一阶公式的实例化

**问题概述：**

输入一个一阶闭公式集，以这个公式集中出现的常量的集合为论域，将这个一阶公式集转化为与它等价的命题公式集。

A1.设计一个语言可以表述上述的一阶公式集，给出这个语言的词法与语法；

A2.对输入的文件进行语法分析:

定义相应的数据结构保存这个公式集和其中出现的常量；

定义相应的语法树，并且可以输出语法分析的结果；

当出现输入错误时，可以指出出错的位置和可能的原因；

A3.将这个公式集实例化并化简,将结果以文本的形式输出；

A4.判断输出的命题逻辑公式集是否是可满足的，如果是则给出一个模型

**系统设计：**

**简述：**

本系统是一个一阶公式实例化工具，主要功能包括：

1. 以文本形式输入一阶公式实例化工具
2. 展示生成语法树
3. 将这个公式集实例化并化简，结果从屏幕输出并保存到相应文本文件
4. 输出命题逻辑公式集是否是可满足的，如果是则输出各变量取值。

同时将结果保存到相应文本文件。

**开发环境：**

操作系统：win7

编译环境：java jdk1.8

开发工具：eclipse、Antlr4（支持包）

**词法与文法设计：**

下面是本系统安照Antlr4要求的格式编写的.g文件（）

///////////////////////////////////////cai.g

--------------------以下为文法

grammar cai;

program:(example ';')\*;

example:sentence;

sentence:lcexp|logexp|compexp|exp;

lcexp:((qu)varexp) (preexp|(lk sentence rk)|lcexp);

logexp:tcexp;

tcexp: orexp(key orexp)\*;

key:(sd|bd);

orexp:andexp (or andexp)\*;

andexp:compexp (and compexp)\*;

compexp: exp ((eq|bl|ab|be|ae|ne) exp)?;

exp:Var | conexp | (not)?(lk sentence rk) | preexp |lcexp;

preexp:(not)? pexp lk(varexp|conexp) rk;

varexp:Var;

conexp:Con;

pexp:Pre;

------------------以下为词法

Pre:'@' Lowercase+;

Var:Uppercase+;

Con:(Digit(Digit)\*)|(Lowercase(Digit|Lowercase)\*);

Lowercase:[a-z];

Uppercase:[A-Z];

Digit: [0-9];

not:'~';

and:'&&';

or:'||';

sd:'->';

bd:'<->';

qu:'%'|'$';

eq:'=';

ab:'>';

bl:'<';

be:'<=';

ae:'>=';

ne:'!=';

lk:'(';

rk:')';

WS : [' '\t\r\n ]+ -> skip;

