推荐使用递归下降分析法（一种自顶向下的分析方法）来实现语法分析。递归下降分析法指的是**为每个非终结符编写一个递归子程序**，以完成该非终结符所对应的语法成分的分析与识别任务，若正确识别，则可以退出该非终结符号的子程序，返回到上一级的子程序继续分析；若发生错误，即源程序不符合文法，则要进行相应的错误信息报告以及错误处理。程序代码的详情可以参考教材或课上内容所给出的示例。

在难点上，主要包括以下几个问题需要处理，此处交给读者先思考一下其解决方案。

语法匹配

在识别接下来的语法成分前，可选择对后续符号进行预读和判断，确定该条语句属于哪一种分支。但由于并不一定所有的文法都是 LL(1) 文法，因此有时只预读一位并没法判断该进入到哪一种语法成分的分支中去。例如，对于如下具有多个分支的Stmt文法：

```
Stmt → LVal '=' Exp ';'
     | LVal '=' 'getint''('')'';'
     ...
```

当我们刚进入Stmt时，根据当前已知信息（预读1位符号）无法判断当前语句究竟属于两种中的哪一种，只有读到 `=` 之后，观察其后的单词为字符串 `getint` 还是非终结符 `Exp` 时，才可以最终确定。而我们的语法分析程序应当想办法能够预先处理、区分开不同的分支情况。

对于这种非 LL(1) 文法，有几种解决思路：一种选择是将文法改写为LL（1）文法，但要保证改写前后文法是等价的，即要保证改写前后的文法所确定的语言是相同的，同时还需要考虑到程序输出顺序要求，但这种做法可能会对原文法有比较大的改动，具有一定挑战性；另一种方法就是继续预读，直到读到那个能够区分该进入哪个分支的单词为止；还有一种思路是设定回退机制，如果进入到了错误的分支能够返回到起始分叉位置。

文法改写

原文法中存在左递归文法，例如：

```
加减表达式 AddExp → MulExp | AddExp ('+' | '-') MulExp
关系表达式 RelExp → AddExp | RelExp ('<' | '>' | '<=' | '>=') AddExp
...
```

采用递归下降分析法时无法对其进行处理，因此，一种可行的处理方法是改写文法、消去原文法中左递归的部分，但需要尤其注意的是，应当保证文法改写前后是等价的。在输出顺序上可能也需要做出一定的调整。

2.3 递归下降分析实现

注意！以下思路仅供参考，仍需自行独立设计方案。

思路1–整体流程

方案1

首先改写文法，消除左递归，采用递归下降分析法，对文法中除去BlockItem, Decl, BType外的所有非终结符编写分析程序，并根据当前的词法成分，以及一定的超前扫描，判断接下来是什么语法成分，并调用相应的语法分析程序，在每个语法分析程序结束时输出该语法成分的名称即可完成题目要求的输出。

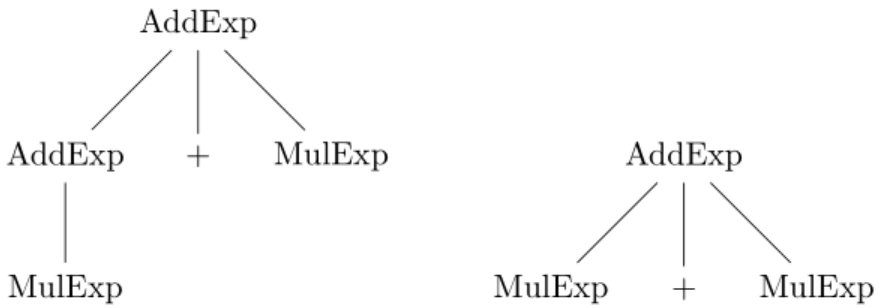
思路2–左递归文法的处理

由于原文法中存在左递归的情况，因此为了避免在语法分析中出现循环的情况，需要消除原文法中的左递归。但由于输出为根据原文法生成的语法树的后缀形式，因此在消除左递归后应适当变换，使输出满足原文法要求。

需要注意！以下部分方案对文法采取了修改，这可能会对后面生成代码造成一定的隐患，因此需要谨慎行事。

方案1

在处理中，可采用转换为扩展的 BNF 范式形式的方法予以解决，使得同种类型的连续表达式将不再通过递归推导而是通过循环直到到达边界得到。在完成分析后，还需要将相关的树结构进行重构，使得其结构符合文法的定义。其具体操作可见下方示例。



(a) 原始结果　　(b) 调整后结果

至于什么是递归下降，采用什么样的写法，可能此时也并不容易理解。当然最终的写法是因人而异的，以下给出一个仅供参考的示例，对函数的写法有一个直观的印象。举例来说，`CompilerUnit` 的文法定义如下：

```
编译单元 CompUnit → {Decl} {FuncDef} MainFuncDef
```

对CompileUnit的处理与调用如下：

```
private void CompUnit() {
    while (/* some case */) {
        Decl();
```

```
    }
    while (/* some case */) {
        FuncDef();
    }
    MainFuncDef();
}
```

深入到Decl()部分，则向底层逐步调用ConstDecl()与VarDecl，依此类推：

```
private void Decl() {
    if (symIs("const")) {
        ConstDecl();
    } else {
        VarDecl();
    }
}
```

对于部分左递归文法，如各种表达式，改写为非递归文法以避免死循环。但是将左递归文法改写为非递归文法后，按改写的文法执行递归下降，建立的语法树等价但不符合输出要求。如5个式子连乘，要求的文法需要做成高度为6的二叉树，但修改文法后可以做到两层的多叉树，这导致输出时结构与期待结构有差异，因此需要重新在树中建立左递归的结构。解决方案是在处理左递归时先按照改写文法处理，但在发现右侧还有成分时，先将左侧向上打包一层，构造出原文法所要求的左递归结构。当前，也存在其他的解决方法，同学们可以自主查找资料进行探索。

最终，程序将会输出一个语法类对象，对应的类型为 compUnit ，即抽象语法树的根节点。逐层展开则可以得到完整的推导过程。注意到消除左递归后，语法成分的输出发生了变化，以语法成分AddExp 为例，其文法改写后为 AddExp –> MulExp {('+' | '–') MulExp} ，如果按照此文法调用语法分析程序会导致少输出一些 <AddExp>。因此，分析程序应当能够处理这一问题

思路3–回溯问题

方案1

回溯问题的存在是因为对于某个非终结符号的规则其右部有多个选择，且其first集相交，从而导致分析到此语法成分时，不能仅根据当前词法成分判断接下来的语法成分是什么。

采用超前扫描，向前多看几个词法成分，直到可以确定接下来需要分析的语法成分，以函数调用和函数的定义的判断为例：

```
private boolean isFuncCall() {
  //当前词法成分是ident，并且下一词法成分为左小括号
  return nowSym().isIdent() && nextSym().isLeftParent();
}
```

此判断根据当前和下一词法成分判断接下来的语法成分是否是函数调用

```
private boolean isFuncDef() {
  return (nowSym().isInt() || nowSym().isVoid()) &&
    nextSym().isIdent() && nextTwoSym().isLeftParent();
}
```

此判断根据当前和接下来的两个词法成分判断接下来的语法成分是否是函数定义

方案2

通过尽可能地拆分文法内多个可选成分以尽可能减少回溯，如在成分 Stmt 推导时，检测下一个 Token的情况，若为 if 则直接转对if语句读取，若为 return 则转对return语句读取，若为 printf 则转对printf语句读取等。

但这样无法完全避免回溯，仍以之前的Stmt语句为例，在需要回溯的地方可以通过抛出错误来上溯到负责回溯的那一层，然后该层丢弃当前结果，复位读取位置，重新尝试读取。

思路4–stmt语句的识别

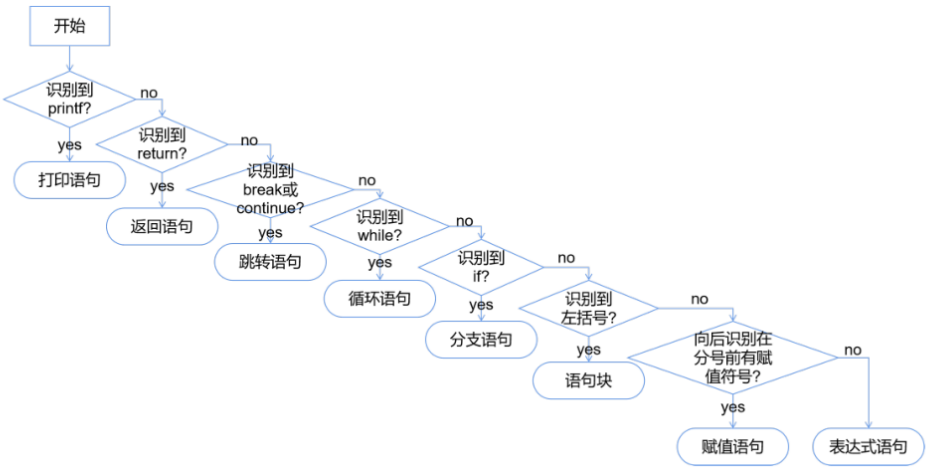
在stmt中，识别到的语句可能性很多，区分比较困难。

```
语句 Stmt → LVal '=' Exp ';'
  | [Exp] ';'
  | Block
  | 'if' '( Cond ')' Stmt [ 'else' Stmt ]
  | 'while' '(' Cond ')' Stmt
  | 'break' ';' | 'continue' ';'
  | 'return' [Exp] ';'
  | LVal = 'getint''('')'';'
  | 'printf' '('FormatString {',' Exp} ')'';'
```

