**Московский авиационный институт**

**(Национальный исследовательский университет)**

Факультет: «Информационные технологии и прикладная математика»

Кафедра: 806 «Вычислительная математика и программирование»

Дисциплина: «Системное программирование»

**Курсовой проект**

Тема: Построение КС-грамматики

Студент: Тояков А. О.

Группа: М8о-307б-18

Преподаватель: Чернышов Л. Н.

Дата:

Оценка:

Оглавление

[1.Постановка задачи 3](#__RefHeading___Toc822_4107025393)

[2.Структура программы 3](#__RefHeading___Toc1659_3709866927)

[3.Описание программы 3](#__RefHeading___Toc1661_3709866927)

[4.Листинг программы 4](#__RefHeading___Toc1663_3709866927)

[5.Результат работы 14](#__RefHeading___Toc1665_3709866927)

[6.Вывод 14](#__RefHeading___Toc1667_3709866927)

[7.Список литературы 15](#__RefHeading___Toc1082_4107025393)

# 

# ПОСТАНОВКА ЗАДАЧИ

Разработать КС-грамматику для строк, задающих множество отрезков на прямой. Например, { [1,2], (3, 5.1) } . В отрезке отмечаются, входят ли границы. Для бесконечности предложить какой-либо знак.

Разработать класс на С++, описывающий такие множества. Должны быть методы:

1. Проверка корректности.
2. Нормализация множества отрезков: отрезки не пересекаются и упорядочены по возрастанию.

# СТРУКТУРА ПРОГРАММЫ

Структура моего проекта:

1. src/line.hpp
2. src/CfreeG.hpp
3. main.cpp
4. CmakeLists.txt

# ОПИСАНИЕ ПРОГРАММЫ

Для того, чтобы решить поставленную задачу мне нужно реализовать:

1. Проверка корректности.
2. Нормализация множества отрезков: отрезки не пересекаются и упорядочены по возрастанию.

В моем классе присутствуют такие приватные поля:

std::string line; // входная строка.

std::vector<std::shared\_ptr<Line<long long>>> pairs; // вектор с моими интервалами.

std::vector<std::pair<size\_t,size\_t>> startPositions; // вектор со значениями начальных и конечных позиций интервалов.

bool normalize; // статут моего множества.

**Проверка корректности:** Для начала я спроектировал грамматику:

Context-free grammar:

S->{A

A->[a,b]B|(a,b)B|[a,b)B|(a,b]B|};

B→,A|}

Знак бесконечности: &. В моем множестве принимаются значения -&,+&, расположенные в круглых скобках.

Реализовал я данную грамматику с помощью конструкции switch — case, где состояниями конструкции являются: S, A, B, End, Error (**фунуция: bool LLParsing(const std::string& str)**). Также для обработки терминальных значений (например: [a,b]), я также использовал конструкцию switch — case, где состояниями конструкции являются: FD, SD, EndB, ErrorB(**функция: bool OutBurst(const std::string& str, size\_t& i)**). Разбор идет слева-направо. В процессе разбора множества интервалов, вектор StartPositions инициализируется необходимыми значениями. Также при разборе проверяется корректность значений, расположенных в интервалах.

**Нормализация множества отрезков:** Основная идея: Добавляю по одному интервалу в вектор с интервалами B, сравнивая с уже установленными значениями в векторе B (**функция: AddInVector(std::string& str)**). Затем же сортирую данный вектор B, по первому значению в интервале (использую std::sort). После сортировки сливаю интервалы в векторе B,если это необходимо, если интервал i уже невозможно слить c i + 1, то i++(**функция: NormalizeVector(std::string& str)**).

# ЛИСТИНГ ПРОГРАММЫ

**//main.cpp**

#include "src/CfreeG.hpp"

void PrintRules() {

std::cout << "Context-free grammar" << std::endl;

std::cout << "Infinity:&" << std::endl;

std::cout << "long long int - type" << std::endl;

std::cout << "S->{A" << std::endl;

std::cout << "A->[a,b]B || (a,b)B ||[a,b)B || (a,b]B || }" << std::endl;

std::cout << "B->,A || } " << std::endl;

}

void PrintMain() {

std::cout << "Enter go:work" << std::endl;

std::cout << "Enter end:exit" << std::endl;

std::cout << "Enter help:help" << std::endl;

}

// ConvertInString

int main() {

std::string out;

PrintRules();

while(std::cin >> out) {

if(out == "go") {

try {

std::string line;

std::cin >> line;

ContextFree\* current = new ContextFree(line);

current->CheckNormalize();

delete current;

} catch (std::logic\_error& err) {

std::cout << err.what() << std::endl;