**生成java文件：**

1. 在win7中配置antlr4的运行环境；

网址如下：http://blog.csdn.net/shine\_journey/article/details/17952163

（2）利用控制台，在控制台中进入到cai.g的路径下，输入以下命令生成java文件：

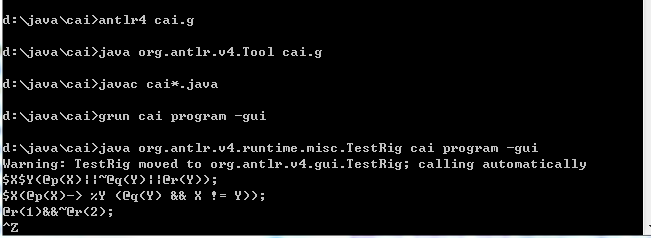
antlr4 cai.g

（3）此外可以利用以下命令生成class文件以及运行程序：

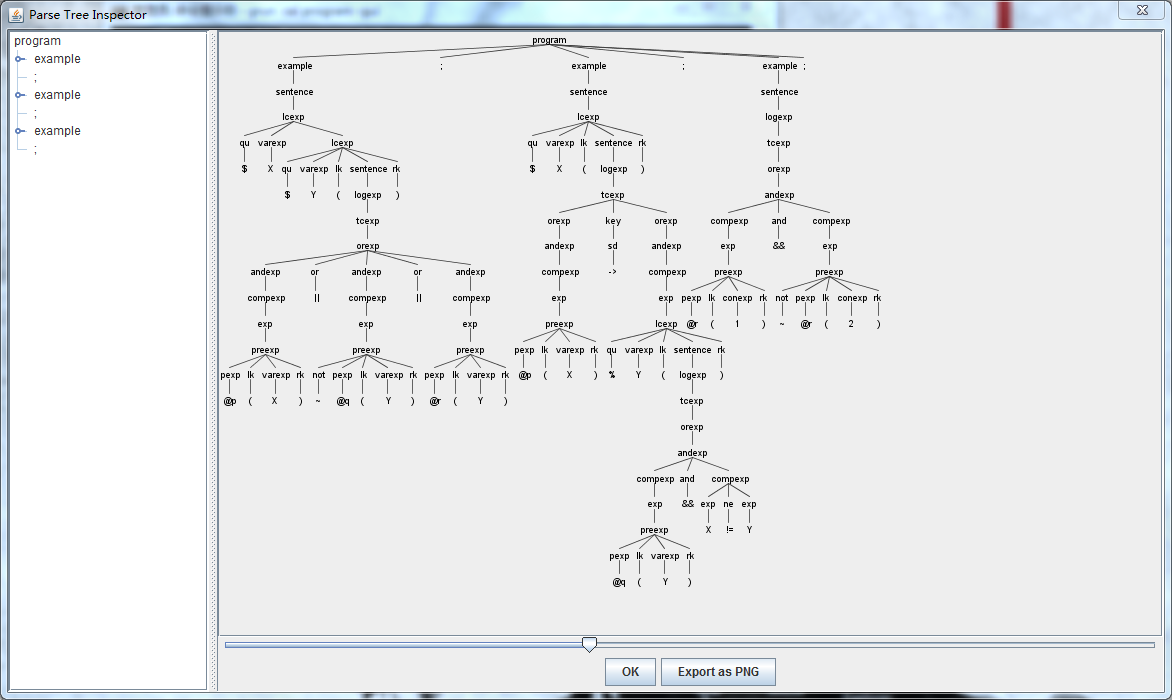
javac cai\*.java

grun cai program -gui （其中program为文法中的程序入口）

具体如下（其中未提示部分为自动生成的指令）：



生成的语法树：



**设计步骤：**

本系统利用antlr4生成语法分析程序，将上面生成的java文件以及antlr-4.5.3-complete.jar包导入到eclipse项目中，主要通过继承caiBaseListenter，实现各种的功能，在继承的类中重载添加我们需要的语义动作，达到我们的需要的寻找论域、代入、化简等效果。

设计详情：

0.读入数据以及其他预处理

**public** **static** String readInput() {

Scanner sc = **new** Scanner(System.***in***);

System.***out***.println("\n请输入公式集于Input.txt中，输入完以后保存并在控制台回车开始程序:");

sc.nextLine();

FileUtils Fileread=**new** FileUtils ();

String input;

**try** {

input = Fileread.*readFile*("./Input.txt");

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

**return** input;

}

**public** **static** caiParser getContext(String nodeSet) {

ANTLRInputStream inputStream = **new** ANTLRInputStream(nodeSet);

caiLexer lexer = **new** caiLexer(inputStream);

CommonTokenStream tokens = **new** CommonTokenStream(lexer);

caiParser parser = **new** caiParser(tokens);

**return** parser;

}

**public** **static** caiParser getContext(String nodeSet) {

ANTLRInputStream inputStream = **new** ANTLRInputStream(nodeSet);

caiLexer lexer = **new** caiLexer(inputStream);

CommonTokenStream tokens = **new** CommonTokenStream(lexer);

caiParser parser = **new** caiParser(tokens);

**return** parser;

}

1. 论域以及谓词常量的确定

我们先扫描一遍输入公式集，重载exitConexp()统计出全部的常量，将其存放在一个mainAntlr类中的constValue中待用；重载exitPreexp()，统计出带常量的谓词语句，并判断其真值，保存于preValue 中待用。

package com.cai.handleListenter;

import java.util.HashMap;

import java.util.Map;

import com.cai.mainAntlr;

import com.cai.antlr.caiBaseListener;

import com.cai.antlr.caiParser;

public class constantListenter extends caiBaseListener {

/\*

\* get constant to constitute the domain of discourse

\*/

@Override

public void exitConexp(caiParser.ConexpContext ctx) {

super.exitConexp(ctx);

// ctx.Con().getText();

mainAntlr.constValue.put(ctx.Con().getText(), true);

}

/\*

\* get constant to constitute the domain of discourse

\*/

@Override

public void exitPreexp(caiParser.PreexpContext ctx) {

super.exitPreexp(ctx);

if (ctx.conexp() != null) {

if(ctx.not()==null){

mainAntlr.preValue.put(ctx.getText(), true);