首先对于打印语句、返回语句、跳转语句、循环语句、分支语句，可以很方便地通过第一个标识符（保留字）进行识别。如果第一个符号是左括号，则判断为语句块。

对于赋值语句和表达式语句，本文的识别方法是这样的：从当前位置向后找，判断先遇到分号还是先遇到赋值符号(‘=’)。如果先遇到赋值符号，则说明是赋值语句，否则是表达式语句。

判断思路如下图所示。



思路5–语法树设计

并不一定必须建立语法树，只是一种可选方案，总的来说都可以实现后续功能。语法树的结构可以自行定义，并没有一种确定的标准，只要在后续拆解语法树读取信息时能够进行识别判断即可。

方案1

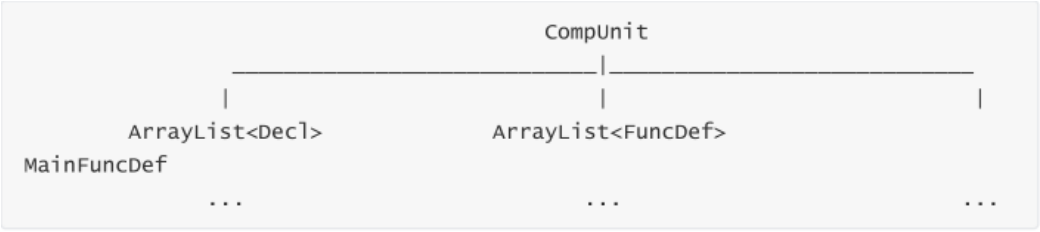
语法分析的初步设计阶段，并没有建立语法树，但是后面发现，如果不建立语法树，递归下降分析程序结束后，原代码信息就都没有了，就意味着需要在递归下降的同时进行错误处理和中

间代码的生成，所以为了结构上的解耦，可以根据需要建立语法树。

在每个语法成分的分析程序运行时，同时建立语法树的节点，并把该节点作为程序的返回值，以AddExp 为例:

```
//注释的部分显示了建立语法树与否的区别
private AddExp addExp() {
  // mulExp();
  MulExp temp = mulExp();
  AddExp addExp = new AddExp(temp);
  while (isOpeLevel2()) {
    if (printGrammarData) { //isOpeLevel2判断当前词法成分是否为'+'或'-'
    print("<AddExp>");
    }
    // getSym();
    // mulExp();
    String ope = ope();
    temp = mulExp();
    addExp = new AddExp(addExp, ope, temp);
  }
  if (printGrammarData) {
    print("<AddExp>");
  }
  return addExp;
}
```

最终的程序顶层语法树结构如下：



2.4 输入–输出评测约定

【问题描述】请根据给定的文法设计并实现语法分析程序，能基于上次作业的词法分析程序所识别出的单词，识别出各类语法成分。输入输出及处理要求如下：

1. 需按文法规则，用递归子程序法对文法中定义的语法成分进行分析；
2. 为了方便进行自动评测，输入的被编译源文件统一命名为testfile.txt（注意不要写错文件名）；输出的结果文件统一命名为output.txt（注意不要写错文件名）；结果文件中包含如下两种信息：
   a. 按词法分析识别单词的顺序，按行输出每个单词的信息（要求同词法分析作业，对于预读的情况不能输出）。
   b. 在文法中出现（除了<BlockItem>, <Decl>, <BType> 之外）的语法分析成分分析结束前，另起一行输出当前语法成分的名字，形如“<Stmt>”（注：未要求输出的语法成分仍需要进行分析，但无需输出）

【样例输入】

```
int main(){
```

```
    int c;
    c = getint();
    printf("%d",c);
    return c;
}
```

【样例输出】

```
INTTK int
MAINTK main
LPARENT (
RPARENT )
LBRACE {
INTTK int
IDENFR c
<VarDef>
SEMICN ;
<VarDecl>
IDENFR c
<LVal>
ASSIGN =
GETINTTK getint
LPARENT (
RPARENT )
SEMICN ;
<Stmt>
PRINTFTK printf
LPARENT (
STRCON "%d"
COMMA ,
IDENFR c
<LVal>
<PrimaryExp>
<UnaryExp>
<MulExp>
<AddExp>
<Exp>
RPARENT )
SEMICN ;
<Stmt>
RETURNTK return
IDENFR c
<LVal>
<PrimaryExp>
<UnaryExp>
<MulExp>
<AddExp>
<Exp>
SEMICN ;
<Stmt>
RBRACE }
<Block>
```

```
<MainFuncDef>
<CompUnit>
```

2.5 建议与感想

往届学长：

> 首先改写文法，消除左递归，采用递归下降分析法，对文法中除去BlockItem, Decl, BType 外的 所有非终结符编写分析程序，并根据当前的词法成分，以及一定的超前扫描，判断接下来是什么语 法成分，并调用相应的语法分析程序，在每个语法分析程序结束时输出该语法成分的名称即可完成 题目要求的输出。

三、错误处理/建立符号表

3.1 概述

错误处理指的是据给定的文法设计并实现错误处理程序，能诊察出常见的语法和语义错误，进行错误局部化处理，并输出错误信息。

3.2 总体思路与难点分析

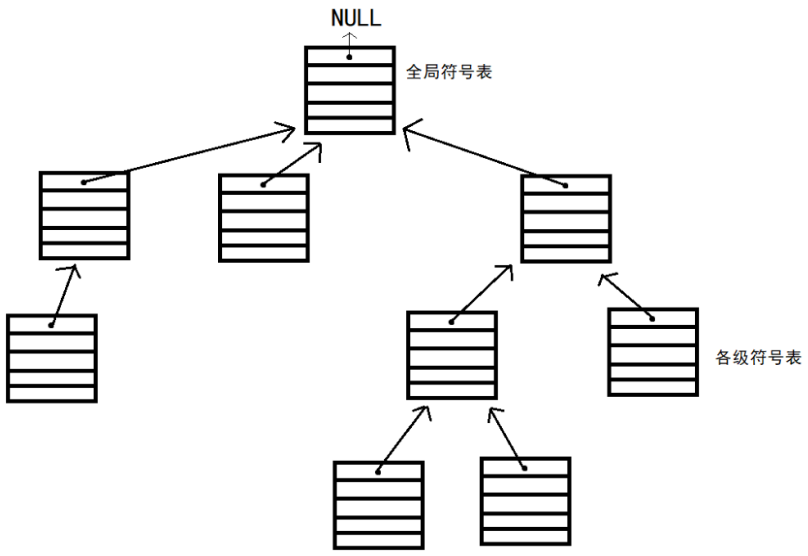
总体来说，错误处理的各个错误类型是相对独立的，彼此之间仅存在一定的联系，可以分别对每一种错误进行处理。但从共性上看，需要能够处理重复命名的变量、数据格式类型错误的变量等情况，而这些错误类型可以借助建立符号表的方式进行。以下是错误处理过程中的一些思路：

注意！以下思路仅供参考，仍需自行独立设计方案。

思路1-符号表

方案1

错误处理部分，符号表系统是重点。参考《Compilers: Principles, Techniques and Tools》一书，可设计单向树结构的符号表，每一级一张独立的符号表，其 parent 指向其直接上一级符号表。全局符号表的 parent 指向 null 。



如此设计下，可以保证每次在一张符号表上能且只能看见该符号表本身及所有所从属的符号表，看不见与其没有直接隶属关系的符号表。且在递归下降分析时，这类单向树符号表

的行为与栈类似，能很好适应递归下降时的符号表操作。但其能够在该过程结束后，进入翻译环节还能继续使用，而不是像栈式符号表一样使用完就被擦除覆写。

在加入符号时，符号表仅扫描本表内是否有重名变量。在查找符号时，从本级开始逐级向上查找，找到最近的一个即可。这就满足了内部符号覆盖外部符号的要求。

在具体实现上，有如下参考：

- 在数据结构上，建立符号类 Symbol 记录符号的名字，地址，维度，声明行号和使用行号等信息。
- 建立符号表类，用 map 的方式存储符号集 Symbol，其中，键为 Symbol 的名称，值为 Symbol 的指针

