Project FLA 实验报告

姓名	学号	邮箱	日期
薛飞阳	181860121	xue.fyang@foxmail.com	2020/12/24

§1. 实验完成度及结果

§1.1. 实验完成度

我完成了实验的全部内容，包括：

- 任务一：实现多带图灵机解析器的普通模式和`verbose`模式
- 任务二：实现多带图灵机模拟器的普通模式和`verbose`模式
- 任务三：实现识别手册中给出的两个语言的图灵机对应的图灵机程序，依次命名为case1.tm、case2.tm

§1.2. 实验结果

得到实验结果的文档结构如下：

（并不是最终提交的文档结构）

```
.
├── README.md
├── programs
│   ├── case1.tm
│   ├── case2.tm
│   ├── palindrome_detector_2tapes.tm #手册提供的文件，仅在测试中使用
│   ├── wcase1.tm
│   ├── wcase2.tm
│   └── wcase3.tm #wcase为展示图灵机程序错误的文件，仅在测试中使用
└── turing-project
    ├── main.cpp
    ├── makefile
    ├── parser.cpp
    ├── simulator.cpp
    ├── terminal.cpp
    ├── turing #编译后得到的可执行程序，仅在测试中使用
    └── turing.h
```

实验结果如下：

§1.2.1. 正确的输入格式

以`palindrome_detector_2tapes.tm`为例，该图灵机程序检测输入字符串是否为二进制回文串

正确输入格式的结果如下：（由于`verbose`模式下输出过多，故截图省略了中间部分过程）

```
mxtremist@GM501GS:~/FLA/turing-project$ ./turing palindrome_detector_2tapes.tm 101
true
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -v palindrome_detector_2tapes.tm 101
Input: 101
================== RUN ==================
Step   : 0
Index0 : 0 1 2
Tape0  : 1 0 1
Head0  : ^
Index1 : 0
Tape1  : _
Head1  : ^
State  : 0
-----------------------------------------
Step   : 1
Index0 : 0 1 2
Tape0  : 1 0 1
Head0  : ^
Index1 : 0
Tape1  : _
Head1  : ^
State  : cp
-----------------------------------------
Step   : 2
Index0 : 0 1 2
Tape0  : 1 0 1
Head0  :   ^
Index1 : 0 1
Tape1  : 1 _
Head1  :   ^
State  : cp
-----------------------------------------
Step   : 3
```

```
-----------------------------------------
Step   : 17
Index0 : 3 4 5 6
Tape0  : t r u e
Head0  :       ^
Index1 : 1
Tape1  : _
Head1  : ^
State  : halt_accept
-----------------------------------------
Result : true
================== END ==================
```

如果输入不接受的字符串例如“10”，则会最终输出false，这里不再展示。

-h模式则将输出提示。

```
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -h
usage: turing [-v|--verbose] [-h|--help] <tm> <input>
```

§1.2.2. 错误的输入格式

这里考虑了多种错误输入，详细可见`§2. 分析与设计思路`

需要注意的是，某些错误提示可能在普通模式和`verbose`模式下输出不同，这里仅展示`verbose`模式。

```
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -v palindrome_detector_2tapes.tm
command error
mxtremist@GM501GS:~/FLA/turing-project$ ./turing v palindrome_detector_2tapes.tm 101
command error
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -v palindrome_detector.tm 101
open file "palindrome_detector.tm" error
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -v wcase1.tm ab
Something wrong in file "wcase1.tm", line 10, "#Q = 0,accept,accept2,accept3,accept4,halt_accept,reject,reject2,reject3,reject4,reject5,halt_reject}", formal error
syntax error
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -v wcase2.tm ab
Something wrong in file "wcase2.tm", line 33, "0 a_ a_ ** cp_a", "cp_a" is not exist
syntax error
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -v wcase3.tm ab
Something wrong in file "wcase3.tm", line 48, "mh aa aaa *1 mh", "aaa" is not matched with tape num
syntax error
mxtremist@GM501GS:~/FLA/turing-project$ ./turing -v palindrome_detector_2tapes.tm 1a1
Input: 1a1
================== ERR ==================
error: 'a' was not declared in the set of input symbols
Input: 1a1
         ^
================== END ==================
mxtremist@GM501GS:~/FLA/turing-project$
```

依次展示的错误为：

- 两种典型的命令行错误
- 文件名错误
- 典型的图灵机程序中出现语法错误(wcase1)
- 图灵机程序出现不存在的状态或符号(wcase2)
- 图灵机程序中出现与纸带数不匹配的状况(wcase3)
- 命令行输入含有不存在的符号