//StringBuilder sb = new StringBuilder(ctx.getText());

//sb =sb.insert(0, "~");

//mainAntlr.preValue.put(sb.toString(), false);

}else

{

//mainAntlr.preValue.put(ctx.getText(), false);

String notStr = ctx.getText().substring(1, ctx.getText().length());

mainAntlr.preValue.put(notStr, false);

}

}

}

}

通过在main函数中调用以下函数，即可统计出论域以及谓词常量：

caiParser parser1 = *getContext*(input);

caiParser.ProgramContext root1 = parser1.program();

constantListenter constantlistener = **new** constantListenter();

ParseTreeWalker walker = **new** ParseTreeWalker();

walker.walk(constantlistener, root1);

1. 代入

同样继承caiBaseListenter，在类中建立一个map为mySentence，用于保存每一步新的遍历时之前的代入结果，主要利用的策略是在重载exit\*的函数，将保存于mySentence中的之前的替换的结果先替换，在利用寻找到的论域，将当前的exitLcexp语句进行重载，将存在量词与全称量词所的作用域中的语句进行常量代入替换，同时删除之前mySentence中属于该部分的子语句，避免干扰；在每次进入一个新的语句时清空以前的语句所做的替换，避免干扰；在退出当前语句时将新生成的代入结果追加到newStr中，以后后面化简使用。具体实现如下，代码实现有具体实现步骤解释：

package com.cai.handleListenter;

import java.util.HashMap;

import java.util.Map;

import java.util.Set;

import com.cai.mainAntlr;

import com.cai.antlr.caiBaseListener;

import com.cai.antlr.caiParser;

//$X(@r(X)||%Y@q(Y));

public class parseListenter extends caiBaseListener {

static private Map<String, String> mySentence = new HashMap<String, String>();

//每一次进去一个新的sentence的时候清空以前的语句所做的替换，避免干扰

@Override

public void enterExample(caiParser.ExampleContext ctx) {

super.enterExample(ctx);

if (mySentence.size() > 0)

mySentence.clear();

}

//拼凑出在正则表达式中不会被转义的新语句，因部分一阶逻辑中使用符号为正则表达式中的保留字符，故需要添加'\\',避免混淆

public String replace(String str) {

str = str + ",";

StringBuilder sb = new StringBuilder(str);

String finalStr = "";

char temp = 0;

for (int i = 0; temp != ','; i++) {

temp = sb.charAt(i);

if (temp == '(' || temp == ')' || temp == '$' || temp == '|') {

sb.insert(i, "\\");

i += 1;

}

}

finalStr = sb.toString();

finalStr = finalStr.substring(0, finalStr.length() - 1);

return finalStr;

}

//退出当前语句时将新生成的代入结果追加到newStr中，以后后面化简使用

@Override

public void exitExample(caiParser.ExampleContext ctx) {

super.exitExample(ctx);

// 替换以前已经做过的子式

String Str = ctx.getText() + ";\r\n";

Set<String> mySet = mySentence.keySet();

for (String key : mySet) {

String Keychange = replace(key);

String Str1 = Str.replaceAll(Keychange, mySentence.get(key));

Str = Str1;

}

mainAntlr.newStr += (Str);

}

//具体代入实现

private void deal(String subStr, boolean f, caiParser.LcexpContext ctx) {

caiParser.QuContext qu = ctx.qu();

String qt = qu.getText();// 取得$或%

//替换以前出现的代入结果

Set<String> mySet = mySentence.keySet();

for (String key : mySet) {

String Keychange = replace(key);

subStr = subStr.replaceAll(Keychange, mySentence.get(key));

}

// 处理新的情况

String VarexpStr = ctx.varexp().getText();

String str = "";

boolean flag = false;

for (String myvalue : mainAntlr.keySet) {

String node = subStr.replaceAll(VarexpStr, myvalue);

if (f)

node = '(' + node + ')';

if (qt.equals("%")) {

if (flag)

str += ("||" + node);

else {

str = node;

flag = !flag;

}

} else {

if (flag)

str += ("&&" + node);

else {

str = node;

flag = !flag;

}

}

}

str = "(" + str + ")";

mySentence.clear();

mySentence.put(ctx.getText(), str);

