数据结构综合训练*

沈俞霖, 陈铮

2017 年 2 月 27 日

组名 白膜法师
成员 软件 51 沈俞霖
2151601013
成员 软件 42 陈铮
2141601026
提交日期 2017 年 2 月 27 日
联系电话 13679119978

*完整代码和附加文件请参见 https://github.com/sylxjtu/dsProject

目录

1 迷宫

1.1 实验名称

迷宫

1.2 需求和规格说明

1.2.1 问题描述

题目要求使用 Union-Find 数据结构完成一个迷宫的生成。迷宫的入口点位于左上角，出口点是在图的右下角，在迷宫的矩形中，左上角的单元被连通到右下角的单元，而且这些单元与相邻的单元通过墙壁分离开来。

1.2.2 输入输出

输入 输入两个整数 $width, height > 0$，代表所生成迷宫对应的宽高。

输出

- 用合理的方式展现生成的迷宫。
- 给出迷宫可以走通的路径。以 SEN…（代表向南，然后向东，然后再向北，等等）的形式给出输出结果。

1.3 设计

1.3.1 设计思想

要生成的迷宫可以看做一个 $width * height$ 的节点矩阵，矩阵的相邻节点之间有边（无墙）或无边（被墙阻挡），这样的一个节点矩阵构成了一个图，规定左上角节点为入口，右下角节点为出口。若要使得每个节点与入口连通且边数最少，这样一个节点图为一棵树，因此问题可以规约为寻找一棵随机生成树。