在算法流程上，首先建立根符号表，并将当前符号表的指针指向根符号表。

```
SymbolTable *symbolTable = new SymbolTable();
SymbolTable *currentSymbolTable = symbolTable;
```

每当进入一个新的 Block 或新的函数时，建立新的子符号表，并将当前符号表的指针 currentSymbolTable 指向新的子符号表。

```
Block* parseBlock() {
  currentSymbolTable = currentSymbolTable->newSon();
  // ...
  currentSymbolTable = currentSymbolTable->back();
  return b;
}
```

当遇到声明（Decl）语句和函数形参声明时，向符号表中填入 Symbol 并检查是否存在重名问题。

在 Stmt 中使用符号时，由当前符号表逐层向上检索所用符号名和符号类型，检查是否存在未定义和改变 const 值等问题。

符号表实现中的困难主要来自于对变量生存周期和作用域的理解，因此需要同学们从宏观上有一个整体的把握，采用类似的策略，对函数也建立函数表进行存储与记录。

思路2–具体实现1

在递归下降部分如果采用含有回溯的处理方式，简单地直接产生错误可能会导致一处错误被多次报告的情况，因此在对如 LVal 等部分进行错误处理时，不再直接处理，而是记录错误情况，作为 Exception 抛出，交给上一层逻辑判断是否将本次报错加入已检测的错误列表。

某些部分如缺少括号、分号之类进行错误处理后，后面的内容会因为检测不到前一处应有的符号而报错，为了保证程序顺利运行、继续检测后续的错误，可能需要设定一些机制保证对缺失符号的处理，例如将缺失的符号补回或跳过读取。

但是需要注意的是，程序中变量的作用域是从定义点开始向后有效。对于定义点之前的变量使用，需要向上一级查找。因此，在查询符号表，特别是已经建立好的完整的符号表的时候，不可以简单地向该层申请查询。例如对于如下的情况：

```
int main() {
  int a = 100;
  {
    {
```

```
      int b = a + 1; // 此处需要查询a
      a = 10086;
    }
    a = 114514;
  }
}
```

在处理注释所在行时，需要在符号表中查询a。若不加以限制地从本级开始递归向上查询，会得到a的值为10086，显然是错误的。因此，查询时需要加上本级的位置。

但仅仅加上本级位置是不够的，它可以绕开本级定义，向上一级查询。但向上一级符号表查询时，没有说明从上一级符号表的何处查询，仍会查询到后面定义的 a = 114514 。因此，在每次新建符号表时，需要记录该符号表对应的执行环境位于上一级执行环境的什么位置。

思路3-具体实现2

- 错误a：FormatString中包含非法符号的情况：处理方式为对每一处FormatString作检查，当检测到包含非法符号时（字符的ASCII码不在指定范围内）抛出a类错误。
- 错误b+c：处理方式为建立符号表，对于b类重定义的情况，每次调用该标识符时检查在当前作用域下是否已定义具有相同名字的函数名或变量名？对于c类错误，则检查表中所有数据检测是否有未定义的情况。
- 错误d+e：均为函数调用语句中，参数个数或类型不匹配的情况。
  - 对于d类错误及函数参数个数不匹配的情况，分两个部分进行处理，首先在函数定义时为函数记录下该函数名拥有的参数以及其类型、个数等特征，之后在函数调用时对参数个数进行计数，若不匹配则抛出d类错误。
  - 错误e：对于e类函数参数类型不匹配的情况，在每一个函数参数进行检查时标记该参数的维度，由于仅有二维数组、一维数组、整数，以及void等类型的情况，且仅需考虑数据的维度是否匹配，之后检测函数对应位置参数的维度与此时传入参数的维度是否相同，若不相同则抛出e类错误。
- 错误f+g：f类与g类错误均与函数的返回值有关，处理思路也非常相近。
  - f类错误为无返回值的函数存在不匹配的return语句，报错的行号为return所在行数。无返回值的函数即为void的类型的函数，由于允许return后面直接加分号的形式存在，因此当检测到了return时不能直接认定存返回值，同时还要检查return后面是否有Exp()类型的表达式语句，若无表达式语句，可以认为仍然是存在一个无返回值的返回语句，并不予报错，若一直到void函数末尾均未检测到return类型，则符合无返回值函数的要求。
  - g类错误则为有返回值的函数缺少return语句，报错的行号为结尾右大括号所在行数。对函数block结构体内最后一句是否为return类型的语句进行检测，若最后一句不为return语句或return语句中无表达式，则认为此时发生该类错误，该函数无返回值。
- 错误h：h类错误为修改了常量的值：检测方式为每当对LVal进行赋值时，检查是否该标识符为Const的常量，不可改变值，若符合，则报错。
- 错误i+j+k：为缺少分号小括号中括号的情况，对于这几类情况在语法分析过程中可顺便处理。当检测到缺少的符号为这几种符号的情况，可直接抛出对应类型的错误及错误码，从而完成错误检测。
- 错误l：l类错误与a类错误类似，均是对字符串检查，对应的检查逻辑为在检查过程中分别计数，print语句中有多少个应当输出的参数，以及逗号后面实际传入参数的个数检查，二者不

匹配时抛出l类错误。

- 错误m：m类错误为非循环体内具有break与continue语句，对于这两种情况，则需要设置一个全局变量存储此时是否为函数循环函数体，若在循环函数体内调用break给continue依据则认为是正常逻辑，否则抛出错误。此处设置了一个cycleDepth参数，标识深入的循环结构层数，当进入一个循环体时计数+1，退出时计数-1，为0则说明此时未在循环体内。

3.3 输入-输出评测约定

输入输出及处理要求如下：

1. 输入的被编译源文件统一命名为 testfile.txt ；错误信息输出到命名为 error.txt 的结果文件中；
2. 结果文件中包含如下两种信息：错误所在的行号 错误的类别码 （行号与类别码之间只有一个空格，类别码严格按照表格中的小写英文字母）。其中错误类别码按下表中的定义输出，行号从1开始计数：

错误类型	错误类别码	解释	对应文法及出错符号(...省略该条规则后续部分)
非法符号	a	格式字符串中出现非法字符报错行号为<FormatString>所在行数。	<FormatString> → ‘“‘{<Char>}’”’
名字重定义	b	函数名或者变量名在**当前作用域**下重复定义。注意，变量一定是同一级作用域下才会判定出错，不同级作用域下，内层会覆盖外层定义。报错行号为<Ident>所在行数。	<ConstDef>→<Ident> … <VarDef>→<Ident> … |<Ident> … <FuncDef>→<FuncType> <Ident> …<FuncFParam> → <BType> <Ident> ...
未定义的名字	c	使用了未定义的标识符报错行号为<Ident>所在行数。	<LVal>→<Ident> … <UnaryExp>→<Ident> …
函数参数个数不匹配	d	函数调用语句中，参数个数与函数定义中的参数个数不匹配。报错行号为函数调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>‘(’[FuncRParams]‘)’
函数参数类型不匹配	e	函数调用语句中，参数类型与函数定义中对应位置的参数类型不匹配。报错行号为函数调用语句的**函数名**所在行数。	<UnaryExp>→<Ident>‘(’[FuncRParams]‘)’
无返回值的函数存在不匹配的return语句	f	报错行号为**‘return’**所在行号。	<Stmt>→‘return’ {‘[’Exp’]’}‘;’
有返回值的函数缺少return语句	g	只需要考虑函数末尾是否存在return语句，**无需考虑数据流**。报错行号为函数**结尾的**’}’**所在行号。	FuncDef → FuncType Ident ‘(’ [FuncFParams] ‘)’ BlockMainFuncDef → ’int’ ’main’ ’(’ ’)’ Block

不能改变常量的值	h	<LVal>为常量时，不能对其修改。报错行号为<LVal>所在行号。	<Stmt>→<LVal>'=' <Exp>';'|<LVal>'=' 'getint' '(' ')' ';'
缺少分号	i	报错行号为分号**前一个非终结符**所在行号。	<Stmt>,<ConstDecl>及<VarDecl>中的';'
缺少右小括号')'	j	报错行号为右小括号**前一个非终结符**所在行号。	函数调用(<UnaryExp>)、函数定义(<FuncDef>)及<Stmt>中的')'
缺少右中括号']'	k	报错行号为右中括号**前一个非终结符**所在行号。	数组定义(<ConstDef>,<VarDef>,<FuncFParam>)和使用(<LVal>)中的']'
printf中格式字符与表达式个数不匹配	l	报错行号为**'printf'**所在行号。	Stmt → 'printf''('FormatString{,Exp}')'';'
在非循环块中使用break和continue语句	m	报错行号为**'break'**与'**continue'**所在行号。	<Stmt>→'break'';'|'continue'';'