}

/\*

\* parse all and exist condition，将存在量词与全称量词所的作用域中的语句进行常量代入替换

\*/

@Override

public void exitLcexp(caiParser.LcexpContext ctx) {

super.exitLcexp(ctx);

if(mainAntlr.keySet.size()==0)

return ;

String subStr = null;

caiParser.PreexpContext Preexp = ctx.preexp();

//讨论三种不同的两次语句情况

if (ctx.preexp() != null) {

subStr = ctx.preexp().getText();

deal(subStr, false, ctx);

} else if (ctx.sentence() != null) {

subStr = ctx.sentence().getText();

deal(subStr, true, ctx);

} else if (ctx.lcexp() != null) {

subStr = ctx.lcexp().getText();

deal(subStr, true, ctx);

}

// 替换以前已经做过的子式

}

}

在main函数中实现以下调用即可实现：

parseListenter parseListenter = **new** parseListenter();

walker.walk(parseListenter, root1);

System.***out***.println(*newStr*);

1. 化简

依然是继承caiBaseListenter类，主要利用的策略是在重载exit\*的函数，将保存于mySentence中的之前的替换的结果先替换，利用COMPexp，ORexp，ANDORexp，TCexp语句的特点进行化简，因为这些语句包含了II，&&，->,<->以及比较运算符，在某些情况下可以确定其真值或是消去无用的部分，并将新结果保存于mySentence中，同时删除之前mySentence中属于该部分的子语句，避免干扰；在每次进入一个新的语句时清空以前的语句所做的替换，避免干扰；在退出当前语句时将新生成的代入结果追加到newStr中，以后后面化简使用。实现如下，代码实现有具体实现步骤解释：

package com.cai.handleListenter;

import java.util.ArrayList;

import java.util.HashMap;

import java.util.List;

import java.util.Map;

import java.util.Set;

import java.util.Stack;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

import com.cai.mainAntlr;

import com.cai.antlr.caiBaseListener;

import com.cai.antlr.caiParser;

import com.cai.antlr.caiParser.AndContext;

public class simplifyListenter extends caiBaseListener {

static private Map<String, String> mySentence = new HashMap<String, String>();

static int a = 0;

//拼凑出在正则表达式中不会被转义的新语句，因部分一阶逻辑中使用符号为正则表达式中的保留字符，故需要添加'\\',避免混淆

public String replace(String str) {

str = str + ",";

StringBuilder sb = new StringBuilder(str);

String finalStr = "";

char temp = 0;

for (int i = 0; temp != ','; i++) {

temp = sb.charAt(i);

if (temp == '(' || temp == ')' || temp == '$' || temp == '|') {

sb.insert(i, "\\");

i += 1;

}

}

finalStr = sb.toString();

finalStr = finalStr.substring(0, finalStr.length() - 1);

return finalStr;

}

//每一次进去一个新的sentence的时候清空以前的语句所做的替换，避免干扰

@Override

public void enterExample(caiParser.ExampleContext ctx) {

super.enterExample(ctx);

if (mySentence.size() > 0)

mySentence.clear();

}

//退出当前语句时将新生成的代入结果追加到newStr中，以后后面可满足性判断使用

@Override

public void exitExample(caiParser.ExampleContext ctx) {

super.exitExample(ctx);

String Str = ctx.getText() + ";\r\n";

Set<String> mySet = mySentence.keySet();

for (String key : mySet) {

String Keychange = replace(key);

String Str1 = Str.replaceAll(Keychange, mySentence.get(key));

Str = Str1;

}

if (Str.equals("false;\r\n"))

mainAntlr.flag = false;

if(!Str.equals("true;\r\n"))

mainAntlr.newStr += Str;

//System.out.println(Str);

}

//化简AND语句中的可化简部分

@Override

public void exitAndexp(caiParser.AndexpContext ctx) {

super.exitAndexp(ctx);

if (ctx.and().size() > 0) {

String s = ctx.getText();

String str = null;

caiParser.CompexpContext Compexp = null;

String temp = "";

int i;

String ss = ctx.getText();

//处理多AND情况

for (i = 0; i < ctx.compexp().size(); i++) {

Compexp = ctx.compexp(i);

String Comstr = Compexp.getText();

// 替换以前已经做过的子式，并删除该子式

Set<String> mySet = mySentence.keySet();

List<String> nodes = new ArrayList<String>();

for (String key : mySet) {

String Keychange = replace(key);

String Str1 = Comstr.replaceAll(Keychange,

mySentence.get(key));

if (!Str1.equals(Comstr))

nodes.add(key);