}

} else if(out == "end") {

break;

} else if(out == "help") {

PrintMain();

}

}

return 0;

}

**//CfreeG.hpp**

#ifndef CFREEG\_HPP

#define CFREEG\_HPP

#include <iostream>

#include <vector>

#include <algorithm>

#include <memory>

#include "line.hpp"

#include <stdlib.h>

// infinity &;

/\*

Context-free grammar

S->{A

A->[a,b]B|(a,b)B|[a,b)B|(a,b]B|};

B->,A|}

\*/

class ContextFree {

public:

ContextFree(const std::string& str) {

//PrintRules();

if(LLParsing(str) == false) {

throw std::logic\_error("Pattern is wrong");

}

line = std::move(str);

normalize = false;

}

void PrintRules() {

std::cout << "Context-free grammar" << std::endl;

std::cout << "Infinity:&" << std::endl;

std::cout << "long long int -` type" << std::endl;

std::cout << "S->{A" << std::endl;

std::cout << "A->[a,b]B || (a,b)B ||[a,b)B || (a,b]B || }" << std::endl;

std::cout << "B->,A || } " << std::endl;

}

// if(normalize == true) -> nothing, else normalize line and print this line!

void CheckNormalize() {

if(normalize == false) {

Normalize(line);

}

std::cout << "Normalize: " << std::endl;

std::cout << line << std::endl;

}

~ContextFree() {

line = "";

pairs.clear();

startPositions.clear();

normalize = false;

}

private:

std::string line;

std::vector<std::shared\_ptr<Line<long long>>> pairs;

std::vector<std::pair<size\_t,size\_t>> startPositions;

bool normalize;

enum States {

S,

A,

B,

End,

Error,

};

enum ConstStates {

FD,

SD,

EndB,

ErrorB

};

// 2

// add

void AddInVector(std::string& str) {

std::shared\_ptr<Line<long long>> current(new Line<long long>(str));

auto it = pairs.begin();

bool merge = false;

while((merge == false) && (it != pairs.end())) {

merge = ((\*it)->Merge(current));

it++;

}

if(merge == false) {

pairs.push\_back(current);

}

}

// sort vector

void SortVector() {

std::sort(pairs.begin(), pairs.end(), [](std::shared\_ptr<Line<long long>>& first,

std::shared\_ptr<Line<long long>>& second) {

return (first->a < second->a);

});

}

void PrintVector() {

std::for\_each(pairs.begin(),pairs.end(),[](std::shared\_ptr<Line<long long>> o)

{

return o->Print();

});

}

void NormalizeVector() {

if(pairs.begin() == pairs.end()) {

return;

}

auto it = pairs.begin();

auto next = pairs.begin() + 1;

while(next != pairs.end()) {

if((\*it)->b >= (\*next)->a) {

if((\*next)->b > (\*it)->b) { // [1,7],[2,8] -> [1,8]

(\*it)->b = (\*next)->b;

(\*it)->right = (\*next)->right;

} else if((\*next)->b == (\*it)->b) { // [1,7],[2,7] -> [1,7]

if((\*it)->b == true || (\*next)->b == true) {

(\*it)->right = true;

}

} else if((\*it)->b == (\*next)->a) { // [1,7],(7,13) -> [1,13)

if((\*next)->a == true || (\*it)->b == true) {

(\*it)->b = (\*next)->b;

(\*it)->right = (\*next)->right;

}

}

pairs.erase(next);

} else { // [1,7], (8,10) -> const;

it++;

next++;

}

}

}

// convert

void ConvertInString() {

std::string newLine = "{";

int count = 0;

for(auto it = pairs.begin(); it != pairs.end(); it++) {

std::string interval = "";

if((\*it)->left == true) {

interval = interval + "[";

} else {

interval = interval + "(";

}

if((\*it)->a == std::numeric\_limits<long long>::min()) {

interval = interval + "-&";

} else {

interval = interval + std::to\_string((\*it)->a);

}

interval = interval + ",";

if((\*it)->b == std::numeric\_limits<long long>::max()) {

interval = interval + "+&";

} else {

interval = interval + std::to\_string((\*it)->b);

}

if((\*it)->right == true) {

interval = interval + "]";

} else {

interval = interval + ")";

}

count++;

if(count != 1) {

newLine = newLine + ",";

newLine = newLine + interval;

} else {

newLine = newLine + interval;

}

}

newLine += "}";

line = std::move(newLine);

}

// all

void Normalize(const std::string& str) {

for(auto it = startPositions.begin(); it != startPositions.end(); it++) {

std::string interval = str.substr(it->first, (it->second - it->first) + 1);

AddInVector(interval);

}

SortVector();

//PrintVector();