主要算法 首先处理出所有可能的边，将此边对应的位置设为墙，进行随机排列后遍历，其中当前边连接的节点若

- 未连通，则连通此边，将此边对应的墙转为空地

- 已连通，则跳过此边

节点之间的连通性使用 Union-Find 数据结构进行维护遍历完成后将迷宫地图输出，再在迷宫地图上进行广度优先搜索寻找到起点到终点的路径。

存储结构 在节点数据结构中存储节点在 Union-Find 数据结构中的父节点的指针在边数据结构中存储边所连接的两个节点

1.3.2 设计表示

迷宫节点 `class Node`

`Node* parent`
代表 Union-Find 数据结构中本节点的父节点指针

`Node* findparent ()`
查询节点在 Union-Find 树上的根节点

`Node ()`
默认初始化函数初始根节点为节点本身

`void merge (Node& that)`
合并两个节点所在集合

`bool operator== (Node& that)`
查询两个节点是否在同一集合

连接节点的边 `class Edge`

`Node *s, *e`
边的起点，终点

`int x, y`
边所代表的障碍所在位置

`Edge (Node* s, Node *e, int x, int y)`
初始化边

`bool destroy ()`
判断一条边上两个节点是否连通，若不连通则合并节点并删除边

1.3.3 实现注释

各项功能已全部实现，新实现了在展示迷宫的同时展示通路的功能

1.3.4 详细设计表示

Maze.cpp

```cpp
#include <bits/stdc++.h>
using namespace std;

// Union-Find 节点
class Node {

  // 父节点
  Node* parent;

  // 查询根节点
  Node* findparent () {
    if(this == parent) {
      return this;
    } else {
      return this->parent = this->parent->findparent();
    }
  }

public:
  // 初始化根节点为自己
  Node () {
    parent = this;
```

```
23    }
24
25    // 合并两个节点所在的集
26    void merge (Node& that) {
27      findparent()->parent = that.findparent();
28    }
29
30    // 检验两个节点是否在同一个集合中
31    bool operator== (Node& that) {
32      return findparent() == that.findparent();
33    }
34    bool operator!= (Node& that) {
35      return !(*this == that);
36    }
37
38  };
39
40  // 连接迷宫节点的边
41  class Edge {
42
43    // 边的起点，终点
44    Node *s, *e;
45
46  public:
47    // 边所代表的障碍所在位置
48    int x, y;
49
50    // 初始化
51    Edge (Node* s, Node *e, int x, int y):
52    s(s), e(e), x(x), y(y){}
53
54    // 判断一条边上两个节点是否连通，若不连通则合并节点并删除边
55    bool destroy () {
```

```
56      if(*s != *e) {
57        s->merge(*e);
58        return true;
59      }
60      return false;
61    }
62
63  };
64
65  // 泛化二维数组
66  template < class T >
67  class Array2d {
68
69    T* data;
70
71  public:
72
73    // 二维数组的宽高（x, y）
74    int width, height;
75
76    // 二维数组构造析构
77    Array2d (int width, int height) : width(width),
      ↪  height(height) {
78      data = new T[width * height];
79    }
80    ~Array2d () {
81      delete[] data;
82    }
83
84    // 操作二维数组（类似于 array[y][x]）
85    T& operator () (int x, int y) {
86      return data[y * width + x];
87    }
```

```
88
89    // 验证下标是否越界
90    bool valid(int x, int y) {
91      return 0 <= x && x < width && 0 <= y && y < height;
92    }
93
94    // 填充二维数组
95    void clear(const T& value) {
96      for(int i = 0; i < width * height; i++) {
97        data[i] = value;
98      }
99    }
100
101 };
102
103 // 迷宫地图节点类型
104 enum dispType {
105   WALL,     // 墙
106   PASSABLE, // 空地
107   WAY,      // 最短路上的节点
108 };
109
110 // 读取输入
111 void readInput (int& width, int& height) {
112   cerr << "Enter width and height" << endl;
113   cerr << "Width: ";
114   cin >> width;
115   cerr << "Height: ";
116   cin >> height;
117 }
118
119 int main () {
120   int width, height;
```

```
121   // 输入并检查合法性 (width > 0 && height > 0)
122   readInput(width, height);
123   try {
124     if(!(width > 0 && height > 0)) throw("Width and height must
      ↪  be greater than 0");
125   } catch(const char* s) {
126     printf("Error: %s\n", s);
127     return 0;
128   }
129
130   // 分配空间
131   Array2d<Node> nodes(width, height);  // 迷宫中的节点
132   Array2d<int> disp((width * 2) + 1, (height * 2) + 1);  // 最
      ↪  终显示出的地图
133   vector<Edge> edges;  // 连接节点的边
134
135   // 初始化地图
136   disp.clear(dispType::WALL);
137   for(int i = 0; i < height; i++) {
138     for(int j = 0; j < width; j++) {
139       disp(j * 2 + 1, i * 2 + 1) = dispType::PASSABLE;
140     }
141   }
142   disp(0, 1) = disp(disp.width - 1, disp.height - 2) =
      ↪  dispType::WAY;
143
144   for(int i = 0; i < height; i++) {
145     for(int j = 0; j < width - 1; j++) {
146       edges.emplace_back(&nodes(j, i), &nodes(j + 1, i), j * 2
        ↪  + 2, i * 2 + 1);
147     }
148   }
149   for(int i = 0; i < height - 1; i++) {
```

```
150       for(int j = 0; j < width; j++) {
151         edges.emplace_back(&nodes(j, i), &nodes(j, i + 1), j * 2
              ↪  + 1, i * 2 + 2);
152       }
153     }
154
155     // 随机拆除边使图最终成为树
156     random_shuffle(edges.begin(), edges.end());
157     for(auto& e: edges) {
158       if(e.destroy()) {
159         disp(e.x, e.y) = dispType::PASSABLE;
160       }
161     }
162
163     // 通过 BFS 查找最短路径
164     Array2d< pair<int, int> > parent(disp.width, disp.height);
165     Array2d<bool> visited(disp.width, disp.height);
166     visited.clear(0);
167     queue<pair<int, int>> bfsq;
168     bfsq.emplace(disp.width - 1, disp.height - 2);
169     while(!bfsq.empty()) {
170       pair<int, int> p = bfsq.front();
171       bfsq.pop();
172       if(p == make_pair(0, 1)) break;
173       visited(p.first, p.second) = 1;
174       for(int i = -1; i <= 1; i += 2) {
175         if( parent.valid(p.first + i, p.second) &&
              ↪  !visited(p.first + i, p.second) && disp(p.first + i,
              ↪  p.second) ) {
176           parent(p.first + i, p.second) = make_pair(p.first,
                ↪  p.second);
177           bfsq.emplace(p.first + i, p.second);
178         }
```

```
179       }
180       for(int i = -1; i <= 1; i += 2) {
181         if( parent.valid(p.first, p.second + i) &&
          ↪  !visited(p.first, p.second + i) && disp(p.first,
          ↪  p.second + i) ) {
182           parent(p.first, p.second + i) = make_pair(p.first,
           ↪  p.second);
183           bfsq.emplace(p.first, p.second + i);
184         }
185       }
186     }
187
188     // 标记最短路径
189     int curx = 0, cury = 1;
190     string route;
191     while(make_pair(curx, cury) != make_pair(disp.width - 1,
       ↪  disp.height - 2)){
192       disp(curx, cury) = dispType::WAY;
193       int nxtx, nxty;
194       tie(nxtx, nxty) = parent(curx, cury);
195       route.push_back("NW ES"[(nxtx - curx) + (nxty - cury) * 2 +
        ↪  2]);
196       tie(curx, cury) = tie(nxtx, nxty);
197     }
198
199     // 打印迷宫
200     for(int i = 0; i < disp.height; i++) {
201       for(int j = 0; j < disp.width; j++) {
202         cout << (disp(j, i) ? "  " : "[redacted]");
203       }
204       cout << endl;
205     }
206     cout << "Press <Enter> to print route" << endl;
```

```
207
208   // 等待用户确认后打印最短路
209   getchar();
210   getchar();
211   for(int i = 0; i < disp.height; i++) {
212     for(int j = 0; j < disp.width; j++) {
213       cout << (
214         disp(j, i) == dispType::PASSABLE ?
215           "  " :
216           (disp(j, i) == dispType::WAY) ?
217             "\e[0;31m██\e[0;0m" :       // 红色
218             "██"
219       );
220     }
221     cout << endl;
222   }
223   cout << route << endl;
224
225   return 0;
226 }
```

