

合肥工业大学

操作系统课程设计报告

设计题目	操作系统接口：Windows 命令接口 3 (1 人，难度：5)
学生姓名	刘辰驭
学 号	2021217189
专业班级	计科 21-4 班
指导教师	田卫东
完成日期	2024.1.5

1. 课程设计任务、要求、目的

1.1 课程设计任务

为 Unix/Linux 操作系统建立兼容的 Windows/DOS 命令接口，批处理文件中的流程命令；

批处理文件的扩展名为 .BATCH

具体命令：IF, FOR, WHILE, SET, SHIFT, ECHO, GOTO, : (标号)，命令格式可参照 Windows 的 CMD.EXE 或 MS-DOS 提供的命令；

设计命令的名称、参数等格式。

可以执行扩展名为 .BATCH 的批处理文件。

1.2 课程设计目的和要求

目的：为了在 Unix/Linux 操作系统上建立一个兼容的 Windows/DOS 命令接口，并实现对批处理文件中的流程命令的支持。通过这个课程设计，能够编写和执行扩展名为 .BATCH 的批处理文件，使用类似于 Windows 的 CMD.EXE 或 MS-DOS 提供的命令。

要求：

1. 支持批处理文件的执行：实现对扩展名为 .BATCH 的批处理文件的解析和执行功能。

2. 实现特定的命令：需要实现一系列的命令，包括 IF、FOR、WHILE、SET、SHIFT、ECHO、GOTO 和 : (标号) 等。这些命令应该具有与 Windows 的 CMD.EXE 或 MS-DOS 提供的命令相似的参数格式和行为。

2. 开发环境

操作系统：Windows 11

编译器：Dev-C++

编译器：Visual studio 2019

操作系统：linux

虚拟机：Vmware Workstation Pro

CentOS-7.5-x86_64

编译器：GCC 编译器

3. 相关原理及算法

相关原理：

为了使用户能方便地使用计算机，操作系统提供了相应的用户接口，帮助用户快速、有效、安全、可靠地操纵计算机系统内的各类资源，完成相关的处理。一般地，操作系统向用户提供了两类接口，即用户接口和程序接口。在本实验中要求实现的是批处理方式的联机命令接口。

批处理方式的联机命令接口：在操作命令的实际使用过程中，经常遇到需要对多条命令的连续使用、或对若干条命令的重复使用、或对不同命令进行选择性的使用，如果用户每次都采用命令行方式将命令一条条由键盘输入，既浪费时间，又容易出错。因此，操作系统都支持一种称为批命令的特别命令方式，允

许用户预先把一系列命令组织在一种称为批命令文件的文件中，一次建立，多次执行。使用这种方式可减少用户输入命令的次数，既节省了时间，减少了出错概率，又方便了用户。通常批命令文件都有特殊的文件扩展名（在本实验中扩展名为 *BATCH*）。

4. 系统结构和主要的算法设计思路

首先设计 *MS-DOS* 联机命令。

MS-DOS 联机命令简介（主要介绍实验中要求的 *IF*、*FOR*、*WHILE*、*SET*、*SHIFT*、*ECHO*、*GOTO* 和 *:*（标号）命令）：

1. *if* 命令：

if 命令的基本语法如下：

if [*NOT*] 条件 *commandA* [*else commandB*]

其中，条件是要被检查的条件语句，如果条件成立（有 *NOT* 则不成立），则执行 *commandA* 命令。如果条件不成立（有 *NOT* 则成立），则执行 *commandB* 命令（如果有 *else* 的话）。

下面是本实验实现的一些常用的 *if* 命令用法：

(1) 查看文件是否存在：

if exist 文件路径 *command*

如果文件存在，则 执行 *command*。

例如：

if exist C:\\Windows\\System32\\cmd.exe echo cmd.exe 存在

(2) 判断字符串是否相等：

if string1==string2 command

string1==string2 如果指定的文字字符串匹配，则执行 *command*。

2. *for* 命令：

for 命令的基本语法如下：

for 条件 *do command*

如果 *for* 循环条件成立，则执行 *command* 命令，直到循环条件不成立退出循环。

下面是本实验实现的一些常用的 *for* 命令用法：

(1) *for %%variable in (set) do command*

%%variable 代表可替换的参数。*for* 命令使用在 *set* 中指定的每个文本字符串（用 *'*，*'* 分隔）替换 *%%variable*，并执行 *command*，直到取完所有的文本字符串为止，结束循环。

例如：

for %%j in (apc,bsd,caq) do set \p pl = %%j

环境变量的值将分别被置为 *apc*，*bsd*，*caq*。

(2) *for /d %%variable in (start,step,end) do command*

for 命令使用以 *start* 为开始，*end* 为结束，*step* 为步长的数字

替换%%variable，并执行 command，直到取到结尾为止，结束循环。

例如：

```
for /d %%j in (1,1,5) do echo %%j
```

输出结果为 1 2 3 4 5。

(3) `for /f %%variable in (文件路径) do command`

for 命令使用文件替换 variable，并执行 command，直到文件读取到末尾，结束循环。

(4) `for /f "skip=a tokens=b,c delims=' p' " %%variable in (文件路径) do command`

for 命令跳过文件前 a 行，取每行 token 指定位置的字符串，以 delims 指定的 ' p' 为分隔符（缺省为 ' ' ），替换 variable，并执行 command，直到文件读取到末尾，结束循环。

3. while 命令：

MS-DOS 未指定 while 命令的实现方式，可以使用 if 语句，goto 语句与标号实现 while 循环。

4. set 命令：

set 命令的基本语法如下：

```
set [\a] [variable [= [var]]]
```

set 命令是用于显示、设置、或清除特定变量的命令，方便用户管理系统的环境变量。

下面是本实验实现的一些常用的 set 命令用法：

(1) `set`

输出结果将会是当前系统中所有的环境变量名及其当前值的列表。

```
C:\Users\pegiions>set
ADSK_3DSMAX_x64_2023=D:\3DSMAX 2023\3ds Max 2023\
ALLUSERSPROFILE=C:\ProgramData
APPDATA=C:\Users\pegiions\AppData\Roaming
CATALINA_HOME=C:\Program Files\apache-tomcat-8.0.50
CCHZPATH=D:\CTEX\CTeX\cct\fonts
```

(2) `set /p variable = value`

设置一个新的字符串型环境变量 variable，值为 value（字符串）

```
C:\Users\pegiions>set /p x = apple
apple|
```

(3) `set /a variable = calculate`

设置一个新的数值型环境变量 variable，将数学计算公式 calculate 计算出的值赋给 variable

```
C:\Users\pegiions>set /a y = 2+3
5
```

(4) `set pre`

显示以 *pre* 为前缀的所有环境变量

```
C:\Users\pegions>set user
USERDOMAIN=LAPTOP-T6STB670
USERDOMAIN_ROAMINGPROFILE=LAPTOP-T6STB670
USERNAME=pegions
USERPROFILE=C:\Users\pegions
```

- (5) *set variable =*
删除环境变量 *variable*

5. *shift* 命令:

shift 命令的基本语法如下:

shift [/n]

调整批处理文件中可替换参数的位置。从第 *n* 个参数开始, 将第 *n*+1 个参数移位至 *n*, 第 *n*+2 个参数移位至 *n*+1, 以此类推。

使用 DOS 窗口查看 *shift* 命令使用手册:

```
C:\Users\pegions>shift /?
更改批处理文件中可替换参数的位置。

SHIFT [/n]

如果命令扩展被启用, SHIFT 命令支持 /n 命令行开关; 该命令行开关告诉命令从第 n 个参数开始移位; n 介于零和八之间。例如:

SHIFT /2

会将 %3 移位到 %2, 将 %4 移位到 %3, 等等; 并且不影响 %0 和 %1。
```

6. *echo* 命令:

echo 命令的基本语法如下:

echo string [>>> 文件路径]

用于实现在 DOS 窗口打印字符串 *string*, 或者将 *string* 写入文件 (>), 或者将 *string* 追加在文件末尾 (>>).