【样例输入】

```
const int const1 = 1, const2 = -100;
int change1;
int gets1(int var1,int var2){
   const1 = 999;
   change1 = var1 + var2          return (change1);
}
int main(){
   change1 = 10;
   printf("Hello World$");
   return 0;
}
```

【样例输出】

```
4 h
5 i
9 a
```

3.4 建议与感想

往届学长：

> 第一个难点：函数参数类型不匹配
>
> 我一开始的设计是当读取完这个函数的全部内容后，才会将这个函数的符号项加入到当前的符 号管理表中，但是当递归函数出现时，处理递归函数里面时，就会发现我找不到这个函数项， 所以就会报错，但实际上并没有错。所以解决办法是在进入到函数时就将该符号项加入到符号 管理表中。
>
> 第二个难点：递归函数的处理

> 我一开始的设计是当读取完这个函数的全部内容后，才会将这个函数的符号项加入到当前的符 号管理表中，但是当递归函数出现时，处理递归函数里面时，就会发现我找不到这个函数项， 所以就会报错，但实际上并没有错。所以解决办法是在进入到函数时就将该符号项加入到符号 管理表中。
>
> 第三个难点：函数作用域的确定
>
> 作用域的确认可能比较简单，那就是 {} 分割，但是，函数形参却是一个特殊的存在，函数 形参的作用域是函数主体的作用域，所以，函数形参不能重名，函数形参也不能和主体作用域 的变量重名。

后端

四、中间代码设计

经历了重重分析，我们终于要对源程序进行转化了，那就是“生成中间代码”。其实，生成中间代码也是伴随着语法分析和语义分析进行的，我们可以这么理解，我们的编译器的工作可以大致分为两部分，第一部分是通过语法分析和语义分析，将源程序翻译为我们自己定义的中间代码（在课设中推荐用四元式的形式）；第二部分是通过分析中间代码，将其翻译为最终的目标代码。

也就是说，目标代码的生成，是完全基于中间代码的。所以，中间代码的正确性以及高效性，将对目标代码的正确性以及效率造成很大影响。这里你可能会有个疑问：如果不做中间代码设计，而是直接将源程序翻译成目标代码，是否也可行？答案是肯定可以实现的，但会造成代码臃肿、难以优化、调试困难等各种问题，而且还会耗费大量的时间和精力。所以中间代码是十分必要且重要的！

下面我们给出几种中间代码的参考格式，同学们完全可以基于自己的想法来自由发挥。只要设计的中间代码可以准确无误地表达源程序的逻辑，并且便于优化，就是好的中间代码。

1. 四元式

此处我们介绍一种中间代码的形式——四元式。书上是这么介绍四元式的：四元式是 N–源表示的一种，四元式的每条指令有 4 个域：

< 操作符 >, < 操作数 1>, < 操作数 2>, < 结果 >

其中，< 操作数 1> 和 < 操作数 2> 分别表示第一个操作数和第二个操作数，< 结果 > 表示计算的结果，该结果通常是一个临时变量，编译程序可以为该变量分配一个寄存器或一个主存地址。

但笔者以为这样并不是十分好理解，遇到某些语句也并不能严格按照 < 操作数 1> 和 < 操作数 2> 来翻译，因此笔者认为把四元式结构改成：< 操作符 >，< label1 >，< label2 >, < 结果 > 更好理解一些。对于符号表的设计，可能不同人所设计的数据结构并不相同，有的记录了 4 个信息，有的记录了 5 个信息，但只要能够保证可以通过符号表查找到自己所需要的全部信息即可。四元式也是如此，只要通过两个 label，能够准确翻译源程序，生成正确的中间代码即可，至于这两个 label 是否都用到了，不必太过死板。比如说下面三条语句：

```
a = b;
a = b + c;
a = num[3];
```

对于第一条语句，不难理解生成的四元式应当是： =, b, , a

对于第二条语句，首先我们要计算 b + c 的值，然后将计算结果赋值给 a，则生成的四元式应当是： +, b, c, t0 和 =, t0，a

对于第三条语句，首先我们要将相对于 num 这个数组的首地址偏移量为 3 的内存 中的值取出，然后将数值赋值给 a，则生成的四元式应当是：[], num, 3, t1 和 =, t1, , a

可以看出，第一条语句生成的四元式中，我们只使用了三个信息，有一个 label 并没有使用；第二条语句中，“+, b, c, t0”是一个标准的“< 操作符 >, < 操作数 1>, < 操作数 2>, < 结果 >”格式；而第三条语句中，取 num[3] 的值所对应的四元式“[], num, 3, t1”。

当然，四元式具体如何设计，或者说是否要使用四元式，这都取决于个人设计。

对于实验要求的文法来说，下面简单分析一下在生成中间代码阶段的各类语句（仅仅是笔者的浅见，完全可以自行设计）：

- 条件语句：文法中的条件语句包括 if-then 和 if-then-else 两种结构。对于这两种结构的设计，都是先计算出 if 条件是否为真：
  - 若条件为真，则执行 if 后的语句（需要注意的是，若存在 else 分支，则 else 分支的语句在 if 后的语句执行完毕后就不必执行了，可以通过在 if 语句最后加入一个跳转至 else 分支的语句之后的中间代码来实现）；
  - 若条件为假，则执行 else 分支的语句（存在 else 分支）或退出该条件语句；
- 循环语句：文法中的循环语句包括 while、do-while 和 for 三种结构。与条件语句类似，这三种结构也是通过在不同条件下实现不同的跳转来实现的，就不过多阐述了；
- 函数调用语句：对于函数调用语句，在生成中间代码时可以通过将先将参数压入栈中，再调用相应的函数来实现；
- 赋值语句：即先计算赋值号右边表达式的值，再将其赋给赋值号左侧的变量；

2. 中间代码参考

原则上按照中缀表达式格式输出中间代码，即，形如 x = y op z，其中 x 为结果，y 为左操作数，z 为右操作数，op 为操作符。以下根据基本语法现象举例说明。

（1）函数

- **函数声明**

```
int foo(int a, int b, int c, int d) {
  // ...
}
```

参考中间代码为：

```
int foo()
para int a
para int b
para int c
para int d
```

- **函数调用**

```
i = tar(x, y);
```

参考中间代码为：

```
push x
```

```
push y
call tar
i = RET      ## RET表示函数tar的返回值
```

- **函数返回**

```
return x + y;
```

参考中间代码为：

```
t1 = x + y
ret t1
```

（2）变量和常量

- **变量声明及初始化**

```
int i;
int j = 1;
```

参考中间代码为：

```
var int i
var int j = 1
```

- **常量声明**

```
const int c = 10;
```

参考中间代码为：

```
const int c = 10
```

（3）分支和跳转

- **标签**

```
label:
  z = x + y
```

- **条件分支**：注意涉及 && 和 || 的地方要满足逻辑短路

```
if (a == 1 && b <= 2) {
  // ...
}
```

参考中间代码为：

```
cmp a, 1
bne end_if
cmp b, 2
bgt end_if
// if body ...
end_if:
```

- **跳转**：典型的如continue和break语句

```
while (/* ... */) {
```

```
  if (/* ... */) {
    break;
  }
}
```

参考中间代码为：

```
loop_begin:
// some statements ...
goto loop_end:
// some statements ...
loop_end:
```

(4) 数组

- **数组定义**

```
int a[4] = {1, 2, 3, 4};
```

参考中间代码为：

```
arr int a[4]
a[0] = 1
a[1] = 2
a[2] = 3
a[3] = 4
```

- **数组读取**

```
z = a[x][y];   // 定义时宣称 int a[4][2];
```

参考中间代码为：

```
t1 = x * 2
t2 = y + t1
z = a[t2]
```

- **数组存储**

```
a[x][y] = z;   // 定义时宣称 int a[4][2];
```

参考中间代码为：

```
t1 = x * 2
t2 = y + t1
a[t2] = z
```

(5) 输入输出

- **输入**

```
i = getint();
```

参考中间代码为：

```
scanf t1
i = t1
```

- **输出**

```
printf("The result of i*i is %d\n", i*i);
```

参考中间代码为：

```
const str str_0 = "The result of i*i is "  // 用于后面输出
const str str_1 = "\n"

printf str_0
printf i
printf str_1
```

注意到我们这里将原输出语句拆成了三部分分别进行输出，这是考虑到了中间代码到目标代码的翻译实现；具体为什么要这么做，在第3节中会有更详细的解释。

(6) 补充示例

```
const int c1 = 2;
const int c2 = 1;
int foo(int a, int b) {
  return a + b;
}

int func() {
  return c1 * c2;
}

int main() {
  int v1 = 0, v2 = 1;
  int res;
  if (v1 == 0 && v2 <= 1) {
    res = foo(v1, v2) + 1;
  } else {
    res = func();
  }
  printf("%d\n", res);
  return 0;
}
```