NormalizeVector();

//PrintVector();

ConvertInString();

normalize = true;

}

// 1

// check and left and right digit in brackets!

bool CheckInfinity(const bool& lInf,const bool& rInf,const bool& lSign,const bool& rSign) {

if((lInf == true && lSign == true) || (rInf == true && rSign == false)) {

return false;

}

return true;

}

bool CheckDigit(const bool& lSign,const bool& rSign,const int& lDigit,const int& rDigit) {

if(lDigit == 0 && rDigit == 0) {

return false;

}

if(lSign == false && rSign == false) {

if(rDigit >= lDigit) {

return false;

}

} else if(lSign == true && rSign == false) {

return false;

} else if(lSign == true && rSign == true) {

if(lDigit >= rDigit) {

return false;

}

}

return true;

}

bool CheckInfinityDigit(const bool& lSign, const bool& lInf) {

if(lInf == true && lSign == true) {

return false;

}

return true;

}

bool CheckDigitInfinity(const bool& rSign, const bool& rInf) {

if(rInf == true && rSign == false) {

return false;

}

return true;

}

// Get digit or infinity

void GetDigit(const std::string& str, size\_t& i, bool& sign, int& digit, bool& statusDigit) {

if((str[i] == '-') || (str[i] == '+')) {

if(str[i] == '-') {

sign = false;

}

i++;

}

while((i < str.length()) && (std::isdigit(str[i]))) {

statusDigit = true;

char current = str[i];

digit = digit \* 10 + (current - '0');

i++;

}

}

bool GetLowInfinity(const std::string& str, size\_t& i, bool& sign, bool& inf) {

if((i + 2 < str.length()) && ((str[i] == '-') && (str[i + 1] == '&'))) {

inf = true;

sign = false;

i = i + 2;

return true;

}

return false;

}

bool GetHighInfinity(const std::string& str, size\_t& i, bool& sign, bool& inf) {

if((i + 2 < str.length()) && ((str[i] == '+') && (str[i + 1] == '&'))) {

inf = true;

sign = true;

i = i + 2;

return true;

}

return false;

}

// Outburst [a,b] || (a,b) || [a,b) || (a,b]

bool OutBurst(const std::string& str, size\_t& i) {

int lBracket = i;

if((str[i] != '[') && (str[i] != '(')) {

return false;

}

i++;

ConstStates state = FD;

bool lInf = false, rInf = false;

bool lSign = true, rSign = true;

bool statusLDigit = false, statusRDigit = false;

int lDigit = 0, rDigit = 0;

while(state != ErrorB && state != EndB) {

switch(state) {

case FD:

if(GetLowInfinity(str, i, lSign, lInf) == false) {

GetDigit(str, i, lSign, lDigit, statusLDigit);

}

if((str[i] == ',') && (lInf == true || statusLDigit == true)) {

i++;

state = SD;

} else {

state = ErrorB;

}

break;

case SD:

if(GetHighInfinity(str, i, rSign, rInf) == false) {

GetDigit(str,i,rSign,rDigit,statusRDigit);

}

if((str[i] == ')' || str[i] == ']') && (rInf == true || statusRDigit == true)) {

startPositions.push\_back(std::make\_pair(lBracket,i));

state = EndB;

} else {

state = ErrorB;

}

break;

case EndB:

break;

case ErrorB:

break;

}

}

if(state == ErrorB) {

return false;

}

// only (-&,+&) true;

if((str[lBracket] != '(' && lInf == true) || (str[i] != ')' && rInf == true)) {

return false;

}

bool st;

if(lInf == true && rInf == true) {

st = CheckInfinity(lInf, rInf, lSign, rSign);

} else if(statusLDigit == true && statusRDigit == true) {

st = CheckDigit(lSign, rSign, lDigit, rDigit);

} else if(lInf == true && statusRDigit == true) {

st = CheckInfinityDigit(lSign, lInf);

} else if(statusLDigit == true && rInf == true) {

st = CheckDigitInfinity(rSign, rInf);

}

return st;

}

// LL-parsing;

bool LLParsing(const std::string& str) {

size\_t length = str.size();

size\_t i = 0;

States state = S;

bool check;

while(i < length) {

char current = str[i];

switch(state) {

case S:

if(current == '{') {

i++;

state = A;

} else {

i = length - 1;

state = Error;

}

break;

case A:

if(current == '}') {

state = End;

}

else if(OutBurst(str, i) == true && (i < length - 1)) {

i++;

state = B;

} else {

i = length - 1;

state = Error;

}

break;

case B:

if(current == ',') {

i++;

state = A;

} else if(current == '}') {

state = End;

} else {

i = length - 1;

state = Error;

