FLA实验报告

分析与实现

PDA解析器

PDA的解析器实际上类似于编译器中的语法分析器。但是由于PDA的语法相对简单，所以我们直接根据模式进行匹配
首先进行对文件的预处理。每次读取文件中的一行时删除 **;** 后的所有内容。这意味着忽视注释。同时删去所有后面多余的空格。

```C++
if (line[line.find_first_of(";")] == ';') {
        line = line.substr(0, line.find_first_of(";"));
}
while (line[line.size() - 1] == ' ') {
    line = line.substr(0, line.size() - 1);
}
```

如果行首元素是 **#** ，那么我们默认这个是对相关集合的声明操作。随后定义相关宏处理相关的语言集合。
宏定义如下：

```C++
#define HANDLE_SET_FUNCTION(name)                                          \
  bool handle##name(string line, PDA &pda, int &pos) {                     \
    if (!checkEqual(line.substr(pos + 1, 3))) {                            \
      pda.handleSyntaxError("' = ' expected");                             \
    }                                                                      \
    string g =                                                             \
        line.substr(line.find_first_of("{"),                               \
                    line.find_last_of("}") - line.find_first_of("{") + 1); \
    if (!checkBrace(g)) {                                                  \
      pda.handleSyntaxError("{...} expected");                             \
    }                                                                      \
    string curr;                                                           \
    int lastComma = 0;                                                     \
    for (int i = 1; i < g.size() - 1; i++) {                               \
      if (g[i] == ',') {                                                   \
        curr = g.substr(lastComma + 1, i - lastComma - 1);                 \
        pda.add##name(curr);                                               \
        lastComma = i;                                                     \
      }                                                                    \
    }                                                                      \
    curr = g.substr(lastComma + 1, g.size() - 1 - lastComma - 1);          \
    pda.add##name(curr);                                                   \
    pos += 3 + g.size();                                                   \
    return true;                                                           \
  }

HANDLE_SET_FUNCTION(StateSet)
HANDLE_SET_FUNCTION(StackSet)
HANDLE_SET_FUNCTION(InputSet)
HANDLE_SET_FUNCTION(FinalStateSet)

#undef HANDLE_SET_FUNCTION
```

该宏严格判定等于号必须遵循 **'='** 的设定。并且 **','** 之间不能有间隔。 随后直接读取 **'{}'** 里的内容。如果里面包含非法字符，则在add##name里面进行修正更改报错。

如果行首元素不是#，那么我们默认其进入转移函数的匹配
转移函数默认具有5个参数，如果参数之间多余一个空格也可以处理，通过切割空格后获取的5个参数进行状态上的检查，确保其在状态集出现过，对于输入以及栈符号也进行类似检查。最后通过一个映射存储该转移函数，该映射的定义如下：

```C++
unordered_map<tuple<string, string, string>, pair<string, string>,
              TupleHasher>
      delta; // transition function Q x sigma x gamma -> Q x gamma*
```

PDA模拟器

PDA模拟器的核心则是以转移函数为核心进行构建。首先将当前状态赋值为初始状态，当前栈符号赋值为初始栈符号。对于每一个input的字符进行转移，并以此更改当前状态。值得注意的是，我们专门在input后加入*方便进入最终转移函数，但是如果input中含有*则会报错。

```C++
this->input += "_";
  for (auto s : this->input) {
    string input = std::string(1, s);
    string top = currentStack.substr(0, 1);
    pair<string, string> nextState =
        delta[make_tuple(currentState, input, top)];
    currentStack = nextState.second == "_"
                       ? currentStack.substr(1)
                       : nextState.second + currentStack.substr(1);
    currentState = nextState.first;
    if (nextState == make_pair("", "")) {
      return 0;
    }
    if (finalState.find(currentState) != finalState.end()) {
      return 1;
    }
  }
```

TM解析器

TM的解析过程跟PDA如出一辙，细节是转移函数当中对于新旧的栈符号进行违规处理时，添加了 * 和 _ 的处理规则，特判了这两个字符。并且得确保栈符号组一定是等于纸带数的。

TM模拟器

TM的模拟过程其实核心也跟处理PDA的过程一致，即读取当前状态，通过转移函数获取新的状态，更改当前状态，如此反复。
实现过程如下：

```C++
while (true) {
    string currentInput = "";
    for (int i = 0; i < this->n; i++) {
      currentInput += "*";
    }
    tuple<string, string, string> next;
    if (verbose == 1)
      printStepLog(currentState, time);
    int ptx = 0;
    if (finalState.find(currentState) != finalState.end() ||
        !dfs(currentInput, ptx)) {
      string result = tape[0];
      result = result.substr(result.find_first_not_of("_"));
      result = result.substr(0, result.find_last_not_of("_") + 1);
      for (auto s : result) {
        if (s == '_')
          cout << ' ';
        else
          cout << s;
      }
      cout << endl;
      if (verbose)
        cout << "===============END===============" << endl;
      break;
    }
    next = delta[make_pair(currentState, currentInput)];
    string output = get<0>(next);
    string direction = get<1>(next);
    string nextState = get<2>(next);

    // change tape
    for (int i = 0; i < this->n; i++) {
      if (output[i] == '*')
        continue;
      tape[i][pos[i]] = output[i];
    }

    // move head of the tape
    for (int i = 0; i < this->n; i++) {
      if (direction[i] == 'l') {
        pos[i]--;
        if (pos[i] < 0) {
          offset++;
          pos[i] = 0;
          for (int j = 0; j < this->n; j++) {
            if (j != i)
              pos[j]++;
            tape[j] = "_" + tape[j];
          }
        }
      } else if (direction[i] == 'r') {
        pos[i]++;
        if (pos[i] >= tape[i].size()) {
          tape[i] += "_";
        }
      } else if (direction[i] == '*') {
        continue;
      }
    }
    currentState = nextState;
    time++;
  }
}
```