对于如上的程序段，按照我们给出的中间代码参考格式，可得如下中间代码：

```
const int c1 = 2
const int c2 = 1
const str str_0 = "\n"      // 这里宣称字符串常量'\n'，用于后面输出

int foo()
para a
para b
t1 = a + b
ret t1
int func()
t1 = c1 * c2
```

```
ret t1
int main()
var int v1 = 0
var int v2 = 1
var int res

cmp v1, 0
bne if_end
cmp v2, 1
bgt if_end

# if body :
push v1
push v2
call foo
t1 = RET
t2 = t1 + 1
res = t2
goto else_end

if_end:    # label
# else body :
call func
t3 = RET
res = t3

else_end:
printf res
printf str_0
```

当然，这里给出的代码示例只包含了一些比较基础的语法，并没有覆盖文法中可能含有的所有情况。对于更复杂的情况，还需要同学自行思考中间代码的设计方案。

3. 其他说明

以上给出的四元式设计和中间代码规范格式都仅仅是**中间代码参考架构**，同学在设计过程中可以结合自己的想法对中间代码格式进行调整。中间代码的设计将直接影响到目标代码生成，所以同学在设计中间代码时一定要考虑到“如何将中间代码翻译成目标代码”，最好能具体到一些细节，以减少目标代码生成阶段中对中间代码的重构。若目标代码为MIPS指令集等汇编语言，中间代码需要尽可能向着汇编语言贴近，有些甚至可能就直接用汇编语言代替。**就目标代码为MIPS而言**，笔者有以下几点设计建议：

（1）输出时区分数值和字符串

```
printf("The result of i*i is %d\n", i*i);
```

在上一小节中，我们将如上的 c 代码翻译成了如下的中间代码形式：

```
const str str_0 = "The result of i*i is "  // 用于后面输出
const str str_1 = "\n"

printf str_0
printf i
```

```
printf str_1
```

可以看到这里我们根据 ”%d” 将原输出语句拆成了三部分分别进行输出。在MIPS指令集中，对于数字和字符串的输出方式是不一样的，系统调用时要设置不同的参数。所以在中间代码生成阶段，可根据 ”%d” 的位置将输出语句进行拆分，然后依次在中间代码中输出。在中间代码生成阶段，可以标识一下此输出字段是**字符串类型还是数字类型**，以便于后面目标代码的生成。对于字符串类型可在中间代码的最开始声明一下，便于生成目标代码时填入到 .data 字段中去。

（2）变量名添加唯一标识

变量名覆盖是符合Sys文法的，因此测试代码中可能会出现多个同名变量在不同的语句块中出现的情况，会给代码生成中变量分配空间带来一些问题，对于代码优化中的数据流分析也十分不友好。在设计中间代码时，需考虑给每个变量添加**唯一标识码。**

例如下面这段代码是符合文法的，但是在多个基本块都定义了变量 i 。设计中间代码时可以标识一下当前这个 i 是全局变量还是局部变量，在第几层的基本块中等等。

```
int i;
int func(int i) {
  //...
  {
      int i;
      //...
  }
}
int main() {
    int i;
    //...
    {
        int i;
        //...
    }
    //..
}
```

（3）参数是数值还是地址

在实验的文法中，函数调用传参时可能会涉及到数组地址操作的问题。设计数组的中间代码时，需考虑传入函数的参数是数组中某个元素的数值还是数组的地址，并能在中间代码中明确表达出**传入的参数是数值还是地址。**对于实验要求的文法，考虑传入的参数为地址的情况可以分为两种：一是传入的参数类型为局部变量或全局变量；二是传参的语句位于函数体中，传入的参数是此函数的参数。

下面这段代码中第10行关于b值计算的语句就是我们所说的第一种情况：传入参数为局部变量或全局变量。

```
int glo_var[2][2] = {{1,2},{3,4}};

int func(int a[]) {
  return a[0] * a[1];
}

```

```
int main() {
  int loc_var[2] = {1,2};
  int b;
  b = func(glo_var[0]) + func(loc_var);       # 传入参数为变量类型
  printf("%d\n", b);
  return 0;
}
```

下面这段代码中第6行关于foo函数返回值的计算就是我们上面所说的第二种情况：传入参数为此函数体的参数。

```
int func(int a[]) {
  return a[0] * a[1];
}

int foo(int b[][]) {
  return func(b[0]) * func(b[1]);            # 传入参数为函数体的参数
}

int main() {
  int array[2][2] = {{1, 2}, {3, 4}};
  int b;
  b = foo(array);
  printf("%d\n", b);
  return 0;
}
```

在目标代码生成阶段，对于这两种情况的处理方式是完全不一样的。以上只是笔者给出的参考思路，建议同学们在设计时一定要将函数传参这块思考足够深，最好先用MIPS指令集翻译一遍，以体会不同传参方式的在目标代码生成上的区别。

五、目标代码生成

中间代码生成后，我们就来到了编译器设计的最后阶段了。目标代码生成通常以编译器此前生成的中间代码、符号表及其他相关信息作为输入，输出与源程序语义等价的目标程序代码。代码生成模块需要面向某一个特定的目标体系结构生成目标代码，这种目标体系结构可以是X86、MIPS、ARM等，因此我们可以把目标代码生成理解成是对中间代码的翻译。对于不同体系结构，中间代码翻译成目标代码的处理方式也不同。下面我们给出一个目标代码为MIPS指令结构的例子。

原代码为：

```
const int c1 = 2;
const int c2 = 1;
int foo(int a, int b) {
  return a + b;
}

int func() {
  return c1 * c2;
}

```

```c
int main() {
  int v1 = 0, v2 = 1;
  int res;
  if (v1 == 0 && v2 <= 1) {
    res = foo(v1, v2) + 1;
  } else {
    res = func();
  }
  printf("%d\n", res);
  return 0;
}
```

对应的中间代码为：

```
const int c1 = 2
const int c2 = 1
const str str_0 = "\n"      // 这里宣称字符串常量'\n'，用于后面输出

int foo()
para a
para b
t1 = a + b
ret t1
int func()
t1 = c1 * c2
ret t1
int main()
var int v1 = 0
var int v2 = 1
var int res

cmp v1, 0
bne if_end
cmp v2, 1
bgt if_end

# if body :
push v1
push v2
call foo
t1 = RET
t2 = t1 + 1
res = t2
j else_end

if_end:    # label
# else body :
call func
t3 = RET
res = t3

else_end:
```

```
printf res
printf str_0
```