§2. 分析与设计

任务三的分析与设计部分已经按照要求写在了每个文件内，故本部分只介绍任务一和任务二的分析与设计。

§2.1. 初步分析与设计

对图灵机程序的解析过程，可以分为如下几步：

1. 得到命令
2. 检测命令，提取出运行模式、文件名、输入
3. 解析器，打开图灵机程序文件，解析出相关内容
4. 模拟器，根据之前解析的内容，运行图灵机程序，得到结果
5. 输出结果

每一步都需要注意对错误的处理

我们将第一步放在main函数里完成，第二步用类`Terminal`处理，第三步用类`parser`处理，第四步和第五步放在`simulator`类中处理。

执行过程应该为：在main函数中会调用terminal，terminal处理完命令后调用parser，parser解析后调用simulator，simulator输出结果后再返回正确结束的信号，再依次回到main。

因此，我们构造的三个类中都会有一个`int run`函数，每个类依次调用下一个类的这一函数并获得其返回值，剩余的应该均是该类自己完成的工作。但是在实现过程中发现，解析器在解析过程中得到的内容是模拟器需要用到的，且内容非常多，如果都作为参数传入写起来比较臃肿复杂，故解析器在调用模拟器之前，会为模拟器内部的变量进行赋值。具体各个部分怎么实现将在接下来进行阐述。

每个类的定义均放在`turing.h`头文件中。

§2.2. 具体实现

§2.2.1. makefile文件的编写

根据§2.1. 的分析，我们在本次实验中可以构建如下的代码结构：

```
├── main.cpp #入口
├── parser.cpp #解析器
├── simulator.cpp #模拟器
├── terminal.cpp #命令行处理
└── turing.h #头文件
```

其中.cpp文件的编译均依赖且仅依赖于于`turing.h`，故makefile文件的编写可以用以下两个target完成：

```
turing:$(OBJS)
    $(CC) -o turing $(OBJS)
    rm -rf *.o

%.o:%.c terminal.h
    $(CC) $(CFLAGS) -c $< -o $@
```

其中，

```
CC = g++
CFLAGS = -c
OBJS = main.o simulator.o parser.o terminal.o
```

这样经过make命令就得到了可执行文件`turing`。

§2.2.2. main

main函数需要做的仅是获取命令后交给terminal处理，而main函数的原型

```
int main(int argc, char *argv[])
```

中，参数argc指明函数数量，参数argv为字符串数组，故将这两个参数交给terminal即完成了自己的任务。main函数的返回值取决于最终的结果，成功运行则返回0，报错则返回1。故其内容仅有三行代码：

```
int main(int argc, char *argv[])
{
    Terminal t(argc, argv);
    int ans = t.run();
    return ans == 0 ? 0 : -1;
}
```

§2.2.3. terminal

Terminal类的定义如下，其写在`turing.h`中：

```
class Terminal
{
private:
    int myargc;
    char **myargv;
    void help();
    bool verbose;
    Parser p;
    string input;
    string filename;

public:
    Terminal();
    Terminal(int argc, char **x);
    int run();
    int terminalcheck();
};
```

`run`函数是Terminal类的入口，在外部调用`Terminal(int argc, char **x)`初始化完成后，其已经相当于得到了分词后的命令和其个数，`run`先调用`terminalcheck()`函数检查命令格式是否正确，并根据返回值做不同操作。同时，`terminalcheck()`函数在正确的情况下解析出`input`和`filename`变量。其返回值有三种：

1. 返回-1，意思是命令要求输出`help`，此时需要调用`help()`函数；

2. 返回0，意思是没有发现错误并成功解析出变量；
3. 返回err code，意思是出现错误，需要输出相应的错误提示；

err code是在`turing.h`中define的一系列错误代码，这里用到的有：

```
#define err1 1 //err1:命令行格式错误
```

run函数在处理完返回值非0的情况后会return 0或err code，整个程序的运行就会结束。在返回0的情况下，将会调用Parser，根据文件名做进一步的解析。由于parser的返回值依然会是非0，且一些错误是由Terminal输出的故其`run`函数的流程如下：

```
int ans = terminalcheck();
if (ans == -1) //help
{
    help();
    return 0;
}
if (ans == 0)
    ans = p.run(input, filename, verbose); //调用Parser
if (ans != 0)
{
    /* error Process */
}
return ans;
```

terminalcheck的过程是较为简单的词法分析，根据argv内容和其位置判断，并且在正确的情况下为input, filename, verbose赋值，其中verbose是指示是否为verbose模式的布尔变量。其详细过程这里不再赘述了。