下面是本实验实现的一些常用的 *echo* 命令用法:

- (1) *echo string*

实现在 DOS 窗口打印字符串 *string*.

```
C:\Users\pegions>echo hello world
hello world
```

- (2) *echo string > 文件路径*

将 *string* 写入文件。

- (3) *echo string >> 文件路径*

将 *string* 追加在文件末尾。

- (4) *echo %var%*

输出环境变量 `var` 的值

```
C:\Users\pegions>echo %username%  
pegions
```

7. `goto` 命令与标号:

`goto` 命令与标号的基本语法如下:

```
: label  
goto label
```

在批处理文件执行过程中遇到 `goto label`, 则跳转到批处理文件 `label` 所在行开始执行批处理文件。

使用 DOS 窗口查看 `goto` 命令使用手册:

```
C:\Users\pegions>goto /?  
将 cmd.exe 定向到批处理程序中带标签的行。  
  
GOTO label  
  
label    指定批处理程序中用作标签的文字字符串。  
  
标签必须单独一行, 并且以冒号打头。
```

8. `copy` 命令:

本实验在原实验要求上扩展了一个 `copy` 命令以展示语句的可嵌套调用。

本实验实现了一条 `copy` 语句:

```
copy 程序路径 1 程序路径 2
```

该语句会将文件 1 `copy` 到文件 2

然后是联机命令接口的实现: MS-DOS 解释程序

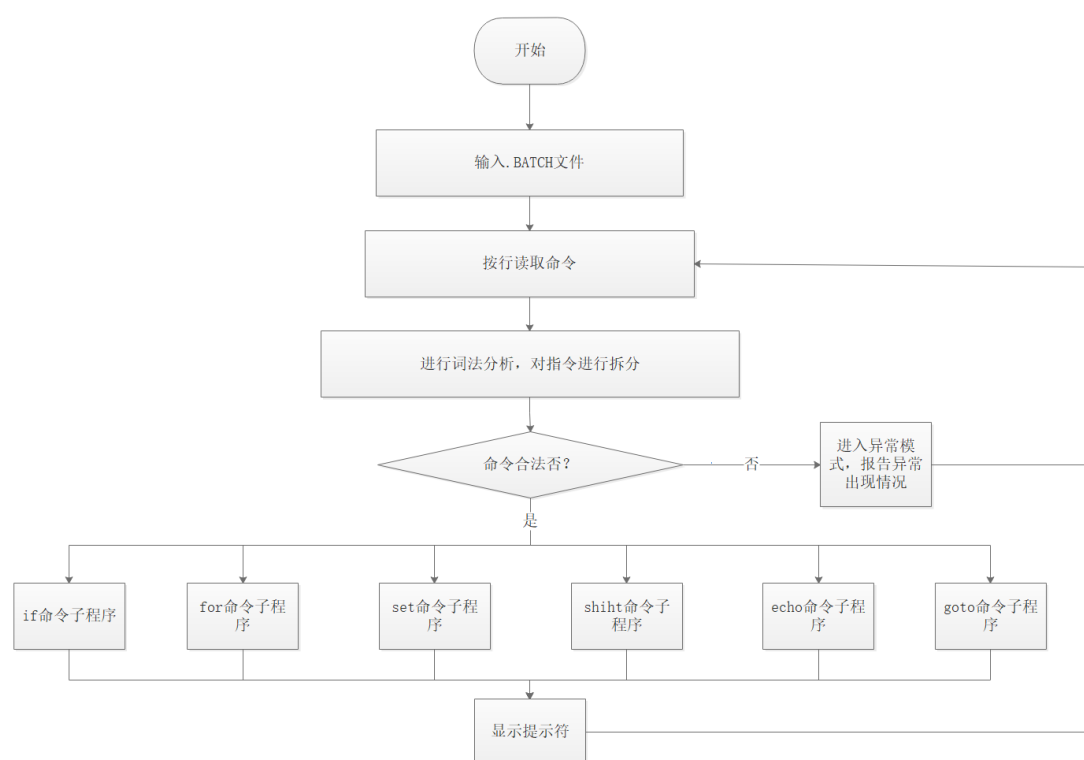
MS-DOS 解释程序相当于一个批处理文件编译器, 在批处理操作方式下, 终端处理程序把用户键入的 DOS 命令保存在 `BATCH` 文件中, 随后控制权交给命令解释程序, 随后 MS-DOS 解释程序读入该 `BATCH` 文件, 识别命令, 再转到相应命令处理程序的入口地址, 把控制权交给该处理程序去执行, 并将处理结果送至屏幕上显示。若用户键入的命令有错, 而命令解释程序未能予以识别, 或在执行终结出现问题时, 则应显示出某一出错信息。

所以 MS-DOS 解释程序应具备以下功能:

1. 命令解析: 当用户在命令行中输入一个命令时, MS-DOS 解释程序会解析该命令并确定要执行的操作。它会识别命令的名称、参数和选项, 并验证命令的语法和正确性。
2. 命令执行: 一旦命令被解析, MS-DOS 解释程序会根据命令的类型和参数调用相应的操作系统程序或执行相应的操作。这涉及到调用内部命令(在本实验中包含 `IF`, `FOR`, `WHILE`, `SET`, `SHIFT`, `ECHO`, `GOTO`, `:` (标号) 以及扩展的一个 `COPY` 命令)。

3. 环境变量管理: MS-DOS 解释程序还负责管理和操作环境变量。环境变量是操作系统中存储和传递配置信息的一种机制。MS-DOS 解释程序允许用户设置、获取和修改环境变量的值,并在命令执行过程中使用这些值。
4. 错误处理: MS-DOS 解释程序能够检测和处理命令执行过程中的错误和异常情况。它可以提供错误消息和适当的错误代码,以使用户了解发生了什么问题,并采取适当的措施。

本实验设计的 MS-DOS 解释程序流程图如下:



设计思路: 设计 MS-DOS 解释程序实际上就是设计一个编译器, 对 MS-DOS 批处理命令进行词法分析、语法分析、最终解释执行。

主要算法包括以下几个方面:

1. 词法分析

词法分析是将输入字符串按照符号表分割成一系列的 *tokens* (词素), 如关键字、标识符、符号等。对 MS-DOS 批处理命令进行词法分析主要识别:

关键字: 如 IF、FOR、GOTO 等命令字

标识符: 如变量名

符号: 如=、*comparision* 运算符等

数字和字符串常量

词法分析器使用有限状态自动机实现词法规则, 并返回各种 *tokens* 给后续步骤处理。

2. 语法分析

语法分析检查 *tokens* 是否遵循正确的语法结构。对 MS-DOS 批处理语法进行上下文无关分析可以构建一个抽象语法树。

3. 解释执行

解释器按序执行生成的中间代码, 包含如变量存储管理、流程控制等。

这样利用上述模块和算法实现了一个基础的 MS-DOS 解释器。其中词法和语法分析是核心, 正确识别语法元素和结构是保证正确执行的基础。语义阶段生成中间代码为后续解释准备。这套流程能够很好地解析和运行批处理程序。

5. 程序实现——主要数据结构

主要实现的数据结构包含以下四种, 存放指令的数据结构 *struct A*, 存放标号的数据结构 *C_int*, 存放整数型环境变量的数据结构 *B_int* 和存放字符串型环境变量的 *B_char*。

数据结构	功能
<i>A</i>	存放拆分前的指令, 和拆分后的指令
<i>C_int</i>	存放标号
<i>B_int</i>	存放整数型环境变量
<i>B_char</i>	存放字符串型环境变量

(1) 存放指令的数据结构 *struct A*

```
struct A{
    char sentence[100];
    char *mystr[20]; //存储每个字符（按照空格分开）
    int numcount;
};
其中 char sentence[100] 为 .BATCH 文件按行读取的指令字符串。
char *mystr[20] 为指令经过词法分析和语法分析后分割成的字符串数组。
numcount 用于指令的计数，表示是第几条指令。
```

(2) 存放标号的数据结构 *C_int*

```
struct C_int{
    int flag;
    char contant[10];
};
```

(3) 存放整数型环境变量的数据结构 *B_int*

```
struct B_int{
    char flag[30];
    char contant[30];
};
```


(4) 存放字符串型环境变量的 *B_char*

```
struct B_char{
    char flag[30]; //未知数标号
    char contant[30];
};
```

6. 程序实现——程序实现细节描述

本实验设计的主要函数如下表：

函数定义	功能
<i>select</i>	词法分析
<i>action</i>	词法分析
<i>del_char</i>	删除指定字符
<i>seperate</i>	词法分析，分割字符串
<i>int_tran</i>	数值型环境变量数组初始化
<i>func_shift_ori</i>	标号表初始化
<i>str_cut</i>	删除字符串头尾
<i>find_charater</i>	查找指定环境变量
<i>char_tran</i>	数字转制
<i>str_judge</i>	前缀匹配(实现 <i>set</i> 前缀匹配查找变量)
<i>func_set</i>	<i>set</i> 子程序
<i>func_if</i>	<i>if</i> 子程序
<i>func_if_exist</i>	<i>if</i> 子程序的 <i>exist</i> 功能
<i>find_if_doc</i>	<i>if</i> 子程序查找文件路径功能
<i>func_if_char</i>	<i>if</i> 子程序实现字符串匹配==功能
<i>func_for</i>	<i>for</i> 子程序
<i>func_for_s</i>	<i>for</i> 子程序实现功能 <i>for</i> %i in (A,B,C) do
<i>func_for_d</i>	<i>for</i> 子程序实现功能 <i>for</i> %i in (1, 1, 5) do
<i>func_for_f</i>	<i>for</i> 子程序实现功能 <i>for</i> /f %c in (d:\abc.txt) do <i>for</i> /f "skip=1 tokens=1,4 delims= " %c in (d:\abc.txt) do
<i>seperate_delim</i>	获取所有分隔符 <i>delim</i>
<i>func_label</i>	实现 <i>label</i> 标号功能
<i>get_ch_num</i>	计算 <i>label</i> 标号的数量
<i>func_goto</i>	<i>goto</i> 子程序
<i>func_shift</i>	<i>shift</i> 子程序
<i>func_copy</i>	<i>copy</i> 子程序
<i>func_echo</i>	<i>echo</i> 子程序

介绍指令拆分与词法分析函数实现:

1. **取指令:** 文件中读取指令内容, 并对读取的内容进行处理和分析:
 - (1) 使用 `fopen` 函数打开 `BATCH` 文件, 以只读模式打开。
 - (2) 如果文件成功打开, 代码进入一个 `do-while` 循环, 循环条件为文件读取至结尾 (`feof(fp)`)。在循环内部, 使用 `fgets` 函数逐行读取文件内容, 并将读取的内容存储到 `buff` 数组中。
 - (3) 通过 `strcpy` 函数将 `buff` 数组中的内容复制到存储指令的结构体 `A` 数组 `word` 的 `sentence` 成员中, 然后使用 `del_char` 函数删除句子中的换行符。接着, 调用 `seperate` 函数对指令进行拆分处理和分析, 将分析结果存储到 `word` 的 `mystr` 成员中。最后, 递增 `count` 计数器, **至此取出一条指令**。
 - (4) 循环结束后调用 `func_shift_ori` 函数初始化标号数组, 至此指令取出完毕。
 - (5) 最后, 使用 `fclose` 函数关闭打开的文件指针 `fp`, 并调用 `select` 函数对读取的内容进行处理和选择, **进入词法分析阶段**。
2. **词法分析:** 函数 `select` 和 `action`, 对指令进行词法分析, 送入对应的处理子程序:
 - (1) 如果 `word[i].mystr[0]` 的值与字符串 `"echo"` 相等, 执行 `func_echo` 函数, 进入 `echo` 处理子程序。
 - (2) 如果 `word[i].mystr[0]` 的值与字符串 `"set"` 相等, 执行 `func_set` 函数, 进入 `set` 处理子程序。
 - (3) 如果 `word[i].mystr[0]` 的值与字符串 `"for"` 相等, 执行 `func_for` 函数, 进入 `for` 处理子程序。
 - (4) 如果 `word[i].mystr[0]` 的值与字符串 `"if"` 相等, 执行 `func_if` 函数, 进入 `if` 处理子程序。
 - (5) 如果 `word[i].mystr[0]` 的值与字符串 `"shift"` 相等, 执行 `func_shift` 函数, 进入 `shift` 处理子程序。
 - (6) 如果 `word[i].mystr[0]` 的值与字符串 `":"` 相等, 执行 `func_label` 函数, 该函数用于处理标号的操作。
 - (7) 如果 `word[i].mystr[0]` 的值与字符串 `"goto"` 相等, 执行 `func_goto` 函数, 进入 `goto` 处理子程序。
 - (8) 如果 `word[i].mystr[0]` 的值与字符串 `"copy"` 相等, 执行 `func_copy` 函数, 进入 `copy` 处理子程序。

```
if (!strcmp(word[i].mystr[0], "echo")) {  
    //printf("in echo");  
    func_echo(word, i, store_str, store_num, number_str, number_num, charater_num);  
}  
if (!strcmp(word[i].mystr[0], "set")) func_set(word, i, store_str, store_num, number_str, number_num);  
//for 的各种职能, 在for_if 中进行实现  
if (!strcmp(word[i].mystr[0], "for")) {  
    func_for(word, i, store_str, store_num, number_str, number_num, charater_num);  
}
```

下面介绍实现 `IF`、`FOR`、`SET`、`SHIFT`、`ECHO`、`GOTO`、`COPY` 和 `:` (标号) 命令)

的子程序关键函数细节:

1. echo 处理子程序 func_echo, 实现 echo 功能:

(1) echo string > 文件路径 命令:

检查 `word[line].mystr[word[line].numcount-2]` 的值是否与字符串 ">" 相等。如果相等, 表示需要将输出写入文件。代码使用 `fopen` 函数打开指定的文件, 以写入模式打开。然后, 通过循环将 `word[line]` 中的内容拼接为一个字符串 `write`, 并使用 `fputs` 函数将字符串写入文件中。最后, 关闭文件并返回函数。

(2) echo string >> 文件路径 命令:

如果 `word[line].mystr[word[line].numcount-2]` 的值与字符串 ">>" 相等, 表示需要将输出追加到文件末尾。代码使用 `fopen` 函数打开指定的文件, 以追加模式打开。然后, 通过循环将 `word[line]` 中的内容拼接为一个字符串 `write`, 并使用 `fputs` 函数将字符串追加到文件中。最后, 关闭文件并返回函数。

(3) 如果以上两个条件都不满足, 说明是输出命令。进入循环, 循环变量 `i` 从 1 开始遍历 `word[line].mystr` 中的元素。

(4) echo %var% 命令:

如果字符串 `ch2` 以 "%" 开头且以 "%" 结尾, 表示需要显示环境变量的值。代码使用 `str_cut` 函数截取 `ch2` 中的字符, 并将截取结果赋值给 `ch1`。然后, 使用 `find_charater` 函数在环境变量数组中查找与 `ch1` 匹配的环境变量, 并将结果赋值给 `val`。如果找到环境变量的值, 使用 `printf` 函数输出该值; 否则, 直接输出字符串 `ch2`。

(5) 如果字符串 `ch2` 以 "%" 开头, 表示需要显示标号变量的值。代码将 `ch2[1]` 转换为整数, 并将转换结果赋值给变量 `j`。然后, 使用 `charater_num` 数组中的元素 `charater_num[j].contant` 来输出标号变量的值。

(6) echo string 命令:

如果以上两个条件都不满足, 则说明是打印输出命令, 使用 `printf` 函数直接输出字符串 `word[line].mystr[i]`。

2. set 处理子程序 func_set, 实现 set 功能:

(1) set 命令:

检查 `word[line].numcount` 的值是否为 1。如果是 1, 表示是 **set 命令**。通过循环遍历环境变量数组, 使用 `printf` 函数输出每个环境变量的键和值。

(2) set pre 命令:

如果 `word[line].numcount` 的值为 2，表示需要输出以指定前缀开头的环境变量的值。使用 `str_judge` 函数判断环境变量的键是否以指定前缀 `judgestr` 开头。如果是，使用 `printf` 函数输出该环境变量的键和值。如果没有找到匹配的环境变量，进入异常模式输出异常信息。

(3) **set var = 命令:**

如果 `word[line].numcount` 的值为 3，表示需要删除指定的环境变量。使用 `strcmp` 函数比较环境变量的键与 `word[line].mystr[1]` 是否相等。如果相等，将后面的环境变量向前移动一位，并将环境变量的数量减一。

(4) **set /a variable = var 命令:**

需要添加新的环境变量。检查 `word[line].mystr[1]` 的值是否为 `"/a"`，如果是，表示需要添加整数类型的环境变量。代码将 `word[line].mystr[2]` 赋值给 `store_num[*number_num].flag` 作为环境变量的键，将 `word[line].mystr[4]` 赋值给 `store_num[*number_num].contant` 作为环境变量的值。然后，使用 `printf` 函数输出新添加的环境变量的键和值，并将键和值写入文件 `"new.txt"` (环境变量存储文件)。最后，将 `(*number_num)` 加一。

(5) **set /p variable = var 命令:**

如果 `word[line].mystr[1]` 的值是 `"/p"`，表示需要添加字符串类型的环境变量。代码将 `word[line].mystr[2]` 赋值给 `store_str[*number_str].flag` 作为环境变量的键，将 `word[line].mystr[4]` 赋值给 `store_str[*number_str].contant` 作为环境变量的值。然后，使用 `printf` 函数输出新添加的环境变量的键和值，并将键和值写入文件 `"new.txt"`。最后，将 `(*number_str)` 加一。

3. if 处理子程序 `func_if`，实现 if 功能

(1) 首先声明了一些变量，包括 `flag`、`str` 和 `str_eq`。`flag` 用于表示条件是否取反，`str` 用于存储条件表达式中的第一个字符串，`str_eq` 用于存储条件表达式中的第二个字符串。

(2) 使用 `strcpy` 函数将 `word[line].mystr[1]` 的值复制给 `str`，将 `word[line].mystr[2]` 的值复制给 `str_eq`。

(3) **if not 命令:**

如果 `str` 的值与字符串 `"NOT"` 相等，表示条件需要取反。将 `flag` 设为 1，并将 `str` 的值更新为 `word[line].mystr[2]`，将 `str_eq` 的值更新为 `word[line].mystr[3]`。

(4) *if exist* 命令:

如果 *str* 的值与字符串 "exist" 相等, 表示条件为判断环境变量是否存在。调用 *func_if_exist* 处理函数:

func_if_exist 函数: 使用递归实现了 *if* 语句嵌套的功能, 同样 *for* 语句也使用了类似的递归实现了嵌套的功能。

- a. 声明了一些变量, 包括 *braket*、*word1* 和 *word2*。*braket* 用于记录括号的数量, *word1* 和 *word2* 是结构体数组, 用于存储括号内的语句。
- b. 通过一个循环遍历 *word[line].mystr* 数组, 从索引 *3+flag* 开始。循环用于构造 *word1* 和 *word2*, 即括号内的语句。当遇到 "(" 时, *braket* 加 1, 然后从当前位置的下一个索引开始遍历, 直到遇到 ")" 为止。根据 *braket* 的值, 将遍历到的字符串存储到 *word1* 或 *word2* 中。
- c. 如果条件表达式的返回值 (通过调用 *find_if_doc* 函数) 与 *flag* 进行异或操作的结果为真, 表示条件满足。代码递归调用 *select* 函数, 传递 *word1* 作为参数, 来处理括号内的语句。
- d. 如果条件表达式的返回值与 *flag* 进行异或操作的结果为假, 表示条件不满足。如果 *braket* 的值为 2, 表示存在第二个括号内的语句。代码调用 *select* 函数, 传递 *word2* 作为参数, 来处理第二个括号内的语句。