翻译成目标代码后可得：

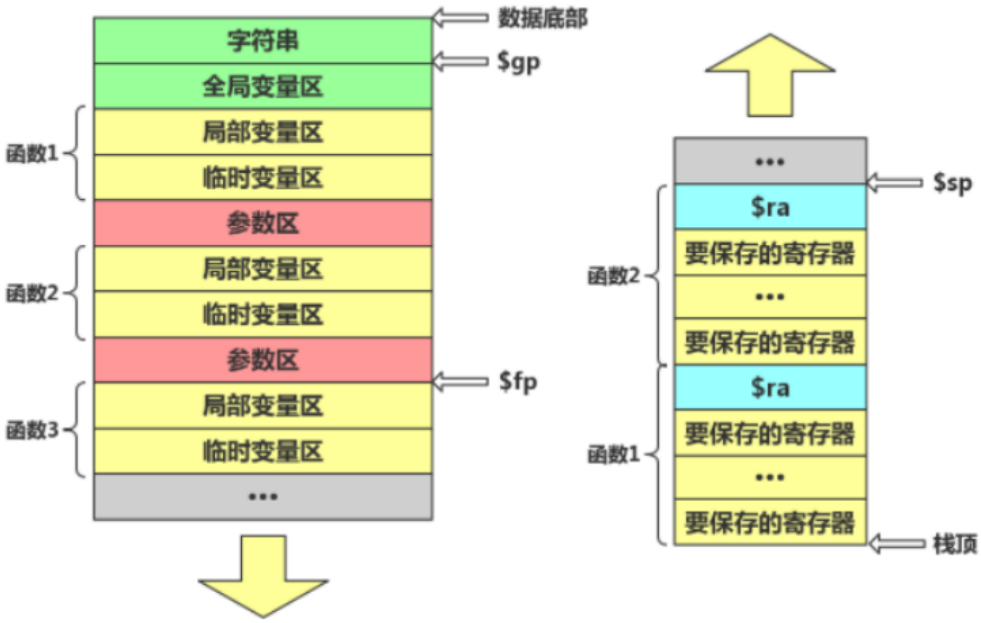
```
.data
str_0: .asciiz "\n"      # str_0 "\n"
.space 4
str_end:
.text
li $fp, 0x10040000

j func_main
nop

func_foo:
lw $s0, 0($fp)       # 加载参数a
lw $s1, 4($fp)       # 加载参数b
add $v0, $s0, $s1    # 计算 a + b，作为函数的返回值
jr $ra

func_func:
li $v0, 2          # 常量c1 * c2 = 2
jr $ra

func_main:
li $s0, 0          # var v1 0
li $s1, 1          # var v2 1

li $t0, 0
bne $s0, $t0, if_end  # if v1 == 0 then ...
li $t1, 1
bgt $s1, $t1, if_end  # if v2 <= 1 then ...

sw $s0, 12($fp)        # 传入参数 v1
sw $s1, 16($fp)        # 传入参数 v2

addi $sp, $sp, -12     # 压栈，保存现场
sw $s0, 0($sp)         # 保存现场，此处保存寄存器$s0
sw $s1, 4($sp)         # 保存现场，此处保存寄存器$s1
sw $ra, 8($sp)         # 保存现场，此处保存寄存器$ra

addi $fp, $fp, 12      # 移动帧指针
jal func_foo           # 调用函数foo()
addi $fp, $fp, -12     # 移动帧指针

lw $s0, 0($sp)         # 恢复现场
lw $s1, 4($sp)         # 恢复现场
lw $ra, 8($sp)         # 恢复现场
addi $sp, $sp, 12      # 弹栈，恢复现场

move $t2, $v0
```

```
addi $t3, $t2, 1       # 计算 res = foo(v1, v2) + 1
move $s2, $t3          # $s2寄存器分配给变量res

j else_end             # 不经过else语句块

if_end:                # label if_end
addi $sp, $sp, -16     # 压栈，保存现场
sw $s0, 0($sp)         # 保存现场，此处保存寄存器$s0
sw $s1, 4($sp)         # 保存现场，此处保存寄存器$s1
sw $s2, 8($sp)         # 保存现场，此处保存寄存器$s2
sw $ra, 12($sp)        # 保存现场，此处保存寄存器$ra

addi $fp, $fp, 12      # 移动帧指针
jal func_func          # 调用函数func()
addi $fp, $fp, -12     # 移动帧指针

lw $s0, 0($sp)         # 恢复现场
lw $s1, 4($sp)         # 恢复现场
lw $s2, 8($sp)         # 恢复现场
lw $ra, 12($sp)        # 恢复现场
addi $sp, $sp, 16      # 弹栈，恢复现场

move $t4, $v0          # 临时寄存器$t4存储函数func()的返回值
move $s2, $t4          # $s2寄存器分配给变量res

else_end:              # label else_end
li $v0, 1
move $a0, $s2          # printf res
syscall
li $v0, 4
la $a0, str_0          # printf "\n"
syscall
li $v0, 10
syscall
```

1. 存储管理

存储空间的分配是目标代码生成阶段的重难点。编译器要在代码生成阶段为目标代码本身，以及全局变量、局部变量、临时变量和其他数据结构指定相应的存储空间。因此在翻译目标代码之前，必需先考虑好存储空间的分配策略。目前两种标准的存储分配策略是静态分配和栈式分配。对于静态分配，内存中的活动记录在编译时便已确定。对于栈式分配，过程的每次执行都将在栈顶压入一个新的活动记录，活动结束时将该记录弹出。

对于简单的类 C 文法来说，运行时活动记录的分配和释放是作为函数调用和返回序列的一部分，也即主要包括：call 和 return。同时还需要思考函数被调用时如何传递参数。下面我们给出一个可供参考的动态存储分配方案（箭头方向表示存储数据增长的方向）



对于左图，每个函数都有对应的活动记录，里面存储了函数内部的参数、局部变量和临时变量。将存储的数据类型归为常量、字符串、全局变量、局部变量、临时变量和参数，它们的存储分配方式为：

- 常量：若是数值，可以直接查符号表得知，不分配存储空间；若是数组，则按照变量的方式分配存储
- 字符串：使用 `.asciiz` 存储在 `.data` 字段中
- 全局变量：以 `$gp` 为栈底，向上存储
- 参数：前4个参数存储在参数寄存器`$a0~$a3`中，序号大于4的则以`$fp`为栈底，在内存中向上增长
- 局部变量：以参数区的栈顶作为局部变量区的栈底，向上增长
- 临时变量：以局部变量区的栈顶作为临时变量区的栈底，向上增长；当一个临时变量被使用后立即释放

右图是关于函数调用时寄存器压入栈的存储分配。每当调用函数时，$sp 都在栈底，直接通过 $sp 进行参数传递(PUSH)，之后 $sp 变为被调用函数的栈底，将 $ra 、 $fp 存储到 $sp 偏移4和8的位置， $fp 变为将函数被调用函数的栈顶，跳转需要保存的寄存器的值也通过 $sp 保存起来。这样跳转到新的函数后，就可以通过 $fp 进行参数和局部变量、临时变量的存取了。跳转返回后，通过 $sp 恢复 $ra 、 $fp 以及其他需要恢复的寄存器的值，通过 $fp 进行原函数的参数和局部变量、临时变量的存取。

2. 一些难点的参考思路

（1）运算表达式

如果中间代码生成阶段没问题的话，此过程应该是比较轻松的。但需要注意的是运算符号的**优先级**，这在文法中已经体现出来了。按照文法给出的优先级关系递归解析即可，解析一次分配一个临时变量。

```
sum = a + b * (c - d) % -e
```

对应的中间代码可表示为：

```
t1 = c - d
```

```
t2 = -e
t3 = b * t1
t4 = t3 % t2
t5 = a + t4
sum = t5
```

（2）函数调用

函数调用的整个过程可以分为：

- 进入函数前保存现场寄存器
- 使用 jal 指令跳转到目标函数并连接当前 PC 值
- 退出函数后恢复现场寄存器

一般来说，$ra（存有返回地址，函数调用使用 jal 指令会改变此寄存器的值）和 $fp 这两个寄存器是函数调用时必须保存的寄存器，需要压入栈中。而其他寄存器是否需要保存，并没有标准答案。保存的寄存器越多，会带来更高的安全性，但同时造成的时间开销也会越大。具体需要保存和恢复哪种类型的寄存器，保存寄存器的数目是否要受到限制，这些需要同学们兼顾正确性和效率自行设计。

（3）短路求值

对于 if 条件分支语句的 && 短路， 可作如下变换：

```
// 变换前
if (a && b) {
  // then_body
} else {
  // else_body
}

// 变换后
if (a) {
  if (b) {
    // then_body
  } else {
    // else_body
  }
} else {
    // else_body
}
```

对于 if 条件分支语句的 || 短路，可作如下变换：

```
// 变换前
if (a || b) {
  // then_body
} else {
  // else_body
}

// 变换后
if (a) {
    // then_body
```

```
} else {
    if (b) {
        // then_body
    } else {
        // else_body
    }
}
```

若有 && 和 || 嵌套，可以按照符号的优先级将其拆成多个判断语句来处理。解析时遵循以下几点：

- 对 LAndExp 中每个 RelExp 进行判断：若该RelExp为真，则往后判断下一个 RelExp；若为假，则直接跳转到当前 LAndExp 的尾部
- 对 LOrExp 中的每个 LAndExp 进行判断：若该LAndExp为真，则直接跳转进入到 if 语句块内部；若为假，则往后判断下一个 LAndExp
- 若所有的 LOrExp 判断完毕后还没发生跳转，说明不满足判断条件，此时跳转到 if 语句块尾部（或 else 语句块首部）

```
if (a || b && c || d) {
  // do...
}
```

上述代码可以展开成如下语句

```
if (!a) {
  goto IF_OR_1
}
goto IF_BEGIN

IF_OR_1:
if (!b) {
  goto IF_OR_2
}
if (!c) {
  goto IF_OR_2
}
goto IF_BEGIN

IF_OR_2:
if (!d) {
  goto IF_OR_3
}
goto IF_BEGIN

IF_OR_3:
goto IF_END

IF_BEGIN:
    // do...
IF_END:
```

可以看到第21、22行中标签 IF_OR_3 后紧跟着跳转到 IF_END 的语句，在后续的代码优化中可以删掉这一冗余代码，减少一次跳转。

（4）数组操作

不同类型数组在存储空间中的分配方案分别为

- 如果数组是全局数组，那么它里面的各个值依次排列在全局数据区
- 如果数组是函数中定义的局部数组，则数组中的值依次排列在对应函数的活动记录中
- 如果数组是参数数组，那么活动记录中记录的是数组的基地址