Comstr = Str1;

}

for (String node : nodes)

mySentence.remove(node);

//代入新式子中出现的已知真值的谓词常量

mySet = mainAntlr.preValue.keySet();

for (String key : mySet) {

String KeychangeT = replace(key);

String KeychangeF = '~' + KeychangeT;

String Str1 = Comstr.replaceAll(KeychangeF,

!mainAntlr.preValue.get(key)+"");

Str1 = Str1.replaceAll(KeychangeT,

mainAntlr.preValue.get(key).toString());

Comstr = Str1;

}

//具体化简过程

if (Comstr.equals("false")) {

temp = "false";

break;

} else {

if (i == 0)

if (Comstr.equals("true"))

temp = "";

else

temp = Comstr;

else if (!Comstr.equals("true"))

if (temp.equals(""))

temp = Comstr;

else

temp = temp + "&&" + Comstr;

}

}

if (temp.equals(""))

mySentence.put(ctx.getText(), "true");

else

mySentence.put(ctx.getText(), temp);

}

}

//化简OR语句中的可化简部分

@Override

public void exitOrexp(caiParser.OrexpContext ctx) {

super.exitOrexp(ctx);

if (ctx.or().size() > 0) {

String s = ctx.getText();

String str = null;

caiParser.AndexpContext Compexp = null;

String temp = "";

int i;

String ss = ctx.getText();

//处理多OR情况

for (i = 0; i < ctx.andexp().size(); i++) {

Compexp = ctx.andexp(i);

String Comstr = Compexp.getText();

// 替换以前已经做过的子式，并删除该子式

Set<String> mySet = mySentence.keySet();

List<String> nodes = new ArrayList<String>();

for (String key : mySet) {

String Keychange = replace(key);

String Str1 = Comstr.replaceAll(Keychange,

mySentence.get(key));

if (!Str1.equals(Comstr))

nodes.add(key);

Comstr = Str1;

}

for (String node : nodes)

mySentence.remove(node);

//代入新式子中出现的已知真值的谓词常量

mySet = mainAntlr.preValue.keySet();

for (String key : mySet) {

String KeychangeT = replace(key);

String KeychangeF = '~' + KeychangeT;

String Str1 = Comstr.replaceAll(KeychangeF,

!mainAntlr.preValue.get(key)+"");

Str1 = Str1.replaceAll(KeychangeT,

mainAntlr.preValue.get(key).toString());

Comstr = Str1;

}

//具体化简过程

if (Comstr.equals("true")) {

temp = "true";

break;

} else {

if (i == 0)

if (Comstr.equals("false"))

temp = "";

else

temp = Comstr;

else if (!Comstr.equals("false"))

if (temp.equals(""))

temp = Comstr;

else

temp = temp + "||" + Comstr;

}

}

if (temp.equals(""))

mySentence.put(ctx.getText(), "false");

else

mySentence.put(ctx.getText(), temp);

}

}

//化简-> 与 <->语句中的可化简部分

public void exitTcexp(caiParser.TcexpContext ctx) {

String s = ctx.getText();

String str = null;

caiParser.OrexpContext Orexp = null;

String temp = "";

int i;

String ss = ctx.getText();

String orestrPre = null;

//处理多->或<->情况

for (i = 0; i < ctx.orexp().size(); i++) {

Orexp = ctx.orexp(i);

String orestr = Orexp.getText();

// 替换以前已经做过的子式，并删除该子式

Set<String> mySet = mySentence.keySet();

List<String> nodes = new ArrayList<String>();

for (String key : mySet) {

String Keychange = replace(key);

String Str1 = orestr.replaceAll(Keychange, mySentence.get(key));

if (!Str1.equals(orestr))

nodes.add(key);

orestr = Str1;

}

for (String node : nodes)

mySentence.remove(node);

//代入新式子中出现的已知真值的谓词常量

mySet = mainAntlr.preValue.keySet();

for (String key : mySet) {

String KeychangeT = replace(key);

String KeychangeF = '~' + KeychangeT;

String Str1 = orestr.replaceAll(KeychangeF, !mainAntlr.preValue

.get(key)+"");

Str1 = Str1.replaceAll(KeychangeT, mainAntlr.preValue.get(key)

.toString());

orestr = Str1;