1.4 调试报告

1.4.1 遇到的主要问题和解决方案

在编写程序的过程中遇到的主要问题是不能很好地实现生成随机排列算法，使用了 `std::random_shuffle` 解决

1.4.2 对于设计编码的回顾讨论和分析

- 设计编码中通过使用操作符重载的特性简化了代码量，增加了可读性
- 设计编码中没有很好地使用模块化特性，代码文件偏长，不能很好定位

1.4.3 时空分析

设迷宫大小 $width * height$ 为 N，程序分几个步骤执行，每个步骤的时间复杂度为

初始化地图和边列表 $O(N)$

随机拆除边使图最终成为树 随机排边 $O(N)$，拆除 N 条边 $O(N)$

通过 BFS 查找和标记最短路径 $O(N)$

输出迷宫 $O(N)$

整个程序的空间复杂度为 $O(N)$

1.4.4 改进设想

- 使用 GUI 界面代替 CLI 界面，使界面更加美观
- 添加交互性模块，使用户能够操作角色在迷宫内活动

1.5 运行结果展示

图 1: 迷宫 1

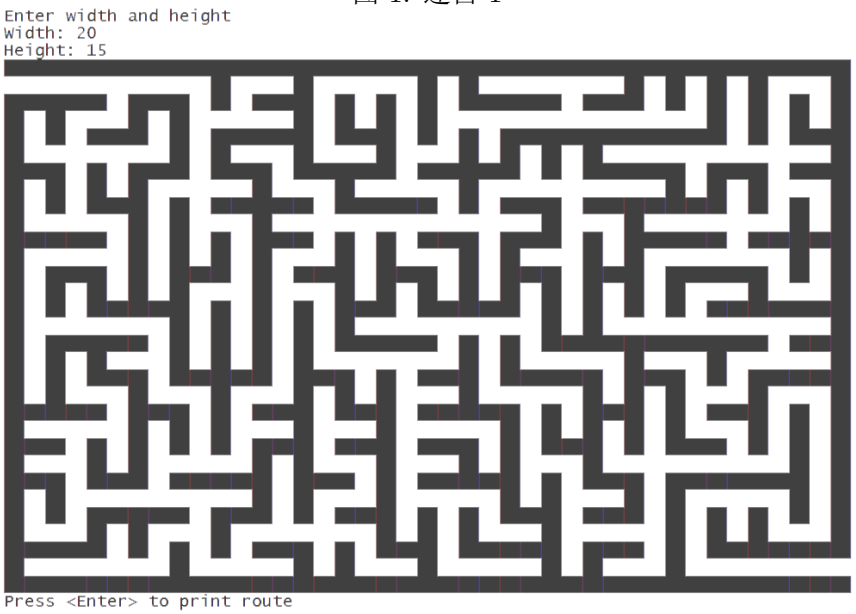
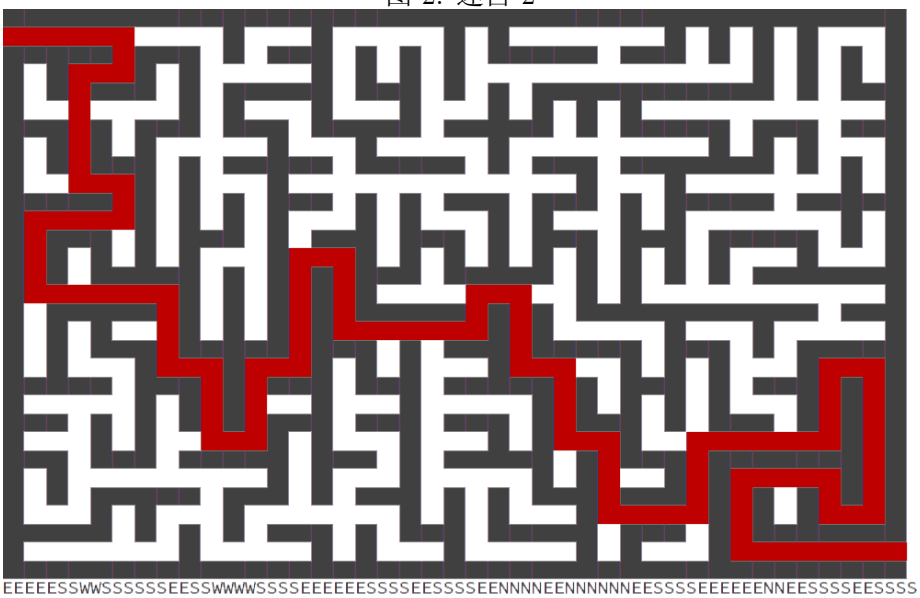