(5) *if string1 == string2* 命令:

如果 *str_eq* 的值与字符串 "==" 相等, 表示条件为判断环境变量的值是否相等。代码调用 *func_if_char* 函数来处理这种情况, 传递相应的参数, 并直接返回。

func_if_char 函数:

func_if_char 函数同样使用递归实现了 *if* 语句嵌套的功能。

- a. 声明了一些变量, 包括 *braket*、*word1* 和 *word2*。*braket* 用于记录括号的数量, *word1* 和 *word2* 是结构体数组, 用于存储括号内的语句。
- b. 通过一个循环遍历 *word[line].mystr* 数组, 从索引 *4+flag* 开始。循环用于构造 *word1* 和 *word2*, 即括号内的语句。当遇到 "(" 时, *braket* 加 1, 然后从当前位置的下一个索引开始遍历, 直到遇到 ")" 为止。根据 *braket* 的值, 将遍历到的字符串存储到 *word1* 或 *word2* 中。
- c. 通过比较 *word[line].mystr[1+flag]* 和 *word[line].mystr[3+flag]* 的值来判断条件表达式是否为真。如果两个字符串相等且 *flag* 为 0, 或者两个字符串不相等且 *flag* 为 1, 表示条件满足。代码调用 *select* 函数, 传递 *word1* 作为参数, 来处理括号内的语句。
- d. 如果条件表达式不满足, 并且 *braket* 的值为 2, 表示存在第二个括号内的语句。代码调用 *select* 函数, 传递

word2 作为参数，来处理第二个括号内的语句。

4. for 处理子程序 func_for, 实现 for 功能

(1) 声明了一个字符数组 str, 并将 word[line].mystr[1] 的值复制给 str。这个字符串用于判断循环类型。

(2) *for %%j in (apc,bsd,caq) do set \p p1 = %%j 命令:*
如果 str[0] 的值为 '%', 表示字符循环。代码调用 func_for_s 函数, 传递 word、line 以及其他参数, 来处理字符循环。

(3) *for /d %%variable in (start,step,end) do command 命令:*
for /f %%variable in (文件路径) do command 命令:
for /f " skip=a tokens=b,c delims=' p' " %%variable in (f2.txt) do command 命令:

如果 str[0] 的值为 '/', 表示数字循环。根据 str[1] 的值进行判断:

a. 如果 str[1] 的值为 'd', 表示进行数字循环。代码调用 func_for_d 函数, 传递 word、line 以及其他参数, 来处理数字循环。

b. 如果 str[1] 的值为 'f', 表示进行文件循环。代码调用 func_for_f 函数, 传递 word、line 以及其他参数, 来处理文件循环。处理完文件循环后, 函数通过 return 语句提前结束。

func_for_s 函数处理字符循环:

(1) 通过 str_cut 函数去除 word[line].mystr[3] 中的括号, 并将结果复制给 strspilit。

(2) 使用循环遍历 strspilit 中的字符, 当遇到逗号, 时, 表示启动一个新的字符串, 更新 index 的值。否则, 将字符存储到 str[index] 中, 并更新 stri[index] 的值。

(3) 构造了一个名为 word1 的结构体数组, 设置其 numcount 属性为 0, 用于构造循环中的 word 参数。

(4) 通过循环遍历 word[line].mystr 数组, 从索引 5 开始。如果当前字符串与 word[line].mystr[1] 相等, 说明是要替换的变量, 将 flag 设置为 word1 中变量的位置。无论是否相等, 都将当前字符串存储到 word1 中, 并更新 word1[0].numcount 的值。

(5) 通过循环遍历 str 中的字符串进行循环操作。如果 flag 不等于 0, 说明有变量需要替换, 将 str[i] 替换到 word1 的相应位置。

(6) 调用 select 函数, 传递 word1 作为参数, 来处理循环内的语句。

这里使用递归调用 `select` 函数，实现嵌套的循环操作。

`func_for_d` 函数处理数字循环：

- (1) 将 `strspilit` 中的字符转换为数字，并分别存储到 `num` 数组中。
`num[0]` 表示起始值，`num[1]` 表示步长，`num[2]` 表示终止值。
- (2) 构造了一个名为 `word1` 的结构体数组，设置其 `numcount` 属性为 0，用于构造循环中的 `word` 参数。
- (3) 通过循环遍历 `word[line].mystr` 数组，从索引 6 开始。如果当前字符串与 `word[line].mystr[2]` 相等，说明是要替换的变量，将 `flag` 设置为 `word1` 中变量的位置。无论是否相等，都将当前字符串存储到 `word1` 中，并更新 `word1[0].numcount` 的值。
- (4) 接下来，通过循环遍历数字循环范围内的值进行循环操作。每次循环增加 `num[1]` 作为步长，直到达到或超过终止值 `num[2]` 为止。
- (5) 如果 `flag` 不等于 0，说明有变量需要替换。代码创建一个名为 `change` 的字符数组，并将当前循环的值转换为字符存储到 `change` 中。然后将 `change` 替换到 `word1` 中相应的位置。
- (6) 调用 `select` 函数，传递 `word1` 作为参数，来处理循环内的语句。这里使用递归调用 `select` 函数，实现嵌套的循环操作。

`func_for_f` 函数处理文件循环：

- (1) 通过 `strcpy` 函数将 `word[line].mystr[2]` 的值复制给 `ch3`，用于判断是哪种类型的文件操作。
- (2) 如果 `ch3` 的第一个字符是 `%`，则执行第一种文件操作类型。代码通过 `strcpy` 将 `word[line].mystr[4]` 的值复制给 `chr`，用于获取文件名。然后打开文件，读取文件内容并逐行输出。
- (3) 如果 `ch3` 的第一个字符是 `"`，则执行第二种文件操作类型。代码通过 `strcpy` 将 `word[line].mystr[9]` 的值复制给 `chr`，用于获取文件名。
- (4) **重点：**通过循环和条件判断获取 `skip` 值，即要跳过的行数。
通过循环和条件判断获取 `tokens` 值，即要提取的字段位置。将每个位置存储在 `token` 数组中。
通过条件判断获取 `delim` 值，即分割方式。如果 `delimain` 的长度为 7，则使用空格分割；否则，取 `delimain` 的第 8 个字符作为分割方式。
- (5) 打开文件，并读取文件的每一行内容。在读取的过程中，跳过前

面的 skip 行。

在每一行内容上进行分割操作。首先，将读取的行内容复制到 word1[0].sentence 中，并通过 del_char 函数去除换行符。然后，通过 seperate_delim 函数使用 delim 作为分隔符对该行内容进行分割，结果存储在 word1[0] 中。

循环遍历 token 数组中的位置，并输出相应的字段内容。

5. Label 处理子程序 func_label, 实现标号功能

:label 命令:

- (1) 函数调用了 get_ch_num 函数，将返回值赋给变量 chnum。
get_ch_num 函数用于获取当前已存储标签数量。
- (2) 通过 strcpy 函数，将 word[line].mystr[1] 的内容复制到 charater_num[chnum].contant 中，即将标签存储起来。
- (3) 将变量 line 的值赋给 charater_num[chnum].flag，用于记录标签所在的指令位置在 word 中的索引（第几条指令）。

6. goto 处理子程序 func_goto, 实现 goto 功能

- (1) 定义了变量 i 和字符数组 ch，并通过 strcpy 函数将 word[line].mystr[1] 的内容复制到 ch 中，即获取要跳转的标签。
- (2) 定义变量 pc（命令指针）并初始化为 0，用于存储要跳转到的位置。
- (3) 调用 get_ch_num 函数获取当前存储的标签数量，并将返回值赋给变量 chnum。
- (4) 使用循环遍历 charater_num 数组，查找与 ch 相匹配的标签，并将对应的指令位置存储在变量 pc 中。
- (5) 通过 for 循环，从 pc+1 处开始重新顺序执行指令，循环范围是从 pc+1 到 count-1。
在每次循环中，调用 action 函数，传递相应的参数，重新顺序执行指令。

7. shift 处理子程序 func_shift, 实现 shift 功能

shift [/n] 命令:

- (1) 首先定义变量 temp 和 sh_time。temp 通过 word[line].mystr[1][0] 获取第一个字符，即移位次数的字符表示。sh_time 通过将 temp 减去字符 '0' 的 ASCII 码值得到移位次数的整数表示。
- (2) 使用 for 循环，从移位次数 sh_time 开始到索引 9 进行迭代。
在每次循环中，通过 strcpy 函数将 charater_num[i+1].contant