§2.2.4. parser

§2.2.4.1. 执行流程

Parserl类的定义如下，其写在`turing.h`中：

```
class Parser
{
private:
    string input;
    string filename;
    Simulator s;
    bool verbose;
    int hashcheck(string line, string firstword); //检查#行错误
    int Qcheck(string line);
    int Scheck(string line);
    int Gcheck(string line);
    int qOcheck(string line);
    int Bcheck(string line);
    int Fcheck(string line);
    int Ncheck(string line);
    int finalcheck(string &line, string &wrongword, int &lseq); //检查是否有err4或
err5
    vector<parserdelta> deltas;
    verboseseqs hashseqs[3];
```

```
    void initSimDelta();

public:
    Parser();
    int parsercheck();
    int run(string i, string f, bool v);
    void verboseOutput3(string line, int lseq);
    void verboseOutput45(string line, string wrongword, int lseq, int err);
};
```

虽然`Parser`类有很多函数，但是从这些函数的函数名就可以看出来基本都是用于进行检查。

`run`函数是Parser类的入口。与`Terminal`类似，其run函数的处理流程为：

```
int ans = parsercheck();
if (ans != 0)
    return ans;
initSimDelta(); //初始化simulator的转移函数
ans = s.run(input); //调用simulator
return ans;
```

`parsercheck()`检查了tm文件是否正确，其可以分为三类：

1. 检查文件名是否正确，即是否可以正确打开文件
2. “#”开头的语句，交由`hashcheck()`做进一步判断
3. 剩下的就是图灵机的转移函数，检查无误后将其暂存以便后续使用

暂存：由于有”#“行出现在转移函数之后的可能，在全部记录完成之后还需要再次检查以便防止转移函数中出现不存在的符号，故需要先将转移函数暂存，这里使用了如下的数据结构：

```
struct parserdelta
{
    string statenow;  //当前状态
    string tapenow;   //当前位置
    string tapewrite; //写入符号
    string tapedir;   //移动方向
    string statenext; //下一状态
    int lseq;         //文件行号
};
```

这里的检查中涉及到以下错误：

```
#define err2 2 //err2:打开文件失败
#define err3 3 //err3:#行的图灵机格式错误
#define err4 4 //err4:出现不存在的状态或字符
#define err5 5 //err5:符号与纸带数不匹配
```

其中err4和err5都是在整个文件扫描完成之后才能检测出来的。

为了避免复杂传参，对于”#“行，在检查无误之后，直接修改了simulator对象内的值。

下面介绍一下parsercheck的过程。

§2.2.4.2. parsercheck

首先检查文件名，默认的文件存储在"../programs/"文件夹内，故利用`fstream`库看是否可以打开文件即可，错误返回err2，这个错误将被terminal捕获并输出相关提示。

打开文件后按行读取，然后检测开头是否为"#"，是则交由`hashcheck()`，这个函数会查看具体属于哪一个定义，以#Q为例，它会检查开头是否符合"#Q = "这样的格式，如果符合就删去这一部分，将本行剩余内容交给`Qcheck()`处理：

```
if (firstword == "#Q")
{
    ans = Qcheck(line.substr(5, line.length() - 1));
}
```

具体的各项内容的检查可以分为：

- 字符串集合类，包括Q、F
- 符号集合类，包括S、G
- 单个字符串类，包括q0、N（数字也视为字符串）
- 单个字符类，即B

后两类的处理较为简单，定义了两个子函数`charcheck`和`stringcheck`用来检查是否符合规范，其中charcheck检查单个字符是否规范，分为两类情况：属于字母表即符合、字母表去除“_”，后者用于之后检查输入符号集（输入不包含空字符）。在检查完之后会直接对simulator值进行修改。以检测B为例，其过程则如下：

```
int Parser::Bcheck(string line)
{
    if (line.length() > 1 || !(charcheckall(line[0])))
        return err3;
    s.blank = line[0];
    return 0;
}
```

前两者的检测较为复杂，同样使用了两个子函数`splitwordsString`和`splitwordsChar`，其功能为：按照传入参数`string delim`进行分词（本次使用的`delim`均为“,”）。分词后则得到了<单个字符串类的数组>和<单个字符类的数组>，然后就可以按照后两类进行处理了。