图 2: 迷宫 2



EEEEESSWWSSSSSSEESSWWWWSSSSEEEEEESSSSEESSSSEENNNNEENNNNNEESSSSEEEEEENNEESSSSEESSSS
EESSSSEEEENNNNEEEEEENNNNEESSSSSSSSSWWNNWWWWSSSSEEEEEEE

2 微型编程语言解释器

2.1 实验名称

微型编程语言解释器

2.2 需求和规格说明

2.2.1 问题描述

解释器（英语：Interpreter），又译为直译器，是一种电脑程序，能够把高级编程语言一行一行直接转译运行。解释器不会一次把整个程序转译出来，只像一位“中间人”，每次运行程序时都要先转成另一种语言再作运行，因此解释器的程序运行速度比较缓慢。它每转译一行程序叙述就立刻运行，然后再转译下一行，再运行，如此不停地进行下去。该题目要求实现的是一个能够解释执行具有赋值语句、函数定义语句以及函数执行语句的小型解释器。下面将通过解释器的输入、输出以及限制等方面描述该问题。

2.2.2 输入输出

输入 输入是由一组赋值语句、函数定义语句以及函数调用语句构成。

赋值语句具有如下的定义形式：ASSIGN <variable> <expression>。variable 是用一个仅包含英文字母所构成的长度不超过 8 个字符的字符串构成；expression 是一个算术四则混合运算表达式，其包含的运算符就是 +、-、*、 / 四个算术运算法，其操作数要么是正数（只需要考虑整数形式）要么是已定义的 variable。表达式中的运算符和操作数使用空格的形式进行分割。一个赋值语句只占用一行，具体举例如下：

```
ASSIGN X 1
ASSIGN X X + 2
ASSIGN Number (X + 2) * (X - 2)
```

函数调用语句的定义形式：CALL <function> 。function即为函数名字，函数名字和variable名字的定义要求是一样的。函数调用语句也只占用一行，具体举例如下：

```
CALL TryThis
```

函数定义语句是由一组语句组成，具体要求如下：

1. 第一行必须是 DEFINE <function>
2. 最后一行必须是 END
3. 除第一行和最后一行之外的所有行只能是赋值语句或者函数调用语句

具体举例如下：

```
DEFINE IncrementX
ASSIGN X X+1
END
DEFINE FF
CALL IncrementX
ASSIGN Y X*X
END
```

备注：输入的源代码中可以包含空行，但这些空行在解释执行过程中都应该被忽略。

输出 解释器执行后，其输出内容应该就是对每一条语句执行结果的显示。具体解释如下：

1. 每当执行一条 ASSIGN 语句，解释器都应该输出如下格式的内容：
   `Assigning <value> to <variable>`
2. 每当执行到函数定义语句，解释器都应该输出如下格式的内容：
   `Defining function <function name>`
3. 每当执行到函数调用语句，解释器都应该输出如下格式的内容：
   `Calling function <function name>`

假设一个源代码如下所示：

```
ASSIGN X 1
ASSIGN Y 1
DEFINE Fib
ASSIGN TMP Y
ASSIGN Y X+Y
ASSIGN X TMP
END
CALL Fib
ASSIGN W X
CALL Fib
ASSIGN Z W * Y - X * X
```

在上面的输入下，解释器的执行结果应该如下：

```
Assigning 1 to X
Assigning 1 to Y
Defining Fib
Calling Fib
Assigning 1 to TMP
Assigning 2 to Y
Assigning 1 to X
Assigning 1 to W
Calling Fib
```

```
Assigning 2 to TMP
Assigning 3 to Y
Assigning 2 to X
Assigning -1 to Z
```

2.2.3 其他要求

解释器可以忽略如下检查

- 输入的源代码格式总是正确的
- 每个变量在访问之前总是有值的
- 每个函数在输入的源代码中只会被定义一次
- 每个函数在调用之前总是被定义好的
- 不支持递归函数的调用，不管是直接的递归调用还是间接的递归调用

2.2.4 扩展要求

使得该编程语言支持循环语句，具体的循环语句格式为：

```
FOR <expression> <statement>
```

循环语句将根据expression计算的结果值执行多次statement语句。其中,statement要么是一条赋值语句,要么是一个函数调用语句;expression只有在进入循环时才会被计算，而且仅被计算这一次。举例如下：

```
FOR 10 ASSIGN N N + N
FOR / N 2 ASSIGN X X + 1
FOR 8 CALL Fib
```