不过在实现的过程中，有一个问题需要处理

读取当前状态

读取当前状态作为转移函数的参数时由于有 * 的存在可以匹配任何字符，所以得确定传入什么才能确保转移函数的正确运行。解决方案就是DFS，去寻找所有可以匹配的字符

实现如下：

```C++
std::function<bool(string &, int)> dfs = [&](string &currentInput,
                                             int ptx) {
      if (ptx >= this->n) {
        if (delta[make_pair(currentState, currentInput)] ==
            make_tuple("", "", "")) {
          return false;
        }
        return true;
      }
      currentInput[ptx] = tape[ptx][pos[ptx]];
      if (dfs(currentInput, ptx + 1)) {
        return true;
      }
      currentInput[ptx] = '*';
      if (dfs(currentInput, ptx + 1)) {
        return true;
      }
      return false;
    };
```

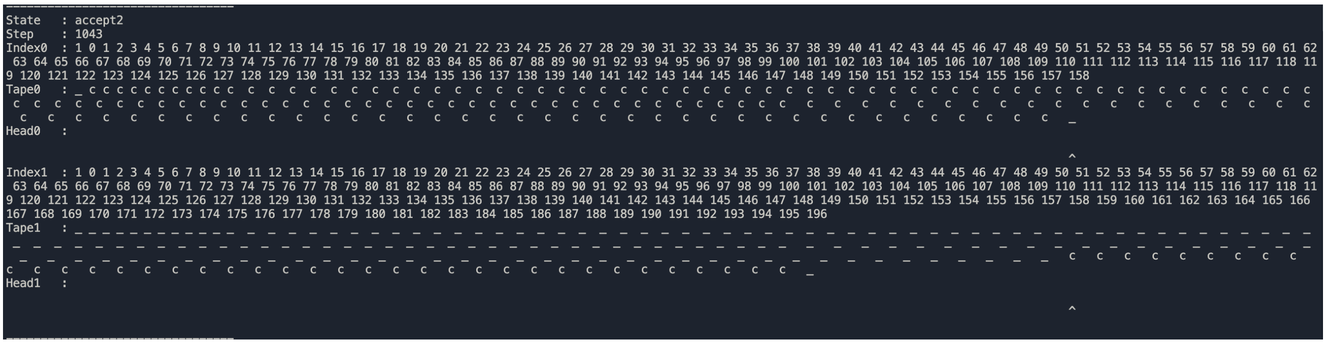
如果当前找不到可以转移的函数或者当前状态进入到接收状态，停机并输出第一条纸带的内容
对于向左写纸带的情况，如果当前磁带头的位置就是纸带首，那么其余全体磁带头往后偏移一位，随后为所有的纸带添加 **字符**
对于向右写字符的情况，如果当前磁带头的位置为磁带尾，则在当前磁带末尾追加 字符。
值得一提的是为了满足verbose输出小于0的情况，添加了offset偏移量，这个offset记录了全体磁带向左偏移了多少。