的内容复制到 `charater_num[i].contant` 中。这样，后一个参数的内容会覆盖前一个参数的内容，实现了参数的移位操作。

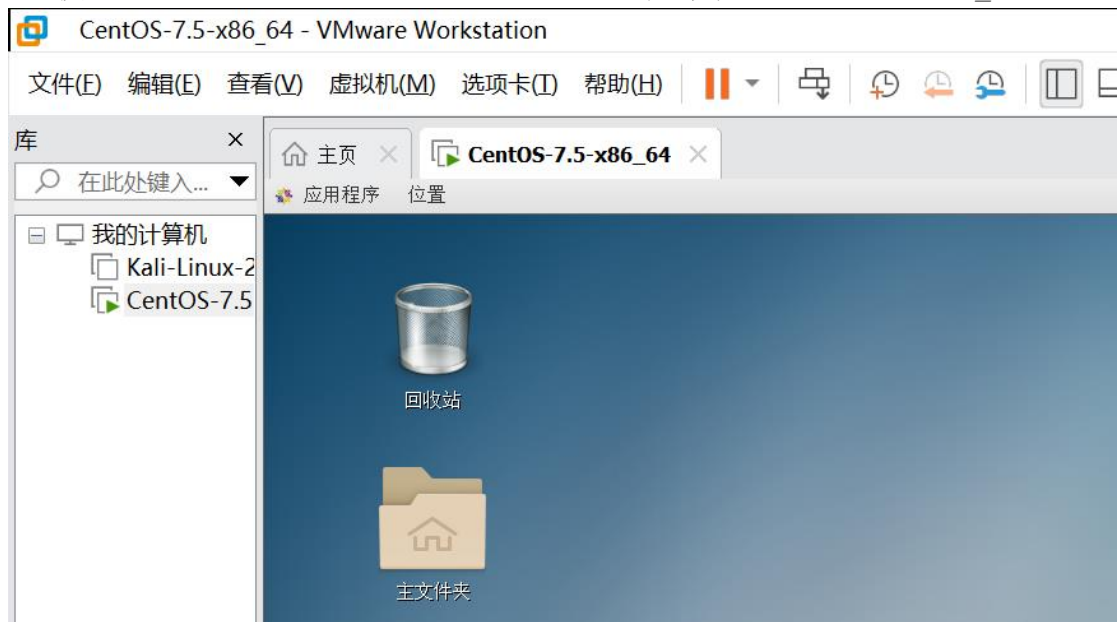
8. *copy* 处理子程序 *func_copy*，实现 *copy* 功能

copy a.txt b.txt 命令：

- (1) 定义了两个 `FILE` 指针变量 `fp1` 和 `fp2`，分别用于打开源文件和目标文件。
- (2) 使用 `fopen` 函数打开源文件，文件路径为 `word[line].mystr[1]`，以只读模式打开。
使用 `fopen` 函数打开目标文件，文件路径为 `word[line].mystr[2]`，以写入模式打开。
- (3) 定义字符变量 `buffer`，通过 `fgetc` 函数从源文件中读取一个字符。定义整数变量 `cnt` 并初始化为 0，用于计数读取的字符数量。在每次循环中，递增 `cnt` 计数，将 `buffer` 字符使用 `fputc` 函数写入目标文件。
循环结束后，关闭目标文件和源文件，分别使用 `fclose` 函数。

7. 程序运行的主要界面和实验结果截图

1. 使用 VMware Workstation Pro 打开 linux 虚拟机 CentOS-7.5-x86_64



2. 使用 `gcc` 编译器编译源代码并执行

```
[root@localhost ~]# gcc -g main.c -o main
[root@localhost ~]# gcc -S main.c
[root@localhost ~]# gcc -E main.c
```

3. 验证 MS-DOS 解释程序功能

(1) *echo* 命令验证

命令:

```
echo print me ! > a.txt
echo this is add >> a.txt
echo print relief
```

第一条指令将 *print me!* 写入 *a.txt*

第二条指令在 *a.txt* 后追加 *this is add*

第三条指令在屏幕输出 *print relief*

预计结果:

文件 *a.txt* 内容:

print me !

this is add

屏幕输出内容:

print relief

实际输出结果:

a.txt 内容:



屏幕输出内容:

```
print relief
```

(2) *set* 命令验证

```
set \a var = 2
set \p var2 = asdf
set
set va
set var3
echo %var2%
set var2 =
echo %var2%
```

第一条命令设置数值型环境变量 *var=2*

第二条命令设置字符串型环境变量 *var2=asdf*

第三条指令显示所有环境变量
第四条指令显示以 `va` 为前缀的环境变量
第五条指令显示以 `var3` 为前缀的环境变量
第六条指令显示环境变量 `var2`
第七条指令删除环境变量 `var2`
第八条指令再次显示环境变量 `var2`

预计结果:

```
var=2
var2=asdf
var=2
var2=asdf
var=2
var2=asdf
环境变量 var3 没有定义
asdf
%var2%
```

实际输出结果:

```
var=2
var2=asdf
var=2
var2=asdf
var=2
var2=asdf
环境变量 var3 没有定义
asdf
%var2%
```

set命令输出所有环境变量

set va 命令输出所有va前缀的环境变量

var3未定义, 输出异常

var2删除前, 输出环境变量var2的值

var2被删除后, %var2%被当作字符串输出

同时环境变量被保存至 `new.txt`



The screenshot shows a text editor window titled "new.txt". The editor has a menu bar with "文件" (File), "编辑" (Edit), and "查看" (View). The text content of the file is:

```
var=2
var2=asdf
```

(3) `if` 命令验证

a. `if exist` 验证, 判断文件是否存在
命令:

```
if exist 1.txt ( echo has this file ) else ( echo didnt has file )
if exist 2.txt ( echo has this file ) else ( echo didnt has file )
```

其中 1.txt 存在, 2.txt 不存在

预计输出结果:

has this file

didnt has file

实际输出结果:

```
has this file
didnt has file
```

b. *if NOT* 验证

命令:

```
if NOT exist 1.txt ( echo has this file ) else ( echo didnt has file )
if NOT exist 2.txt ( echo has this file ) else ( echo didnt has file )
.
```

NOT 将判断条件取反, 输出结果应与上文相反

预计输出结果:

didnt has file

has this file

实际输出结果:

```
didnt has file
has this file
```

c. *if string1 == string2* 验证:

命令:

```
if 1 == 2 ( echo 1 equal 2 ) else ( echo 1 not equal 2 )
if NOT 1 == 2 ( echo 1 equal 2 ) else ( echo 1 not equal 2 )
```

第一条指令若 1==2 则输出 1 equal 2, 否则输出 1 not equal 2。

第二条指令若 1==2 则输出 1 not equal 2, 否则输出 1 equal 2。

预计输出结果:

1 not equal 2

1 equal 2

实际输出结果:

```
1 not equal 2
1 equal 2
```

d. *if* 嵌套 *copy* 命令验证:

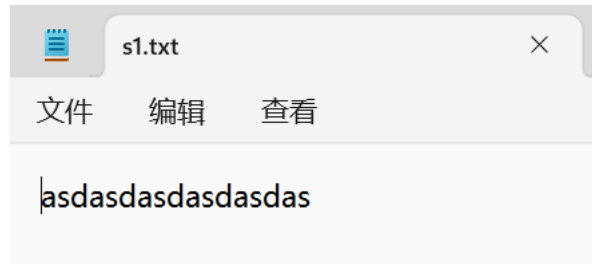
命令:

```
if NOT exist 2.txt ( copy s1.txt s2.txt )|
```

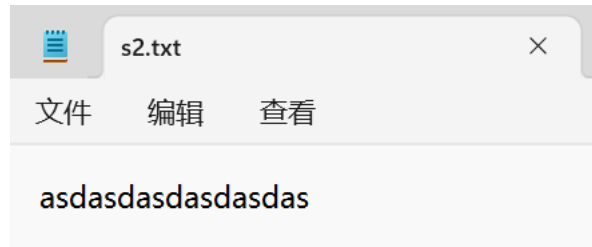
若 2.txt 不存在则将 s1.txt 复制到 s2.txt

输出结果:

s1.txt



s2.txt



成功将 s1.txt 复制到 s2.txt

(4) *for* 命令验证 (同时验证嵌套调用)

a. *for* %%variable in (set) do command 命令验证:

输入命令:

```
for %%j in (apc,bsd,caq) do set \p p1 = %%j|
```

此命令会将环境变量 *p1* 的值定义三次, 分别为 *apc*, *bsd*, *caq*

预计输出结果:

p1=apc

p1=bsd

p1=caq

实际输出结果:

```
p1=apc
p1=bsd
p1=caq
```

b. *for* /d %%variable in (set) do command 命令验证:

输入命令:

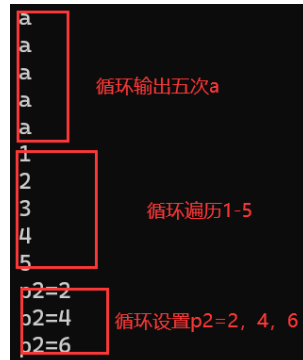
```
for /d %%i in (1,1,5) do echo a
for /d %%j in (1,1,5) do echo %%j
for /d %%i in (2,2,6) do set \p p2 = %%i
```

第一条命令循环 5 次, 输出 5 次 a

第二条命令循环 5 次, 输出 1, 2, 3, 4, 5

第三条命令循环 3 次, 3 次定义环境变量 p2 的值为 2, 4, 6

实际输出结果:



c. *for /f %%variable in (文件路径) do command* 命令验证:

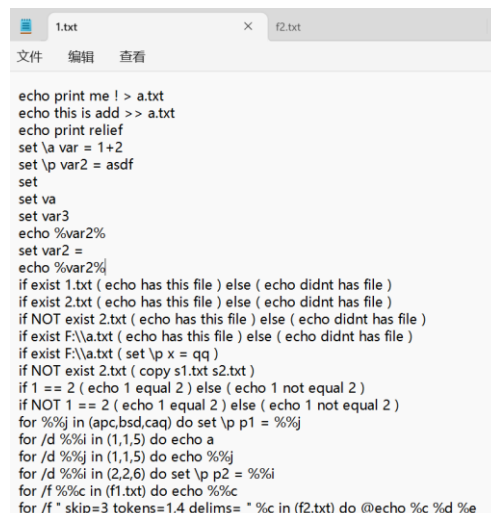
输入命令:

```
for /f %%c in (f1.txt) do echo %%c
```

此指令将循环读取文件 *f1.txt* 并输出

实际输出结果:

f1.txt 内容



输出结果

```

echo print me ! > a.txt
echo this is add >> a.txt n echo print relief
set var = 1+2
set p var2 = asdf
set
set va
set var3
echo %var2%
set var2 =
echo %var2%
if exist 1.txt ( echo has this file ) else ( echo didnt has file )
if exist 2.txt ( echo has this file ) else ( echo didnt has file )
if NOT exist 2.txt ( echo has this file ) else ( echo didnt has file )
if exist F:\a.txt ( echo has this file ) else ( echo didnt has file )
if exist F:\a.txt ( set p x = qq )
if NOT exist 2.txt ( copy sl.txt s2.txt )
if 1 == 2 ( echo 1 equal 2 ) else ( echo 1 not equal 2 )
if NOT 1 == 2 ( echo 1 equal 2 ) else ( echo 1 not equal 2 )
for %%j in (apc,bsd,caq) do set p pl = %%j
for /d %%i in (1,1,5) do echo a
for /d %%j in (1,1,5) do echo %%j
for /d %%i in (2,2,6) do set p p2 = %%i
for /f %%c in (f1.txt) do echo %%c
for /f " skip=3 tokens=1,4 delims= " %c in (f2.txt) do @echo %c %d %e

```

- d. *for /f " skip=a tokens=b, c delims=' p' " %%variable in (文件路径) do command* 命令验证:
输入命令:

```
for /f " skip=3 tokens=1,4 delims= " %c in (f2.txt) do @echo %c %d %e
```

跳过前 3 条指令，以空格为分割，输出每行第 1, 4 个字符串

实际输出结果:

```

set =
set =
set
set
set
set
echo
set
echo
if (
if (
if 2.txt
if (

```

(5) *goto*、标号、*while* 命令验证

输入命令:

```

: again
echo a
echo b
echo c
goto again

```

此命令将会循环执行输出 *a*, *b*, *c*, 实现了 *while* 功能

实际输出结果:

预计结果均与与实际输出结果相符, 说明实验功能正确。

8. 总结和感想体会

总结:

通过此次实验,我实现了在 Linux 系统上开发一个兼容 Windows 批处理文件的命令解释器。实验的目的是为 Linux 系统提供类似 WindowsCMD 的批处理能力。

在实验中,我首先创建了一个 MS-DOS 解释程序的解析器,它能够解析 .BAT 扩展名的批处理文件,并按照批处理文件中的流程命令进行逐行执行。其次,我实现了批处理文件常见的条件判断(IF)、循环(FOR、WHILE)以及变量设置(SET)和参数替换(SHIFT)等命令。这些命令的参数格式和使用方法参考了 Windows CMD,能够很好地兼容 Windows 批处理文件。

在开发过程中, 我使用 C 语言编写了命令行解析和执行模块。针对不同类型的流程命令, 设计了对应的函数来执行不同任务。同时, 定义了数据结构来存储变量和参数信息, 实现命令的参数传递。为了调试方便, 采用了 `printf` 输出方式打印执行信息。

这个实验让我熟悉了批处理程序的设计思路,掌握了命令解释器的开发方法。它也让我体会到面向不同操作系统兼容设计的重要性。总体来说,此次实验锻炼了我的软件开发能力和问题解决能力。未来在类似场景下开发可移植应用将受到帮助。

感想与体会:

- (1) 通过这个实验,我深入理解了批处理程序的工作原理,学习到了命令解释器的设计思路。了解了词法分析、语法解析、语义分析等算法,这对我理解操作系统原理很有好处。

- (2) 实现这个具有跨平台兼容性的批处理解释器, 需要考虑 Windows 和 Linux 两个系统的差异, 这给我带来了很好的设计训练。
- (3) 在开发过程中, 我熟练运用 C 语言设计模块、实现不同命令的功能以及数据结构如符号表的应用。这有效提升了我的语言应用能力。
- (4) 最后实验得以成功部署运行, 给了我很大的成就感。它也为我的学习和研究操作系统奠定了扎实的基础。

参考文献

MS-DOS 命令手册

附录 1: 程序清单(部分)

```
void action(struct A* word, int i, struct B_char* store_str, struct
B_int* store_num, int* number_str, int* number_num, struct C_int*
charater_num, int count)
{
    if (!strcmp(word[i].mystr[0], "echo")) {
        //printf("in echo");

        func_echo(word, i, store_str, store_num, number_str, number_num, charat
er_num);
    }
    if (!strcmp(word[i].mystr[0], "set"))
func_set(word, i, store_str, store_num, number_str, number_num);
    //for 的各种职能, 在 for_if 中进行实现
    if (!strcmp(word[i].mystr[0], "for")) {
        func_for(word,
i, store_str, store_num, number_str, number_num, charater_num);
    }
    if (!strcmp(word[i].mystr[0], "if")) {
        func_if(word,
i, store_str, store_num, number_str, number_num, charater_num);
    }
    if (!strcmp(word[i].mystr[0], "shift")) {

        func_shift(word, i, store_str, store_num, number_str, number_num, chara
ter_num);
    }
    if (!strcmp(word[i].mystr[0], ":")) { //标号
        func_label(word,
i, store_str, store_num, number_str, number_num, charater_num);
```

```

    }
    if (!strcmp(word[i].mystr[0], "goto")) {
        func_goto(word,
i, store_str, store_num, number_str, number_num, charater_num, count);
    }
    if (!strcmp(word[i].mystr[0], "copy")) {
        func_copy(word,
i, store_str, store_num, number_str, number_num, charater_num);
    }
}

void func_echo(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{
    if (!strcmp(word[line].mystr[word[line].numcount-2], ">")){ //
文本写入文件
        FILE* file =
fopen(word[line].mystr[word[line].numcount-1], "w");
        char write[100] = "";
        for(int i=1;i<word[line].numcount-2;i++){ //构造写入字符串
write
            strcat(write, word[line].mystr[i]);
            strcat(write, " ");
        }
        //printf(write);
        fputs(write, file);
        fputs("\n", file);
        fclose(file);
        return;
    }

    if (!strcmp(word[line].mystr[word[line].numcount-2], ">>")){ //
文件追加文本
        FILE* file =
fopen(word[line].mystr[word[line].numcount-1], "ab");
        char write[100] = "";
        for(int i=1;i<word[line].numcount-2;i++){ //构造追加字符串
write
            strcat(write, word[line].mystr[i]);
            strcat(write, " ");
        }
        //printf(write);
        fputs(write, file);

```

```

        fputs("\n", file);
        fclose(file);
        return;
    }
    for (int i=1;i<word[line].numcount;i++)
    {
        char* val; //值
        char ch2[30];
        strcpy(ch2, word[line].mystr[i]);
        if ((ch2[0] == '%') && (ch2[strlen(ch2)-1] == '%')) //
显示环境变量的值
        {
            //printf("to find");
            char *ch1;
            ch1=str_cut(ch2);

            val=find_charater(ch1, store_str, store_num, number_str, number_num);
            if(val != NULL){ //找到输出环境变量值
                printf("%s ", val);
            }
            else{ //未找到直接输出字符串
                printf("%s ", ch2);
            }
            continue;
        }
        if ((ch2[0] == '%')) //实现标号变量
        {
            int j;
            j = char_tran(ch2[1]);
            printf("%s", charater_num[j].contant);
            continue;
        }
        printf("%s ", word[line].mystr[i]); //显示字符串
    }
    printf("\n");
}

```

```

void func_set(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int* number_num)
{
    //显示当前所有环境变量
    //set
    if (word[line].numcount==1)
    {

```

```

        for (int i = 0; i < *number_num;i++)
        {

printf("%s=%s\n", store_num[i].flag, store_num[i].content);
        }
        for (int i = 0; i < *number_str;i++)
        {

printf("%s=%s\n", store_str[i].flag, store_str[i].content);
        }
        return;
    }

//输出前缀包含 c 的环境变量值
//set c
if (word[line].numcount == 2)
{
    //printf("in set c");
    int key = 0;
    char *judgestr;//set 给出的判断 c
    judgestr = word[line].mystr[1];
    //printf("strlen(c)=%d\n", strlen(judgestr)); //前缀长
    for (int i = 0; i < *number_num;i++)
    {
        //printf("strlen(key)=%d\n", strlen(store_num[i].flag));
//键长
        if (str_judge(store_num[i].flag, judgestr))//找到包含
前缀 c 的环境变量
        {
            //printf("in pipei");

printf("%s=%s\n", store_num[i].flag, store_num[i].content);
            key = 1;
        }
    }
    for (int i = 0; i < *number_str;i++)
    {
        //printf("strlen(c)=%d\n", strlen(judgestr)); //前缀长
        //printf("strlen(key)=%d\n", strlen(store_str[i].flag));
//键长
        if (str_judge(store_str[i].flag, judgestr))
        {
            //printf("in pipei");