当执行循环语句时，输出的要求只具体到输出循环语句中对赋值语句或者函数调用语句的输出，而不用专门写对循环语句的输出内容。

2.3 设计

2.3.1 设计思想

该解释器可以分为两个主要模块，一个模块负责解释执行代码，另一模块负责表达式计算。

解释器以行为基本单位执行代码，每行有一个简单语句或复合语句，其中

- FOR 语句为复合语句
- 其他语句为简单语句

对每个简单语句定义一个函数进行处理，这个函数负责执行语句，输出调试信息和更新当前行号。

定义一个全局符号表，使用 std::unordered_map 作为 hash 表存储符号值，其中

- 对变量型符号，符号表存储变量的值
- 对函数型符号，符号表存储函数定义行的行号

表达式计算模块负责处理和计算表达式，输入为一个表达式字符串和一个符号表，输出为表达式的值，内部使用抽象语法树 AST 作为表达式的中间存储格式，以增强本程序的扩展性。

主要算法

解释器 首先以行为单位读入程序，初始化当前执行行为第一行，然后以行为单位进行迭代执行，直到当前执行行超出最后一行边界停止执行。对于每个语句

ASSIGN 从字符串中获取变量名和表达式，计算变量值，将变量值写入符号表中，跳转到下一行

DEFINE 从字符串中获取函数名，向下寻找到END语句，跳转到函数定义结束

CALL 从字符串中获取函数名，从函数起始行开始执行，直到END语句

FOR 从字符串中获取循环次数表达式和语句，计算出循环次数，以此次数执行循环体语句

表达式计算器 初始化运算符优先级，通过 LL1 算法解析出抽象语法树，通过在抽象语法树上递归计算得到表达式值

存储结构 在符号表中存储符号和值的对应关系程序代码以 `std::string` 的线性表方式按行存储抽象语法树以节点为存储单位，每个节点存储节点类型、值和其 child 节点

2.3.2 设计表示

抽象语法树节点 `class ASTNode`

`enum Type`
节点类型

`INVALID` 无效节点

`HUB` 非叶节点

`INTEGER` 整数字面值节点

`OPERATOR` 运算符节点

`SYMBOL` 变量节点

`uint64_t value`
节点值 (针对符号节点和整数字面值节点)

`std::string symbol`
节点名称 (针对变量节点)

`std::vector<ASTNode> children`
子节点 (针对非叶节点)

`int resolve(std::unordered_map<std::string, Symbol>& scope)`
求值 (传入作用域)

符号 (代表变量和函数) `class Symbol`

`std::string name`
符号名

enum Type

Integer 整数变量

Function 函数

int value
符号值 (变量的值或函数的起始行号)

Symbol(std::string name, Type type, int value)
初始化函数

表达式解析器 namespace Parser

constexpr int maxLevel = 1
最高运算符优先级

const std::array<std::unordered_set<char>, maxLevel + 1> operators
运算符表

void nextToken (const std::string& expression, int& cursor)
将指针移向下一个单元 (运算符，变量名，数字) 的第一个字符

ASTNode expr (const std::string& expression, int& cursor)
解析一个表达式

ASTNode expn (const std::string& expression, int& cursor, int n)
解析一个只含有优先级大于等于 n 的运算符的表达式

ASTNode rxpn (const std::string& expression, int& cursor, int n)
解析一个以运算符开头的只含有优先级大于等于 n 的运算符的表达式

ASTNode atom (const std::string& expression, int& cursor)
解析一个数字、变量、负号表达式

2.3.3 实现注释

各项功能已全部实现，新实现了 FOR 循环嵌套的功能和打印表达式树的功能

2.3.4 详细设计表示

AST 节点定义 ASTNode.h

```
#ifndef AST_NODE_H
#define AST_NODE_H

#include "Symbol.h"
#include <unordered_map>
#include <string>
#include <vector>

// 抽象语法树节点
class ASTNode {

public:
  // 节点类型
  enum Type {
    INVALID,  // 无效节点
    HUB,      // 非叶节点
    INTEGER,  // 整数字面值节点
    OPERATOR, // 符号节点
    SYMBOL,   // 变量节点
  };
  Type type;

  // 节点值（针对符号节点和整数字面值节点）
```

```
24    uint64_t value;
25
26    // 节点名称（针对变量节点）
27    std::string symbol;
28
29    // 子节点（针对非叶节点）
30    std::vector<ASTNode> children;
31
32    // 打印 AST(仅用作 debug)
33    void display();
34
35    // 求值（传入作用域）
36    int resolve(std::unordered_map<std::string, Symbol>& scope);
37  };
38
39  #endif
```