}

//具体化简过程,通过具体推导，只存在以下几种情况，实现即可，只化简为OR语句，在可满足判断是重新建树的时候判断化简即可

if (i != 0) {

// String

caiParser.KeyContext Key = ctx.key(i - 1);

String keyStr = Key.getText();

if (orestrPre.equals("true")) {

temp = orestr;

} else if (orestrPre.equals("flase")) {

if (keyStr.equals("->")) {

temp = "true";

} else {

temp = "~" + orestr;

}

} else if (orestr.equals("false"))

temp = "~" + orestrPre;

else if (orestr.equals("true")) {

if (keyStr.equals("->")) {

temp = "true";

} else {

temp = "~" + orestrPre;

}

} else {

if (keyStr.equals("->")) {

temp = "~" + orestrPre + "||" + orestr;

} else {

temp = "(~" + orestrPre + "||" + orestr + ")&&("

+ orestrPre + "||~" + orestr + ")";

}

}

} else

temp = orestr;

orestrPre = orestr;// A

}

if (i != 0)

mySentence.put(ctx.getText(), temp);

}

//判断当前语句是否包含最外层括号，是则在父节点的括号可去掉

private boolean moveParenthesis(String str) {

Stack<Integer> mystack = new Stack<Integer>();

StringBuilder sb = new StringBuilder(str);

String finalStr = "";

char temp = 0;

Integer index = 0;

for (int i = 0; i < str.length(); i++) {

temp = sb.charAt(i);

if (temp == '(') {

mystack.push(i);

}

if (temp == ')') {

index = mystack.pop();

}

}

if (index == 0)

return true;

else

return false;

}

// ((@p(1)->((@q(1)&&1!=1)||(@q(2)&&1!=2)))&&(@p(2)->((@q(1)&&2!=1)||(@q(2)&&2!=2))));

// ((~@p(1)||@q(2))&&(~@p(2)||@q(1)));

//化简多余括弧

@Override

public void exitExp(caiParser.ExpContext ctx) {

super.exitExp(ctx);

if (ctx.lk() == null)

return;

String Comstr = ctx.sentence().getText();

// 替换以前已经做过的子式，并删除该子式

Set<String> mySet = mySentence.keySet();

List<String> nodes = new ArrayList<String>();

for (String key : mySet) {

String Keychange = replace(key);

String Str1 = Comstr.replaceAll(Keychange, mySentence.get(key));

if (!Str1.equals(Comstr))

nodes.add(key);

Comstr = Str1;

}

for (String node : nodes)

mySentence.remove(node);

//代入新式子中出现的已知真值的谓词常量

mySet = mainAntlr.preValue.keySet();

for (String key : mySet) {

String KeychangeT = replace(key);

String KeychangeF = '~' + KeychangeT;

String Str1 = Comstr.replaceAll(KeychangeF,

mainAntlr.preValue.get(key).toString());

Str1 = Str1.replaceAll(KeychangeT, mainAntlr.preValue.get(key)

.toString());

Comstr = Str1;

}

//利用正则表达式找出出现的谓词语句

Pattern pat = Pattern

.compile("~?@[a-z]+\\(((\\d\\d\*)|([a-z](\\d|[a-z])\*))\\)");

Matcher mat = pat.matcher(Comstr);

int flag = 0;

String mysubStr = "";

while (mat.find()) {

flag++;

mysubStr = mat.group();

}

//判断是否为单谓词语句，是则可去括号

if ((flag == 1 && Comstr.equals(mysubStr))) {

if (ctx.not() == null)

mySentence.put(ctx.getText(), Comstr);

else

mySentence.put(ctx.getText(), '~' + Comstr);

return;

}

//判断是否为false语句，是则可去括号

if (Comstr.equals("false")) {

if (ctx.not() == null)

mySentence.put(ctx.getText(), "false");

else

mySentence.put(ctx.getText(), "true");

return;

}

//判断是否为true语句，是则可去括号

if (Comstr.equals("true")) {

if (ctx.not() == null)

mySentence.put(ctx.getText(), "true");

else

mySentence.put(ctx.getText(), "false");

return;

}

//判断最外层是否的括号是否多余，是则去掉

String temp1 = Comstr.substring(0, 1);

String temp2 = Comstr.substring(Comstr.length() - 1, Comstr.length());

if (temp1.equals("(") && temp2.equals(")")) {

if (moveParenthesis(Comstr))

mySentence.put(ctx.getText(), Comstr);

else

mySentence.put(ctx.getText(), "(" + Comstr + ")");