```

```

printf("%s=%s\n", store_str[i].flag, store_str[i].content);
    key = 1;
}
}
if(key == 0){
    printf("环境变量 %s 没有定义\n", judgestr);
}
return;
}

```

//删除环境变量

```

if (word[line].numcount == 3){
    //printf("yes");
    for (int i = 0; i < *number_num; i++){
        if(!strcmp(word[line].mystr[1], store_num[i].flag)){
            for(int j = i; j < *number_num; j++){
                store_num[i] = store_num[i+1];
            }
            (*number_num)--;
        }
    }
    for (int i = 0; i < *number_str; i++){
        if(!strcmp(word[line].mystr[1], store_str[i].flag)){
            for(int j = i; j < *number_str; j++){
                store_str[i] = store_str[i+1];
            }
            (*number_str)--;
        }
    }
    /*for (int i = 0; i < number_num; i++){
        if(!strcmp(word[line].mystr[1], store_num[i].flag)){
            store_num[i]=store_num[number_num-1];
            number_num--;
        }
    }
    for (int i = 0; i < number_str; i++){
        if(!strcmp(word[line].mystr[1], store_str[i].flag)){
            store_str[i]=store_str[number_str-1];
            number_str--;
        }
    }
    */
    return;
}

```

```

//set \p var = 1234
//添加环境变量
/*if (!strcmp(word[line].mystr[1], "\p"))
{
    strcpy(store_str[number_str].flag, word[line].mystr[2]);
    strcpy(store_str[number_str].content,
word[line].mystr[4]);
    number_str++;
}*/
//set \a var = 1243
if (!strcmp(word[line].mystr[1], "\a"))
{
    strcpy(store_num[*number_num].flag, word[line].mystr[2]);
    /*
    int x;
    x=char_tran(word[line].mystr[4]);
    store_num[number_num].content = x;
    */
    strcpy(store_num[*number_num].content,
word[line].mystr[4]);
    printf("%s=%s\n", store_num[*number_num].flag, store_num[*number_num].content);
    FILE* file = fopen("new.txt", "ab");
    fputs(store_num[*number_num].flag, file);
    fputs("=", file);
    fputs(store_num[*number_num].content, file);
    fputs("\n", file);
    fclose(file);
    (*number_num)++;
}
else{
    strcpy(store_str[*number_str].flag, word[line].mystr[2]);
    strcpy(store_str[*number_str].content,
word[line].mystr[4]);
    printf("%s=%s\n", store_str[*number_str].flag, store_str[*number_str].content);
    FILE* file = fopen("new.txt", "ab");
    fputs(store_str[*number_str].flag, file);
    fputs("=", file);
    fputs(store_str[*number_str].content, file);
    fputs("\n", file);
    fclose(file);
    (*number_str)++;
}

```

```

    }
}

void func_if_exist(struct A* word, int line, int flag, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{
    int braket = 0; //记录括号数
    //int braket1[2]={0, 0};
    //int braket2[2]={0, 0}; //记录两个括号
    struct A word1[1];
    struct A word2[1]; //括号内语句
    word1[0].numcount = 0;
    word2[0].numcount = 0;
    //printf("yes");
    for(int i = 3+flag; i<word[line].numcount; i++) // 构造
word1, word2
    {
        //printf("yes");
        if (!strcmp(word[line].mystr[i], "("))
        {
            //printf("%d", i);
            braket = braket + 1;
            for(int j=i+1; strcmp(word[line].mystr[j], ")"); j++)
            {
                if(braket == 1)
                {
                    word1[0].mystr[word1[0].numcount] =
word[line].mystr[j];
                    word1[0].numcount++;
                }
                else if(braket == 2)
                {
                    word2[0].mystr[word2[0].numcount] =
word[line].mystr[j];
                    word2[0].numcount++;
                }
            }
        }
    }
}

/*printf("%d\n", braket);
printf("%d\n", word1[0].numcount);
for(int i = 0; i < word1[0].numcount; i++)
{

```

```

        printf("%s  ", word1[0].mystr[i]);
    }
    printf("\n");
    printf("%d\n", word2[0].numcount);
    for(int i = 0; i < word2[0].numcount; i++)
    {
        printf("%s  ", word2[0].mystr[i]);
    }
    printf("\n");*/
    if ((find_if_doc(word, line)) ^ flag)
    {
        //递归调用 select

        select(word1, 1, store_str, store_num, number_str, number_num, charater
_num);

        //func_echo(word1, 0, store_str, store_num, number_str, number_num, cha
rater_num);
        //printf("%s\n", word[line].mystr[5+flag]);
        //echo_if(line, 5+flag);
        return;
    }
    else
    {
        if (braket == 2)
        {

            select(word2, 1, store_str, store_num, number_str, number_num, charater
_num);

            //func_echo(word2, 0, store_str, store_num, number_str, number_num, cha
rater_num);
            //printf("%s\n", word[line].mystr[10+flag]);
            //echo_if(line, 9+flag);
        }
    }
    return;
}

int find_if_doc(struct A* word, int line)
{
    char ch2[50];
    strcpy(ch2, word[line].mystr[2]);

```



```

FILE *f = NULL;
//printf("%s", ch2);
f = fopen(ch2, "r");
if(f != NULL){
    //printf("yes fopen");
    fclose(f);
    return 1;
}
else{
    //printf("no fopen");
    fclose(f);
    return 0;
}

/*char *ret;
ret = strchr(ch2, '/');
if (ret == NULL)
{
    return find_doc5("/Users/sherry/Desktop/osFile", ch2);
}
return find_doc6(ch2);*/
}

```

```

void func_if_char(struct A* word, int line, int flag, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{

```

```

    int braket = 0; //记录括号数
    //int braket1[2]={0, 0};
    //int braket2[2]={0, 0}; //记录两个括号
    struct A word1[1];
    struct A word2[1]; //括号内语句
    word1[0].numcount = 0;
    word2[0].numcount = 0;
    //printf("yes");
    for(int i = 4+flag; i < word[line].numcount; i++) // 构造
word1, word2
    {
        //printf("yes");
        if (!strcmp(word[line].mystr[i], "("))
        {
            //printf("%d", i);
            braket = braket + 1;
            for(int j=i+1; strcmp(word[line].mystr[j], ")"); j++)

```

```

        {
            if(braket == 1)
            {
                word1[0].mystr[word1[0].numcount]
word[line].mystr[j];
                word1[0].numcount++;
            }
            else if(braket == 2)
            {
                word2[0].mystr[word2[0].numcount]
word[line].mystr[j];
                word2[0].numcount++;
            }
        }
    }

    //printf("flag=%d\n", flag);
    //printf("left=%s\n", word[line].mystr[1+flag]);
    //printf("right=%s\n", word[line].mystr[3+flag]);
    if
    ((!strcmp(word[line].mystr[1+flag], word[line].mystr[3+flag])) ^ flag)
    {

select(word1, 1, store_str, store_num, number_str, number_num, charater_num
);

        return;
    }
    else
    {
        if (braket == 2)
        {

select(word2, 1, store_str, store_num, number_str, number_num, charater_num
);

        }
    }
    return;
}

void func_if(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{

```

```

    int flag = 0;
    char str[30];
    char str_eq[30];
    strcpy(str, word[line].mystr[1]);
    strcpy(str_eq, word[line].mystr[2]);
    if (!strcmp(str, "NOT"))
    {
        flag = 1;
        //printf("flag=%d", flag);
        strcpy(str, word[line].mystr[2]);
        strcpy(str_eq, word[line].mystr[3]);
    }
    if (!strcmp(str, "exist"))
    {

func_if_exist(word, line, flag, store_str, store_num, number_str, number_num,
charater_num);
        return;
    }
    if (!strcmp(str_eq, "=="))
    {

func_if_char(word, line, flag, store_str, store_num, number_str, number_num,
charater_num);
        return;
    }
}

void func_for_s(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{
    int i;
    char strspilit[30]; //去括号后字符串
    char str[10][20];
    int stri[10];
    int index=0; //拆分括号后的字符串数组
    struct A word1[1];
    word1[0].numcount = 0; //构造的word[1]
    int flag = 0; //标记变量替换位置

    strcpy(strspilit, str_cut(word[line].mystr[3])); //除去括号
    //printf("strspilit=%s\n", strspilit);
    //printf("length=%d\n", strlen(strspilit));