AST 节点实现 ASTNode.cpp

```
1   #include "include/ASTNode.h"
2   #include <cstdio>
3
4   // 打印 AST(仅用作 debug)
5   void ASTNode::display() {
6     // 结果为 LISP 表达式
7     if(type == HUB) {
8       printf("(");
9       for(unsigned i = 0; i < children.size(); i++) {
10        children[i].display();
11        printf(i == children.size() - 1 ? "" : " ");
12      }
13      printf(")");
14    } else if(type == INTEGER) {
15      printf("%d", value);
```

```
16    } else if(type == OPERATOR) {
17      printf("%c", value);
18    } else if(type == SYMBOL) {
19      printf("%s", symbol.c_str());
20    } else {
21      printf("!INVALID NODE!");
22    }
23  }
24
25  // 求值（传入作用域）
26  int ASTNode::resolve(std::unordered_map<std::string, Symbol>&
    ↪  scope) {
27    if(type == HUB) {
28      // 对非叶节点递归求值
29      // 一元表达式
30      if(children.size() == 2) {
31        if(children[0].value == '-') {
32          return -children[1].resolve(scope);
33        }
34      }
35      // 二元表达式
36      if(children.size() == 3) {
37        int a = children[1].resolve(scope);
38        int b = children[2].resolve(scope);
39        switch (children[0].value) {
40          case '+':
41            return a + b;
42            break;
43          case '-':
44            return a - b;
45            break;
46          case '*':
47            return a * b;
```

```
            break;
          case '/':
            return a / b;
            break;
          case '%':
            return a % b;
            break;
        }
      }
    } else if(type == INTEGER) {
      // 整数节点直接求值
      return value;
    } else if(type == OPERATOR) {
      // 符号节点不可求值
      throw "Invalid resolve";
    } else if(type == SYMBOL) {
      // 变量节点到作用域查询值
      return scope[symbol].value;
    }
    throw "Invalid resolve";
}
```

表达式解析器定义 **Parser.h**

```
#ifndef PARSER_H
#define PARSER_H

#include "ASTNode.h"
#include <vector>
#include <string>
#include <unordered_set>

// 表达式解析器
namespace Parser {
```

```
11    // 最高运算符优先级
12    constexpr int maxLevel = 1;
13
14    // 运算符表
15    const std::array<std::unordered_set<char>, maxLevel + 1>
      ↪  operators{
16      std::unordered_set<char>{'+', '-'},       // 优先级 0
17      std::unordered_set<char>{'*', '/', '%'}   // 优先级 1
18    };
19
20    // 将指针移向下一个单元（运算符，变量名，数字）的第一个字符
21    void nextToken (const std::string& expression, int& cursor);
22
23    // 解析器函数
24    ASTNode expr (const std::string& expression, int& cursor);
      ↪  // 解析一个表达式
25    ASTNode expn (const std::string& expression, int& cursor, int
      ↪  n);   // 解析一个只含有优先级大于等于 n 的运算符的表达式
26    ASTNode rxpn (const std::string& expression, int& cursor, int
      ↪  n);   // 解析一个以运算符开头的只含有优先级大于等于 n 的运
      ↪  算符的表达式
27    ASTNode atom (const std::string& expression, int& cursor);
      ↪  // 解析一个数字、变量、负号表达式
28  }
29
30  #endif
```

表达式解析器实现 **Parser.cpp**

```
1   #include "include/Parser.h"
2   #include <cctype>
3   #include <array>
4
5   // 表达式解析器
```

```
6  namespace Parser {
7    // 将指针移向下一个单元（运算符，变量名，数字）的第一个字符
8    void nextToken (const std::string& expression, int& cursor) {
9      // 跳过当前单元
10     if(isalnum(expression[cursor])) {
11       while(isalnum(expression[cursor])) {
12         cursor++;
13       }
14     }
15     else {
16       cursor++;
17     }
18
19     // 跳过空白字符
20     while(expression[cursor] == ' ') {
21       cursor++;
22     }
23   }
24
25   // 解析一个表达式
26   ASTNode expr (const std::string& expression, int& cursor) {
27     return expn(expression, cursor, 0);
28   }
29
30   // 解析一个只含有优先级大于等于 n 的运算符的表达式
31   ASTNode expn (const std::string& expression, int& cursor, int
     ↪  n) {
32     // n 大于最大优先级时不可能有运算符满足，转为求解原子表达式
33     if(n > maxLevel) {
34       return atom(expression, cursor);
35     }
36
37     // 先解析当前位置的高阶表达式
```

```
38      ASTNode node = expn(expression, cursor, n + 1);
39
40      // 当前符号为优先级 n
41      while(operators[n].count(expression[cursor])) {
42        // 解析剩余表达式
43        ASTNode tmpnode = rxpn(expression, cursor, n);
44        // 构造 AST 节点
45        node.children = std::vector<ASTNode>{tmpnode.children[0],
          ↪  node, tmpnode.children[1]};
46        node.type = ASTNode::Type::HUB;
47      }
48      return node;
49    }
50
51    // 解析一个以符号开头的 n 级剩余表达式
52    ASTNode rxpn (const std::string& expression, int& cursor, int
      ↪  n) {
53      ASTNode node, op, value;
54
55      // 读取运算符
56      op.type = ASTNode::Type::OPERATOR;
57      op.value = expression[cursor];
58      nextToken(expression, cursor);
59
60      // 读取剩余部分
61      value = expn(expression, cursor, n + 1);
62      node.children.push_back(op);
63      node.children.push_back(value);
64
65      // 返回临时 AST 节点
66      node.type = ASTNode::Type::HUB;
67      return node;
68    }
```

```
69
70    ASTNode atom (const std::string& expression, int& cursor) {
71      ASTNode node;
72
73      if(expression[cursor] == '(') {
74        // 略过括号，解析括号内表达式
75        nextToken(expression, cursor);
76        node = expr(expression, cursor);
77        nextToken(expression, cursor);
78      } else if(isdigit(expression[cursor])) {
79        // 解析整数字面值
80        int ret;
81        sscanf(expression.c_str() + cursor, "%d", &ret);
82        node.type = ASTNode::Type::INTEGER;
83        node.value = ret;
84        nextToken(expression, cursor);
85      } else if(isalpha(expression[cursor])) {
86        // 解析变量名
87        node.type = ASTNode::Type::SYMBOL;
88        int tmpcursor = cursor;
89        while(isalpha(expression[tmpcursor])) {
90          node.symbol.push_back(expression[tmpcursor]);
91          tmpcursor++;
92        }
93        nextToken(expression, cursor);
94      } else if(expression[cursor] == '-') {
95        // 解析负号表达式
96        node = rxpn(expression, cursor, 99);
97      }
98      return node;
99    }
100 }
```