`hashcheck`完成之后，对于三类特殊的hash（q0, B, F），还需要记下其完整信息，这是因为这三类和转移函数一样，除了词法语法错误之外，还可能出现集合中不存在的内容，这都要放在文件处理完成之后再做。而在verbose模式下输出会记录其所在行的完整内容，所以需要做一个记录。

除了`hashcheck`，文件中还会出现转移函数，由于转移函数的检查依赖于#行的定义，故先存在之前提到的parserdelta中，于是只需要使用`sstream`类检查数量是否为5，然后按顺序存入即可：

```
parserdelta d = {words[0], words[1], words[2], words[3], words[4], lseq};
deltas.push_back(d);
```

这样，就检测完了文件，排除了err2和err3，接下来调用`finalcheck()`进行下一步的检查。

`finalcheck()`首先检查前面提到的特殊的q0, B, F，查看其内内容是否存在在集合中，如果不在则属于err4，然后检查转移函数。转移函数中可能会出现如下几种错误：

- delta中statenow、statenext不存在
- delta中方向存在无意义字符

- 方向数量、纸带符号数量、修改后纸带数量与N不匹配
- delta中tapenow、tapenext存在无意义字符

各个变量的含义在§2.2.4.1. 中有。

这几种错误的检测都较为简单，比较麻烦的是在`verbose`模式下对出错位置进行标记，故`finalcheck()`使用如下原型：

```
int Parser::finalcheck(string &line, string &wrongword, int &lseq)
```

line为行内容，lseq为行号，wrongword为具体出错的单词或字符，在检测出对应错误后就会修改这三个变量的值以便之后输出。

§2.2.4.3. initSimDelta

`finalcheck()`完成之后需要进行simulator中转移函数的初始化，我们分析一下simulator对于转移函数的需求，在运行过程中，是根据当前状态、当前纸带内容两个变量来决定接下来该怎么走，故我们使用这样两个结构体存储：

```
struct From
{
    string statenow;
    string tapenow;
    //重载操作符，以便使用map
    bool operator<(const From &other) const
    {
        if (statenow != other.statenow)
        {
            return statenow < other.statenow;
        }
        else
        {
            return tapenow < other.tapenow;
        }
    }
};
struct To
{
    string tapewrite;
    string tapedir;
    string statenext;
};
```

From里重载了操作符，这是为了之后simulator存储得以使用map结构。这样simulator在查找时只需要根据现有信息组合出一个From即可快速查询到To。

转移函数的初始化即为遍历原先存下的已经检查过的parserdelta并组合出From,To再加入simulator。

§2.2.4.4. verbose下的输出

在之前的过程中的任意一步出现错误，如果是`verbose`模式下，则会调用相关的函数，我们可以分为以下两类：

- 格式错误，err3
- 出现未知符号，err4和err5

对于第一种错误，`verbose`模式下会使用之前存储的错误行号，输出格式错误；对于第二种错误，`verbose`模式下会使用之前存储的错误行内容、具体错误词/符号·错误行号进行输出。

§2.2.5. simulator

§2.2.5.1. 运行过程

Simulator类的定义如下，其写在`turing.h`中：

```
class Simulator
{
private:
    //void printall();
    string getTapeNow();
    bool inputcheck(int &seq);
    void verboseState(int step, string state);
    void verboseInput(bool s, int seq);
    void TuringChange(To t, string &StateNow, string &TapeNow);
    void FinalOutput();

public:
    set<string> states;
    set<char> inputsymbols;
    set<char> tapesymbols;
    string start;
    set<string> finals;
    int tapenum;
    char blank;
    vector<string> tapes;
    vector<int> heads;
    vector<vector<int>> indexs;
    map<From, To> deltas;
    int run(string i);
    bool verbose;
    string input;
};
```

simulator存储的states、inputsymbols、tapesymbols、start、finals、tapenum、blank、deltas分别对应文件中的Q、S、G、q0、B、F、N、所有的delta，已经在解析器中进行了赋值。

simulator的运行过程为：

1. 根据input初始化纸带
2. 根据存储内容循环运行图灵机
3. 输出最终结果

在之前内容检查无误后，simulator只会出现一种错误：input中出现了不存在的字符：

```
#define err6 6 //err6:输入不合法
```

所以在检测完这个错误后就可以开始初始化纸带，该检查放在`inputcheck()`进行，过程较为简单，依次遍历并查询是否在`set<char> inputsymbols`即可。