```

```

for(i = 0;i < strlen(strspilit);i++)
{
    if(strspilit[i] == ',')
    {
        index++;//启动下一个新的字符串
    }
    else
    {
        str[index][stri[index]]=strspilit[i]; //生成字符串
        stri[index]++;
    }
}
//for(i = 0;i<=index;i++)
// printf("str=%s\n", str[i]);
//构造 word[0]
//printf("word[line].numcount=%d", word[line].numcount);
for(i = 5;i < word[line].numcount;i++)
{
    if (!strcmp(word[line].mystr[i], word[line].mystr[1]))
        flag=word1[0].numcount; //标记变量在 word1 中的位置
    word1[0].mystr[word1[0].numcount] = word[line].mystr[i];
    word1[0].numcount++;
}

//for(i = 0;i<word1[0].numcount;i++)
// printf("word=%s\n", word1[0].mystr[i]);
//printf("flag=%d", flag);
//实现循环
for(i = 0;i<=index;i++)
{
    if(flag!=0) //说明有变量替换
        word1[0].mystr[flag]=str[i];    //替换变量

    select(word1, 1, store_str, store_num, number_str, number_num, charater
_num); //递归调用 select
}
//for(i = 0;i<word1[0].numcount;i++)
// printf("word=%s\n", word1[0].mystr[i]);
}

void func_for_d(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{

```

```

int i;
int num[3]; //起始, 步长, 终止数组
char strspilit[30]; //去括号后字符串
struct A word1[1];
word1[0].numcount = 0; //构造的word[1]
int flag = 0; //标记变量替换位置

strcpy(strspilit, str_cut(word[line].mystr[4])); //除去括号
//printf("strspilit=%s\n", strspilit);
//printf("length=%d\n", strlen(strspilit));
num[0] = strspilit[0] - '0';
num[1] = strspilit[2] - '0';
num[2] = strspilit[4] - '0';
/*for(int i = 0; i < 3; i++)
{
    printf("num=%d", num[i]);
}*/
/*for(i = 0; i < 3; i++)
{
    num[i] = strspilit[2*i] - '0';
}
for(i = 0; i < 3; i++)
    printf("num=%s\n", num[i]);*/

//构造 word[0]
//printf("word[line].numcount=%d", word[line].numcount);
for(i = 6; i < word[line].numcount; i++)
{
    if (!strcmp(word[line].mystr[i], word[line].mystr[2]))
        flag = word1[0].numcount; //标记变量在 word1 中的位置
    word1[0].mystr[word1[0].numcount] = word[line].mystr[i];
    word1[0].numcount++;
}

//for(i = 0; i < word1[0].numcount; i++)
// printf("word=%s\n", word1[0].mystr[i]);
//printf("flag=%d", flag);
//实现循环
for(i = num[0]; i <= num[2]; i += num[1])
{
    if(flag != 0) //说明有变量替换
    {
        char change[10]; //替换用的字符串
        change[0] = i + '0';
    }
}

```

```

        word1[0].mystr[flag]=change;    //替换变量
    }

    select(word1, 1, store_str, store_num, number_str, number_num, charater
_num); //递归调用 select
    }
    //for(i = 0;i<word1[0].numcount;i++)
    // printf("word=%s\n", word1[0].mystr[i]);
}

void func_for(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{
    char str[30];
    strcpy(str, word[line].mystr[1]);
    //for %%i in (A,B,C) do
    if (str[0]=='%') //字符循环
    {

        func_for_s(word, line, store_str, store_num, number_str, number_num, ch
arater_num);
    }
    if (str[0]=='/')
    {
        if(str[1]=='d') //数字循环
        {

            func_for_d(word, line, store_str, store_num, number_str, number_num, ch
arater_num);
        }
        if (str[1] == 'f')
        {

            func_for_f(word, line, store_str, store_num, number_str, number_num, charat
er_num);

            return;
        }
    }
}

void func_for_f(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)

```

```

{
    int i = 0;
    //printf("in f");
    //第三个元素, 判断是哪一种
    char ch3[10];
    strcpy(ch3, word[line].mystr[2]);
    //取出文件的名字
    char chr[30];

    //for /f %c in (abc.txt) do echo %c 只读入每行首给%c, 只输出%c
    //缺省条件下的判断
    //自动按照空格判断
    if (ch3[0]=='%')
    {
        strcpy(chr, str_cut(word[line].mystr[4])); //获取文件名
        //printf("filename=%s", chr);

        //读文件
        char buff[255];
        FILE *filein = NULL;
        filein = fopen(chr, "r");
        if (filein==NULL)
        {
            printf("error file open");
        }
        fscanf(filein, "%s", buff);
        do
        {
            printf("%s\n", buff);
            fgets(buff, 255, (FILE*)filein);
            fscanf(filein, "%s", buff);
        }while (!feof(filein));

        fclose(filein);
        return;
    }

    //for /f "skip=1 tokens=1,4 delims= " %c in (d:\abc.txt) do
    @echo %c %d
    //delims 为分割字符
    if (ch3[0]=='"')
    {
        strcpy(chr, str_cut(word[line].mystr[9])); //获取文件名
    }
}

```

```

//printf("filename=%s", chr);

//获取 skip
int skip;
skip=tran_tri(word[line].mystr[3], 5); //获取 skip
//printf("skip=%d", skip);

//获取 tokens
int token[5];
int num = 0; //token 的下标
for (i = 7 ; i<strlen(word[line].mystr[4]); i++)
{
    //printf("%c", word[line].mystr[4][i]);
    if(word[line].mystr[4][i] == ',')
        continue;
    else
    {
        //printf("yes");
        //保存取第几个被分隔的字符串段
        token[num]=word[line].mystr[4][i] - '0';
        //printf("token=%d", token[num]);
        num++;
    }
}

//取出 delim delim 为分割方式
char delim[2];
char delimain[10];
strcpy(delimain, word[line].mystr[5]); //取出 delims 字段
if (strlen(delimain)==7) //缺省为空格分割
{
    //printf("yes");
    //delim = ' ';
    strcpy(delim, " ");
}
else{ //否则取出分割字段
    delim[0] = word[line].mystr[5][7];
    //printf("%s", delim);
}
//printf("delim=%s", delim);

//读文件
FILE *fp = NULL;
char buff[255];

```



```

fp = fopen(chr, "r"); //读入文件
if(fp == NULL) {
    printf("error to open file");
} else {
    //读入
    int pc=0; //用于跳过前 skip 条指令
    while (!feof(fp)) {
        struct A word1[1]; //保存每行分割后的字符串数组
        word1[0].numcount = 0;
        fgets(buff, 255, (FILE*)fp);
        if(pc<skip)
        {
            pc++;
            continue;
        }
        //printf("%s", buff);
        if (!feof(fp)) {
            //读一行文件并分割
            strcpy(word1[0].sentence, buff);
            //printf("word1 = %s \n", word1[0].sentence);
            del_char(word1[0].sentence, '\n');
            //printf("word1 = %s \n", word1[0].sentence);
            seperate_delim(word1, 0, delim);
            //printf("word.mystr = %s \n", word1[0].mystr[0]);
            for(i = 0; i<num; i++)
            {
                //输出 tokens 指定的位置的字段
                printf("%s      ", word1[0].mystr[token[i]-1]);

            }
            printf("\n");
        } //sentence 空
    }
}
}

```

```

void func_label(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{
    int chnum = get_ch_num(charater_num);
    //printf("chnum=%d", chnum);
    strcpy(charater_num[chnum].contant, word[line].mystr[1]); //
存入标号

```

```

        //printf("label=%s", charater_num[chnum].contant);
        charater_num[chnum].flag = line;    //存入指令位置(在 word 中的
位置)
        //printf("pc=%d", charater_num[chnum].flag);
        printf("chnum=%d", get_ch_num(charater_num));
    }

```

```

void func_goto(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num, int count)
{
    int i = 0;
    char ch[30];
    strcpy(ch, word[line].mystr[1]); //获取要 goto 的标号
    int pc = 0; //要跳转到的位置
    //printf("goto %s", ch);
    int chnum = get_ch_num(charater_num);
    //printf("chnum=%d", chnum);

    //找到要跳转的位置
    for(i = 0; i < chnum; i++)
    {
        if(!strcmp(ch, charater_num[i].contant))
            pc = charater_num[i].flag;
    }
    //printf("pc=%d", pc); \

    //从 pc+1 处开始重新顺序执行指令
    for(i = pc+1; i < count; i++)
    {

        action(word, i, store_str, store_num, number_str, number_num, charater_
num, count);    //重新顺序执行
    }
}

```

```

void func_shift(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{
    char temp = word[line].mystr[1][0];
    int sh_time = temp - '0'; //从第 sh_time 个参数开始移位
    for (int i = sh_time; i < 9; i++)
    {

```

```

        strcpy(charater_num[i].contant,
charater_num[i+1].contant);
    }
}

```

```

void func_copy(struct A* word, int line, struct B_char*
store_str, struct B_int* store_num, int* number_str, int*
number_num, struct C_int* charater_num)
{
    //printf("src=%s", word[line].mystr[1]);
    //printf("dest=%s", word[line].mystr[2]);
    FILE *fp1 = fopen(word[line].mystr[2], "w");
    FILE *fp2 = fopen(word[line].mystr[1], "r");
    if(fp1 == NULL) {
        printf("error dest");
        return;
    }
    if(fp2 == NULL) {
        perror("error src");
        return;
    }
    char buffer = fgetc(fp2);
    int cnt = 0;
    while(!feof(fp2)) {
        cnt++;
        fputc(buffer, fp1);
        buffer = fgetc(fp2);
    }
    fclose(fp1);
    fclose(fp2);
}

```