符号定义 Symbol.h

```
#ifndef SYMBOL_H
#define SYMBOL_H

#include <string>

// 符号（代表变量和函数）
class Symbol {
public:
  // 符号名
  std::string name;

  // 符号类型
  enum Type {
    Integer, Function
  };
  Type type;

  // 符号值（变量的值或函数的起始行号）
  int value;

  Symbol() = default;
  Symbol(std::string name, Type type, int value):
    name(name),
    type(type),
    value(value){}
};

#endif
```

解释器实现 Interpreter.cpp

```
#include <bits/stdc++.h>
#include "include/Parser.h"
```

```cpp
#include "include/Symbol.h"
using namespace std;

// 全局变量
unordered_map<string, Symbol> symbols;   // 符号表
vector<string> program;                  // 程序代码
char buffer[2][1024];                    // 临时缓存

// 辅助函数
// 检验某字符串是否以模式串开头
bool beginWith(const string& source, const string& pattern) {
  return source.substr(0, pattern.size()) == pattern;
}

// 解析表达式
int parseExpression (string expression) {
  int cursor = 0;
  // 防止 nextToken 越过表达式末尾
  expression.push_back('$');
  return Parser::expr(expression, cursor).resolve(symbols);
}

// 读取代码
void readProgram () {
  string temp;
  while(getline(cin, temp)) {
    // 当代码行不以字母开头时结束
    if(temp != "" && isalpha(temp[0])) {
      program.push_back(temp);
    }
  }
}

```

```
36 // 语句执行函数
37 bool executeStatement (const string& statement, unsigned&
   ↪ lineno);
38 bool executeLine (const string& line, unsigned& lineno);
39 void executeAssign (const string& statement, unsigned& lineno);
40 void executeDefine (const string& statement, unsigned& lineno);
41 void executeCall (const string& statement, unsigned& lineno);
42
43 // 赋值
44 void executeAssign (const string& statement, unsigned& lineno)
   ↪ {
45   sscanf(statement.c_str(), "%*s%s %[^\n]", buffer[0],
     ↪ buffer[1]);
46   string symbol(buffer[0]), expression(buffer[1]);
47   int value = parseExpression(expression);
48   symbols[symbol] = Symbol(symbol, Symbol::Integer, value);
49   cout << "Assigning " << value << " to " << symbol << endl;
50   lineno++;
51 }
52
53 // 定义函数
54 void executeDefine (const string& statement, unsigned& lineno)
   ↪ {
55   sscanf(statement.c_str(), "%*s%s", buffer[0]);
56   string symbol(buffer[0]);
57   symbols.emplace(make_pair(symbol, Symbol(symbol,
     ↪ Symbol::Function, lineno)));
58   cout << "Defining " << symbol << endl;
59   // 跳转到函数定义结束
60   while(!beginWith(program[lineno], "END")) {
61     if(lineno >= program.size()) {
62       cout << "ERROR: \"END\" not found in function definition"
         ↪ << endl;
```

```
63          exit(0);
64        }
65        lineno++;
66      }
67      lineno++;
68    }
69
70    // 执行函数
71    void executeCall (const string& statement, unsigned& lineno) {
72      sscanf(statement.c_str(), "%*s%s", buffer[0]);
73      string symbol(buffer[0]);
74      cout << "Calling " << symbol << endl;
75      unsigned sl = symbols[symbol].value + 1;
76      while(executeLine(program[sl], sl))
77      ;
78      lineno++;
79    }
80
81    // 执行语句
82    bool executeStatement (const string& statement, unsigned&
      ↪  lineno) {
83      if(beginWith(statement, "ASSIGN")) {
84        executeAssign(statement, lineno);
85      } else if(beginWith(statement, "DEFINE")) {
86        executeDefine(statement, lineno);
87      } else if(beginWith(statement, "END")) {
88        // 函数块结束
89        return false;
90      } else if(beginWith(statement, "CALL")) {
91        executeCall(statement, lineno);
92      }
93      // 函数块未结束
94      return true;
```

```
95  }
96
97  // 执行代码行
98  bool executeLine (const string& line, unsigned& lineno) {
99    if(!beginWith(line, "FOR")) {
100     // 若不以 FOR 开头，则为单一语句
101     return executeStatement(line, lineno);
102   } else {
103     int cursor = 0;
104     sscanf(line.c_str(), "%*s %[^\n]", buffer[0]);
105     string blendExpr(buffer[0]);
106     int times = Parser::expr(blendExpr,
        ↪  cursor).resolve(symbols);
107     // 不需要传递代码行， 用 dummy 代替
108     unsigned dummy = 0;
109     for(int i = 0; i < times; i++) {
110       executeLine(blendExpr.substr(cursor), dummy);
111     }
112     lineno++;
113     return true;
114   }
115 }
116
117 int main () {
118   readProgram();
119   // 当前执行行号
120   unsigned lp = 0;
121   // 主循环
122   while(lp < program.size()) {
123     executeLine(program[lp], lp);
124   }
125 }
```