需要注意的是，为数组分配存储空间时不特别区分一维数组与二维数组：它们的区别仅仅在于访问数组元素时的偏移计算方法有差异。

我们将与数组相关的操作分为两种：数组存取，数组地址传递。

- 数组存取：当从某个数组中获取值或者向某个数组中存储值时，首先判断这个数组是全局数组还是局部数组，然后获得其相对于静态数据区（或者当前活动记录基地址）的偏移，与$gp（或$fp）求和，获得数组基地址。然后根据访问数组的位置，求出相对于数组基地址的偏移。最后用lw或者sw访存。
- 数组地址传递：仅出现在函数调用时。 当调用某个函数时，如果这个函数需要一个数组作为参数，则需要把某个全局数组或者局部数组或参数数组的地址传入。若需要传入一个非参数数组的地址，那么可以在编译时确定数组相对于静态数据区或当前活动记录的偏移，从而与$gp或$fp求和，获得数组地址并传入。如果是参数数组，那么从活动记录中获取到的数组值已经是数组地址，直接传入即可。如果需要传入二维数组中的某个一维数组的地址，那么还需要加上偏移（动态计算） 。

（5）寄存器分配

在不考虑优化的情况下，完全可以将所有变量都存储在内存中，寄存器仅暂存变量的值，操作结束后将结果写入内存中的相应位置。但是操作数在寄存器中的数据通常要比操作在内存中的数据具有更高的效率。充分利用寄存器以生成好的代码就显得尤为重要了，因此鼓励同学们多考虑如何使用寄存器，而不是每次操作都访问内存。**如果寄存器分配做得好的话，即使不做其他优化，最后可能也会达到很好的效果！**寄存器的使用可以分为两个子问题：

- 在寄存器分配期间，在程序的某一点选择要驻留在寄存器中的变量集；
- 在随后的寄存器指派阶段，调出变量将要驻留的具体寄存器。

而选择最优的寄存器指派方案是非常困难的，只能选择一个相对较好的方案。目前成熟的分配方案由引用计数和图着色算法，建议大家使用这两种较规范的寄存器分配方法。此外，建议遵循MIPS通用寄存器的使用约定和调用规范：

REGISTER	NAME	USAGE
$0	$zero	常量0
$1	$at	保留给汇编器
$2–$3	$v0–$v1	函数调用返回值
$4–$7	$a0–$a3	函数调用参数
$8–$15	$t0–$t7	临时变量寄存器
$16–$23	$s0–$s7	子函数变量寄存器
$24–$25	$t8–$t9	更多临时变量寄存器
$26–$27	$k0–$k1	保留给中断或异常处理程序
$28	$gp	全局指针
$29	$sp	栈指针
$30	$fp	帧指针
$31	$ra	函数调用返回地址

六、代码优化

完成代码生成后，我们的编译器就可以保证执行结果的正确性了。但是为了提高编译器的效率和性能，我们还需要做代码优化处理，对中间代码和目标代码进行优化处理。

在我们这门课程中，代码优化阶段建议大家完成的优化包括基本块内部的公共子表达式删除（DAG 图）、全局寄存器分配（引用计数或着色算法）、数据流分析（通过活跃变量分析，或利用定义–使用链建网等方法建立冲突图）等。除此之外，学有余力的同学还可以自行选择其他的优化方案，例如常量合并、基本块内的复制传播和常量传播、跨基本块的复制传播和常量传播、循环强度削弱、运算强度削弱、函数内联、死代码删除、选择高效指令、窥孔优化等。但需要注意的是，不是说做了代码优化最终就一定能有靠前的排名。竞速排名的参考标准以当课程组发布的Cycle计算公式为准，建议同学们根据Cycle计算公式选择着重考虑的优化方案。

1. 基本块划分和流图

大部分的优化方法都离不开数据流分析，因此需要先将中间代码划分成基本块并构建流图。课本中基本块的划分是先要确定每个基本块的入口语句，以此来划分，入口语句的定义如下：

1. 整个语句序列的第一条语句属于入口语句
2. 任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句
3. 紧跟在跳转语句之后的第一条语句属于入口语句

确定好每个基本块的入口语句后，就可以划分基本块并构建流图了。具体实现过程可参考以下几条原则：

- 基本块只在函数内部进行连接，不能在两个函数之间连接基本块
- 开始时先初始化一个基本块，向下解析，此过程中：
  - 碰到 无条件跳转 语句时，结束当前基本块，将当前基本块与即将跳转的基本块连接，再新开一个基本块
  - 碰到 条件跳转 语句时，结束当前基本块，将当前基本块与即将跳转的基本块连接，再新开一个基本块，并将当前基本块与新开的基本块进行连接

    - 碰到 标签 语句时，结束当前基本块，新开一个基本块，并将当前基本块与其连接
- 连接基本块时，给每个基本块设置了 prev 和 next 指针数组来记录此基本块的前驱或后继基本块， 以便后面的数据流分析

2. 到达定义分析

到达定义分析针对的元素是**中间代码语句**，利用到达定义的信息，可以分析出：任意一条语句中使用的任意一个变量的值是在哪里定义的。具体的算法步骤可参考如下：

- 计算每个基本块的 gen 集合，由于 gen 是针对某个变量而言，因此可以参考使用 Map<String, MediLine> 的数据结构来存储，其中MediLine是一个中间代码类，一个MediLine对象对应一条中间代码，而String对应变量名。由此，基本块内每个变量名对应一个中间代码语句
- 计算每个基本块的 kill 集合
- 根据公式 $in[B] = \cup_{(B的每个前驱基本块P)} out[P]$ 和公式 $out[B] = gen[B] \cup (in[B] - kill[B])$ 计算得到每个基本块的in和out集合
- 检查上一步的out集合是否有变化，若发生变化则重复上一步，否则结束算法

进行到达定义分析后，可以做常量传播和复写传播等优化。实现时建议可以分成**基本块内部**的复写传播与常量传播、和**跨基本块**的复写传播与常量传播，具体算法由同学们自己实现。其实现效果展示如下。

假设**同一个基本块**内有中间代码：

```
// a和b是局部变量，t1是临时变量
a = 4
t1 = a * a
b = t1 / a
```

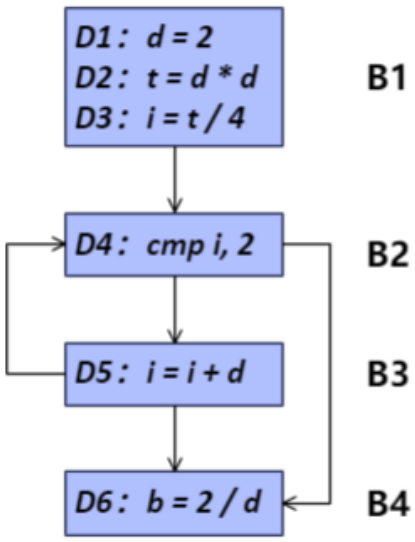
由于在第2行对变量a进行了赋值操作，其后的t1的值和b的值都可以直接计算出来，故优化后的代码为：

```
// a和b是局部变量，t1是临时变量
a = 4
b = 4
```

假设一个跨基本块的源c代码如下：

```
// b,d,i是局部变量，t是临时变量
d = 2;
t = d * d;
i = t / 4;
while (i < 2) {
  i = i + d;
}
b = 2 / d;
// ...
```

下面我们对上述代码进行基本块划分和流图构建，得到如下流图（注意！此流图省略了入口和出口处）：



做到达定义分析后可得各基本块的in和out集合为：

基本块	gen	kill	in	out
B1	{D1, D3}	{D5}	{}	{D1, D3}
B2	{}	{}	{D1, D3, D5}	{D1, D3, D5}
B3	{D5}	{D3}	{D1, D3, D5}	{D1, D5}
B4	{D6}	{}	{D1, D3, D5}	{D1, D3, D5, D6}

根据上述数据流分析信息，对每个基本块分别进行优化分析：

- B1：此基本块的in集合为空，基本块内部有D1, D2, D3三条语句。在基本块内顺序扫描，可发现语句D2中临时变量的值是由语句D1决定的，故可以将语句D1中变量d的值代入语句D2中，得到 t = 4 ；而语句D3中变量d的值是由语句D2决定的，故可以将语句D2中变量t的值代入语句D3中，得到 i = 1
- B2：此基本块的in集合中定义了变量 i 的语句有两条即D3和D5，故此处i的值不能确定，无法做常数传播
- B3：此基本块的in集合中定义了变量 i 的语句有D3和D5、定义了变量 d 的语句有D1，故此处i的值无法确定，而变量d的值可由D1得出，语句D5可优化为 i = i + 2
- B4：此基本块的in集合中定义了变量 d 的语句只有D1，故变量 d 的值可由语句D1给出，语句D6可优化为 b = 1

将每个基本块的优化结合起来，可得优化后的代码为：

```
// ...
d = 2;
i = 1;
while (i < 2) {
  i = i + 2;
}
b = 1;
// ...
```