初始化纸带的过程为：

- 如果input为空，将其赋值为blank，因为无法处理空字符串
- 纸带数组加入N个纸带，第一个纸带的内容为input，其余为blank
- 初始化每个纸带对应的heads和indexs

tapes为纸带数组，heads为纸带指针位置，index为对应标号，index是用于`verbose`模式下的输出。

初始化完成后就可以开始运行了，运行放在一个while循环里，有两个退出循环的条件：到达终止态或无相关的转移函数，其代码如下：

```
while (1)
{
    if (verbose)
    {
        verboseState(seq, StateNow);    //输出verbose模式下相关信息
    }
    if (finals.find(StateNow) != finals.end()) //到达终止态，运行结束
    {
        break;
    }
    From f = {StateNow, TapeNow};
    map<From, To>::iterator it = deltas.find(f); //读取delta函数
    if (it == deltas.end())                      //无相关转移函数，运行结束
    {
        break;
    }
    To t = it->second;
    TuringChange(t, StateNow, TapeNow);
    seq++;
}
```

TuringChange通过读取的To结构变量t，输入StateNow和TapeNow，根据这三者修改StateNow和TapeNow，以及成员变量tapes、indexs、heads的值，从而进入下一步。seq变量代表运行步数。

我们回顾以一下To结构体内的变量：

```
struct To
{
    string tapewrite;
    string tapedir;
    string statenext;
};
```

tapes的修改只要将每个纸带的head指向位置修改为tapewrite对应的纸带值即可；indexs和heads的修改取决于tapedir标识的下一步的移动方向，需要额外注意一下空纸带的处理；StateNow在最后修改为statenext；TapeNow的修改是读取移动后的各个纸带head指向字符拼接而成。

§2.2.5.2. 运行输出

除了运行图灵机，还需要考虑输出。普通模式下较为简单，结束时输出第一个纸带内容即可，下面着重介绍一下`verbose`模式下的输出。

`verbose`模式下的输出可以分为三部分：

- 初始输出，放在`verboseInput()`里
- 每一步的输出，放在`verboseState()`里
- 最后的输出，放在`FinalOutput()`里

`verboseInput()`在err（输入不合法）下的输出和正确情况有所区别，正确情况下只输出Input和"RUN"，而错误情况下会输出input、”ERR“以及详细的错误信息，并且最后输出“END”（此时不会进入`FinalOutPut()`）；

`verboseState()`的输出根据成员变量的值不难确定，较为困难的是对齐，这里输出开头使用库`iomanip`里的`std::left`指定左对齐，使用`setw`指定宽度，由于index和tapenum都是数字，其可能不止一位，故需要用变量记录纸带数量的位数和index的位数。而在具体输出内容时先用字符串将内容和应该加的空格数全部记录下来，然后再在最后输出，这是为了避免多次循环。另外在输出的时候注意负数不需要输出负号。

输出纸带i的index如下：

```
cout << std::left << std::setw(w + 7) << index;
cout << ": " << indexContent << endl;
```

其中w是纸带数量这个数字的位数，index是字符串"Index "+当前纸带序号的字符串形式，indexContent是需要输出的index内容的字符串形式，输出tape和head与之类似。

`FinalOutput()`在`verbose`模式下的输出也需要进行左对齐，这个过程可以参考`verboseState()`的输出。输出时只要输出第一个非空字符开始到最后一个非空字符即可。字符串输出完毕后再加上"END"。这样就完成了最终结果的输出。

至此，已经介绍完了任务一和任务二的全部过程。

§3. 总结·感想·建议

这次实验难度适中，在没有任何框架代码的情况下自己设计框架、慢慢填充并看着自己的代码最终成功运行是非常有成就感的事情。

通过这次实验，我对图灵机的运行细节和图灵机程序的组成有了更加深入、更加系统化的了解，此外，此前我一直没有系统化学习过makefile文件编写，通过这次实验也间接性地学会了makefile文件的编写。

建议：

1. 希望对手册中”编写一个用户友好的命令行工具“有更准确的阐述，笔者以及笔者所知道的一些同学一开始都误以为，这句话的意思是要自己实现一个命令行程序，在进入运行时while循环输出命令行提示符$，blahblah这样的方式。有这个想法可能是因为在以前的学习中有实验是这样的做法。此外，实例中给出的`turing`命令看起来像是需要将可执行文件加入到环境变量里，后来笔者的同学问了助教才知道并不是强制性的要求。
2. 或许可以在图灵机程序语法部分加上一些限制，比如限制转移函数delta只会出现在其他六者的最后（或者说明不加以限制）。笔者此次实验是假设没有该限制并做了对应的鲁棒性处理。