2.4 调试报告

2.4.1 遇到的主要问题和解决方案

在编写程序的过程中遇到的主要问题是不能很好地处理负号表达式和运算结合性，最终通过将递归算法调整为迭代算法进行解决

2.4.2 对于设计编码的回顾讨论和分析

- 设计编码中使用了定义实现分离，代码可读性更强
- 设计编码中使用了过多全局变量，代码风格不够好

2.4.3 时空分析

表达式解析器 通过 LL1 算法，表达式解析器大致能做到 $O(N)(N$ 为表达式长度) 的时间复杂度，$O(M)(M$ 为符号数) 的空间复杂度

解释器 通过 hash 表，解释器可以在 $O(1)$ 时间内进行符号值的查找和更改，整个执行过程中时间复杂度为 $O(N)(N$ 为代码长度)

2.4.4 改进设想

- 添加交互模式，语句一经输入立刻执行
- 添加对数组的支持
- 添加条件变量和条件语句
- 添加 IO 语句

2.5 运行结果展示

图 3: 计算从 1 加到 4

```
DEFINE add
ASSIGN sum sum + i
ASSIGN i i + 1
END

ASSIGN sum 0
ASSIGN i 1
FOR 4 CALL add

Defining add
Assigning 0 to sum
Assigning 1 to i
Calling add
Assigning 1 to sum
Assigning 2 to i
Calling add
Assigning 3 to sum
Assigning 3 to i
Calling add
Assigning 6 to sum
Assigning 4 to i
Calling add
Assigning 10 to sum
Assigning 5 to i
```

图 4: 题目测试

```
ASSIGN X 1
ASSIGN Y 1

DEFINE Fib
ASSIGN TMP Y
ASSIGN Y X+Y
ASSIGN X TMP
END

CALL Fib
ASSIGN W X
CALL Fib
ASSIGN Z W*Y-X*X

Assigning 1 to X
Assigning 1 to Y
Defining Fib
Calling Fib
Assigning 1 to TMP
Assigning 2 to Y
Assigning 1 to X
Assigning 1 to W
Calling Fib
Assigning 2 to TMP
Assigning 3 to Y
Assigning 2 to X
Assigning -1 to Z
```

图 5: 嵌套FOR循环

```
ASSIGN x 0
FOR 2*2 FOR 2*2 ASSIGN x x + 1

Assigning 1 to x
Assigning 2 to x
Assigning 3 to x
Assigning 4 to x
Assigning 5 to x
Assigning 6 to x
Assigning 7 to x
Assigning 8 to x
Assigning 9 to x
Assigning 10 to x
Assigning 11 to x
Assigning 12 to x
Assigning 13 to x
Assigning 14 to x
Assigning 15 to x
Assigning 16 to x
```

图 6: 测试表达式树解析

```
1+1*2-3+aLongVarible/x%2
(+ (- (+ 1 (* 1 2)) 3) (% (/ aLongVarible x) 2))
```

3 实验总结

通过这次实验我们系统地学习了 Union-Find 数据结构，并且学习了一些编译原理的算法。在进行这次实验的过程中，我们适当地使用了 C++ 的面向对象特性，简化了代码增加了代码可读性。在项目管理方面，我们使用了`make`命令通过撰写 Makefile 来进行自动化构建，并且使用了`git`进行版本控制和团队协作。总体上来说在各个方面都有不小的收获。

参考文献

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein 算法导论. 机械工业出版社，2013.1