通过上述分析，大家应该对“如何通过数据流分析做基本块内的常数传播和跨基本块的常数传播”有了初步的认识。对于上面给出的示例代码，你可能会觉得这个优化的效果并不明显，从

总行数上来看似乎只减少了一条代码。但在运算强度上，此优化减少了一条乘法语句和除法语句，而且如果这是多重循环内的除法语句，其优化效果将更显著。

这里我们给出的示例代码较为简单，而实际情况下我们面对的将是非常复杂的代码，所以在编程实现的时候还需要注意保证算法的正确性！

3. 活跃变量分析

活跃变量分析针对的元素是**变量**，算法要比到达定义数据流简单一些，具体操作步骤如下：

- 计算每个基本块的use和def集合
- 根据公式 $out[B] = \cup_{(B的每个后继基本块P)} in[P]$ 和公式 $in[b] = use[B] \cup (out[B] - def[B])$ 计算得到每个基本块的in和out集合
- 检查上一步的in集合是否有变化，若发生变化则重复上一步，否则结束算法

进行活跃变量分析后，可以做死代码删除和寄存器分配等优化。死代码删除的思想是：如果变量x在某个定义点后不再被使用，则可以将它删除。

在如下的源代码中：

```
int d = 7;
int main() {
  int i = 2, j = 5;
  i = getint();
  j = getint();
  int k = 5;
  int n = 10;
  while (n < k*k*k*k*k*k) {
    d = d * d % 10000;
    n = n + 1;
  }
  printf("%d, %d, %d\n", i, j, k);
  return 0;
}
```

进行活跃变量分析后可得知，循环体结构中对变量d进行了多次引用和赋值，但在其后的代码中都没有使用过变量d，故可以将循环体内对d的赋值语句删除。死代码删除中有两点需要特别注意：

- 变量是全局变量时不能删除
- 中间代码是函数调用或输入语句时也不能删除

4. 消除公共子表达式

在前一步骤完成后，就可以进行建立 DAG 图和从 DAG 图重新导出中间代码，实现公共子表达式删除了。

有向无环图（DAG）是实现基本块变换的一种非常有用的数据结构。DAG 图说明了基本块中每一个语句计算的值是如何被本块中的后继语句引用的。

基本块的 DAG 是一种其节点带有如下标号的无环有向图：

- 叶节点由唯一的标识符所标记，即变量名或常量。根据作用到名字上的操作符可以确定需要的是名字的左值还是右值。叶节点代表名字的初始值，为了避免与指示名字当前值的标号相混淆，我们给这些叶节点加上下标 0。
- 内部节点用操作符符号标记。

- 节点还可能为标号附以一系列标识符，目的是用内部节点表示已经计算的值，而且标记节点的标识符也具有该值。

5. 循环结构优化

如下结构的while循环语句：

```
while (cond) {
  // do...
}
```

该语句会被直观翻译成如下形式：

```
COND:
if (!cond) {
  goto THEN
}
// do...
goto COND
THEN:
```

实际上，将while语句改写成do-while语句可以有效地减少跳转次数。对于一个n次循环，一个while语句需要执行n次分支和n次跳转，但如果先将while语句转成do-while语句，则可以翻译成如下形式：

```
if (!cond) {
  goto THEN
}
BEGIN:
// do...
if (cond) {
  goto BEGIN
}
THEN:
```

这样对于一个执行了n次的循环，就只需执行n次分支，原本需经过2n+1次跳转，优化后只需n+1次跳转即可实现

6. 运算强度削弱

（1）乘法优化

乘法指令执行的周期较长，因此可以使用移位指令来代替。可以参考如下几条优化规则：

- 若乘数的绝对值为2的幂，可用一条移位指令
- 若乘数的绝对值为2的幂+1，可用一条移位指令和加法指令
- 若乘数的绝对值为2的幂-1，可用一条移位指令和减法指令
- 若乘数为负数，将结果取反即可

比如：

```
a * 4      // a << 2
a * 5      // (a << 2) + a
a * -7     // -((a << 3) - a)
```

（2）除法优化

除法指令执行的周期比乘法指令还要长，所以应尽可能用其他高效计算指令来代替除法指令。进行除法优化之前，得明确一点，只有当除数是常数的时候才可以做除法优化，如果除数是变量，则无法优化。如果除数为常量，优化效果如下：

```
a / 64     //    ((a >> 31) >>> 26 + a) >>> 6
a / 2021   //    (a + ((0x3320550555 * a) >>> 32) >> 10) + (a < 0)
```

除法优化涉及到的计算原理比较复杂，在此无法详述。对具体的优化算法和原理感兴趣的同学，可参考论文《Division by Invariant Integers using Multiplication》。

7. 寄存器分配策略

寄存器分配是整个代码优化中对性能影响最大的一个环节。我们课程组是建议大家使用规范的全局寄存器分配方法（例如引用计数或图着色算法）分配全局寄存器，并使用临时寄存器池管理和分配临时寄存器。这两种寄存器分配算法的实现，课本或课件中已有十分详细的叙述，这里不再赘述。这里笔者主要想分享一下自己的优化方案。

在mips架构中，仅仅提供了两类寄存器，即s类和t类。s类用于保存局部变量（或全局变量），t类寄存器保存临时变量。经过多次测试后，笔者认为这种分配方案并不妥当的，应当再加一类寄存器用于保存全局变量，因此总共有三类寄存器可供使用，其行为规范如下：

- 全局寄存器（跨函数）：任何时刻都可以直接调用寄存器内的值，直接对此寄存器进行操作，任何时刻都不需要写回内存
- 局部寄存器（跨基本块）：在一个函数中可以直接调用该寄存器，在此函数内任何操作都不需要写回内存，但是在调用函数时，需保存现场，压到栈里，调用结束后再恢复现场，从栈中取回
- 临时寄存器（不跨基本块）：在一个基本块内可以直接调用该寄存器，但碰到跳转语句等跨基本块行为时，需要把寄存器的值写回内存，将寄存器池清空，没有保存恢复现场的操作。

以上给出的是笔者设计的三种寄存器的使用规范。具体的寄存器分配策略，可以参考引用计数法或图着色算法来实现。寄存器分配是一个无最优解问题，因此也鼓励大家在保证算法正确性的基础上，可以在引用计数法和图着色算法的基础上进行改良，或是综合两者设计出更合理的算法。

8. 窥孔优化

逐条语句进行的代码生成策略经常产生含有大量冗余指令和次优结构的目标代码。 通过对这些目标代码进行“优化”转换可以提高这种目标代码的质量。有许多简单的转换可以显著地改善目标程序的运行时间和空间需求，所以知道哪种转换在实际中是非常重要的。

窥孔优化是一种简单有效的局部优化方法，它通过检查目标程序的短序列（称为窥孔），并尽可能用更小更短的指令序列代替这些指令以提高目标程序的性能。尽管我们将窥孔优化作为改善目标代码质量的技术，它也可以直接用在中间代码生成之后以提高中间代码的质量。

以删除冗余代码为例，递归产生四元式时，对于赋值语句，是先求出右表达式的值，再把此值赋值给左值：

```
// 源代码
a = b + c;

// 中间代码
t = b + c;
a = t;
```

可以看到上面代码中国的临时变量t的过渡是冗余的，可以在中间代码中进行替换，将过渡的中间代码删除。 对于输入语句，输出语句，返回值语句等，同理也可以进行优化 。

当然除了上述几点优化，还有函数内联、SSA等优化方法，感兴趣的同学可以自行查阅相关资料。

9. 笔者的一些心得

代码优化是编译器设计中的重难点，也是一个开放性的设计问题。笔者在反复重构中间代码的过程中，更体会到了**设计优先**的原则：**用于设计的精力一定要大于编码的精力！！！** 由于时间有限，笔者也没能把课本上的优化全都做一遍，这里仅就笔者实现的优化谈几点认识：

- 优化效果是叠加生效的：很多优化方案不是说做了一个优化就一定能达到效果，是需要**多个优化方案结合**起来才能发挥出威力。比如常量传播看似没有很大的功效，但它对于寄存器分配方案，死代码删除，以及除法优化等，都有不可小觑的作用
- 寄存器分配方案无最优解：笔者对比了图着色算法和改进的引用计数法两种方案，发现两者的作用效果差不多，对于某些测试点，改进的引用计数的效果甚至比图着色算法好。原因在于图着色算法可以降低循环上访问内存的开销，但容易增大在函数调用上的开销。因此同学们可以在引用计数法和图着色算法的基础上进行改良，大胆地去设计和尝试

结语

至此为止，编译器的课程设计就算是基本完成了，希望本手册能对读者有所帮助。笔者也是去年才刚刚完成编译器课程设计的学生，水平有限，书中难免有各种错漏之处，希望读者不吝指教。