}

break;

case End:

if(i == length - 1) {

check = true;

i = length;

} else {

check = false;

i = length;

}

break;

case Error:

check = false;

i = length;

break;

}

}

return check;

}

};

#endif

**//line.hpp**

#ifndef LINE\_HPP

#define LINE\_HPP

/\*

left -> true = [, false = (

\*/

// infinity = std::numeric\_limits;

//

template < class T>

class Line {

public:

friend class ContextFree;

Line(std::string& str) {

auto it = str.begin();

this->left = false;

this->a = 0;

this->b = 0;

this->right = false;

CheckBracket(it);

T lSign;

CheckSign(it, lSign);

if(\*it == '&') {

this->a = std::numeric\_limits<T>::min();

it++;

} else {

GetDigit(it,a);

this->a = lSign \* a;

}

it++;

T rSign;

CheckSign(it, rSign);

if(\*it == '&') {

this->b = std::numeric\_limits<T>::max();

it++;

} else {

GetDigit(it,b);

this->b = rSign \* b;

}

CheckBracket(it);

}

Line(bool nleft, bool nlinfinity, T& na, T& nb, bool nright, bool nrinfinity) :

left(nleft), a(na), b(nb), right(nright)

{}

void Print() {

std::cout << left << "," << a << "," << b << "," << right << std::endl;

}

bool Merge(std::shared\_ptr<Line<T>>& other) {

int st = 0;

if(this->a < other->a && this->b > other->b) /\* a oa ob b \*/ {

//std::cout << "1" << std::endl;

return true;

} else if(other->a <= this->a && other->b >= this->b) /\* oa <-a b-> ob \*/ {

if(other->a == this->a) {

if(other->left == true) {

this->left = true;

}

} else if(other->a < this->a) {

this->a = other->a;

this->left = other->left;

}

if(other->b == this->b) {

if(other->right == true) {

this->right = true;

}

} else if(other->b > this->b) {

this->b = other->b;

this->right = other->b;

}

//std::cout << "2" << std::endl;

st++;

} else if(((this->a == other->b) && (this->left == true || other->right == true)) ||

((other->a < this->a) && (this->b > other->b && this->a < other->b))) /\* oa a,ob-> b \*/ {

this->a = other->a;

this->left = other->left;

st++;

//std::cout << "3" << std::endl;

} else if(((other->a == this->b) && (this->right == true || other->left == true)) ||

((other->a > this->a && this->b > other->a) && (other->b > this->b))) /\* a oa-> b ob \*/ {

this->b = other->b;

this->right = other->right;

st++;

//std::cout << "4" << std::endl;

}

return st > 0;

}

private:

bool left;

T a;

T b;

bool right;

void CheckBracket(std::string::iterator& it) {

if(\*it == '[') {

left = true;

}

if(\*it == ']') {

right = true;

}

it++;

}

void CheckSign(std::string::iterator& it,T& sign) {

sign = 1;

if(\*it == '+') {

it++;

}

if(\*it == '-') {

sign = -1;

it++;

}

}

void GetDigit(std::string::iterator& it,T& a) {

while(std::isdigit(\*it)) {

a = a \* 10 + (\*it - '0');

it++;

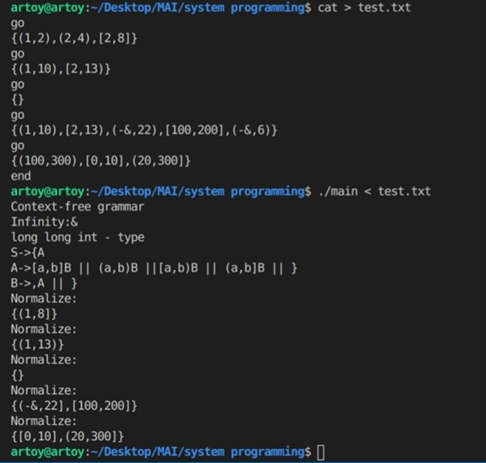
}

}

};

#endif

# РЕЗУЛЬТАТ РАБОТЫ



# ВЫВОД

Таким образом, в данной работе я спроектировал КС-грамматику (частный случай формальной грамматики (тип 2 по иерархии Хомского), у которой левые части всех продукций являются одиночными нетерминалами). В данной работе мне пришлось реализовать два детерминированных автомата: один и есть моя КС-грамматика, а другой соответственно для обработки терминальных символов: [a,b], (a,b) и т. д. Также благодаря данной работе я изучил и использовал несколько новых функций из алгоритмов STL.

# СПИСОК ЛИТЕРАТУРЫ

Порождающие и распознающие системы формальных языков - А.С. Семенов.