实验完成度

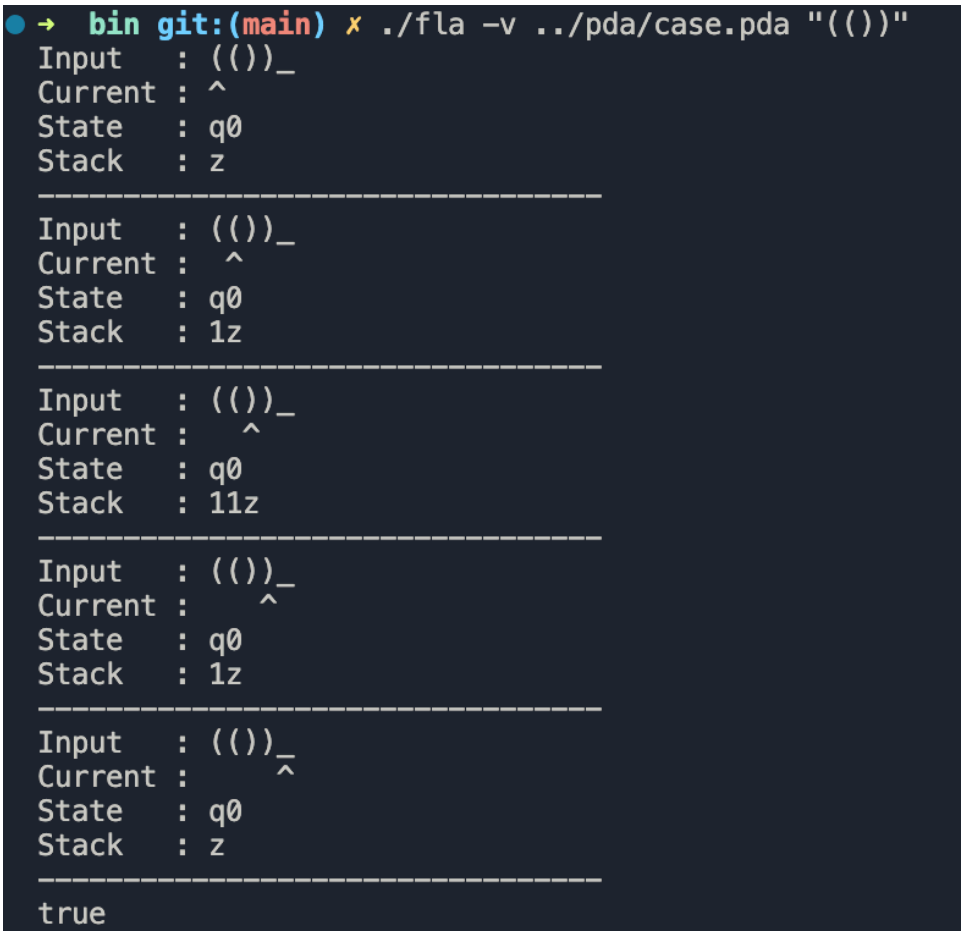
- 完成了TM和PDA的verbose模式，可以输出详细记录信息用来调试图灵程序
  TM的verbos模式

```
---------------END---------------
● ➜  bin git:(main) ✗ ./fla -v ../tm/case1.tm ab
  Input: ab
  ===============RUN===============
  State    : 0
  Step     : 0
  Index0   : 0 1
  Tape0    : a b
  Head0    : ^
  Index1   : 0
  Tape1    : _
  Head1    : ^
  ---------------------------------
  State    : 0
  Step     : 1
  Index0   : 0 1
  Tape0    : a b
  Head0    :   ^
  Index1   : 0
  Tape1    : _
  Head1    : ^
  ---------------------------------
  State    : 1
  Step     : 2
  Index0   : 0 1 2
  Tape0    : a b _
  Head0    :     ^
  Index1   : 0
  Tape1    : _
  Head1    : ^
  ---------------------------------
  State    : multi
  Step     : 3
  Index0   : 0 1 2
  Tape0    : a b _
  Head0    :   ^
  Index1   : 0
  Tape1    : _
  Head1    : ^
  ---------------------------------
```

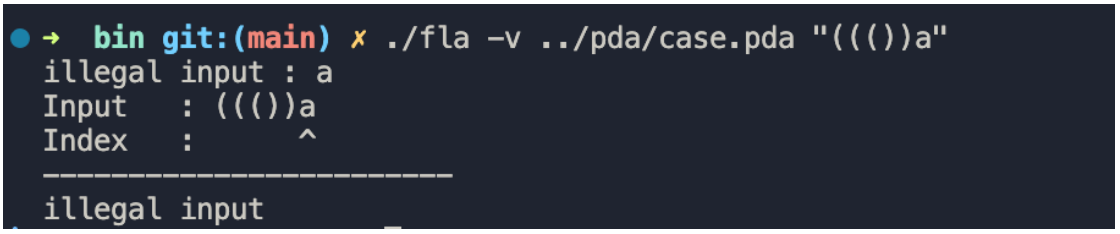
并且当输入过大时也可以对齐



可以看到，满足当index大于10甚至100的时候输出日志依然保持对齐
PDA的verbose模式：
该模式下会打印出当前的input，状态以及栈元素。



如果有不符合的输入串



- 基本要求全部达成

实验出现的问题

1. 参数的解析
   由于本次实验的参数只有两个，不考虑其扩展性我们直接进行特判
2. c++特性的运用
   初次使用unordered_map<tuple, pair>时，得需要实现tuple和pair的默认哈希函数才可以。
3. 处理相关任务比较繁琐
   处理语法上的，语义上的问题太多。只能慢慢磨

实验改进的地方

后续可以更加模块化，不能main.cpp写到底。初期以为项目的规模不大（实际上确实不大）就直接在主文件中写入全部的实现。
随着项目的持续进行，错误的处理范式应该更早的确定，而不是拆东墙补西墙

一些建议

实验手册的部分可以优化为PDA一个章节，TM一个章节。这样在做实验的时候不会因为不确定这个需求到底是PDA的还是TM的捣鼓半天。
对于最后的输出要求其实可以整合一下，比如对语法的错误输出要求，对于程序的输出要求放在一起，这样可以减少同学们对实验输出要求描述的模糊感。