} else

mySentence.put(ctx.getText(), "(" + Comstr + ")");

}

//化简数值比较结果，该数值比较只可为数字常量的比较

@Override

public void exitCompexp(caiParser.CompexpContext ctx) {

super.exitCompexp(ctx);

String str = null;

if (ctx.ne() != null) {

caiParser.ExpContext exp1 = ctx.exp(0);

caiParser.ExpContext exp2 = ctx.exp(1);

if (exp1.conexp() == null || exp2.conexp() == null || exp2 == null) {

return;

}

String e1 = exp1.getText();

String e2 = exp2.getText();

int E1 = Integer.parseInt(e1);

int E2 = Integer.parseInt(e2);

caiParser.NeContext ne = ctx.ne();

caiParser.AeContext ae = ctx.ae();

caiParser.BeContext be = ctx.be();

caiParser.EqContext eq = ctx.eq();

caiParser.BlContext bl = ctx.bl();

caiParser.AbContext ab = ctx.ab();

boolean result = false;

if (ne != null) {

result = (E1 != E2);

} else if (ab != null) {

result = (E1 > E2);

} else if (ae != null) {

result = (E1 >= E2);

} else if (be != null) {

result = (E1 <= E2);

} else if (eq != null) {

result = (E1 == E2);

} else if (bl != null) {

result = (E1 < E2);

}

if (result) {

mySentence.put(ctx.getText(), "true");

} else {

mySentence.put(ctx.getText(), "false");

}

}

}

}

在main函数中调用以下语句即可实现化简过程：

caiParser.ProgramContext root2 = parser2.program();

simplifyListenter simplifylistenter = **new** simplifyListenter();

walker.walk(simplifylistenter, root2);

System.***out***.println(*newStr*);

1. 判断可满足性

我们采用的是遍历所有可能赋值的方法来进行判断。

我们将已经化简得到的公式集，即化简后的公式集可满足，则原公式集可满足。利用正则表达式的方式，寻找出化简后的公式集的所有谓词常量语句，采用回溯法遍历其中变量的所有谓词常量语句，分别以true与false填入。将每种赋值情况代入，再利用之前化简阶段的那些语义函数得到化简结果，得到了在这种赋值情况下整个公式集是否满足的结果。如果遍历到一个可以满足，则输出相应变量的赋值，并结束该过程，否则持续这个过程，直到所有情况都不可，则不可满足。因化简类写的足够全面，故足以实现判断可满足与否的过程。

回溯函数：

//采用回溯法，拼凑出各种的谓词真值情况，并调用化简过程判断是否判断结果为真，是则返回该真值形式，且命题可满足，不然重复该过程,直到遍历所有结果仍不可为真，即公式集不可满足

public static boolean changeSet(String[] SubSet, int i) {

//回溯结束条件，调用新的建立新树，并判断是否可满足

if (i == max) {

caiParser parser = getContext(judgeStr);

caiParser.ProgramContext root = parser.program();

ParseTreeWalker walker = new ParseTreeWalker();

simplifyListenter simplifylistenter = new simplifyListenter();

walker.walk(simplifylistenter, root);

if (flag)

return true;

else

return false;

}

//回溯过程

preValue.put(SubSet[i], true);

boolean f = changeSet(SubSet, i + 1);

if (f) {

return true;

} else {

preValue.put(SubSet[i], false);

return changeSet(SubSet, i + 1);

}

}

//判断是否可满足

public static void Judge() {

//原化简结果即不可满足，则可成功退出

if (flag == false) {

System.out.println("命题集不可满足！！");

return;

}

//找到化简后的公式集中的所有谓词常量

Pattern pat = Pattern

.compile("@[a-z]+\\(((\\d\\d\*)|([a-z](\\d|[a-z])\*))\\)");

Matcher mat = pat.matcher(newStr);

int f = 0;

preValue = new HashMap<String, Boolean>();

while (mat.find()) {

f++;

preValue.put(mat.group(), true);

}

//转化为key值的数组，便于利用回溯法赋值

flag = true;

Set<String> mySet = preValue.keySet();

List<String> nodes = new ArrayList<String>();

String[] newsub = new String[mySet.size()];

max = mySet.size();

int i = 0;

for (String key : mySet) {

// String Keychange = replace(key);

newsub[i++] = key;

}

judgeStr = newStr;

//调用回溯函数

boolean F = changeSet(newsub, 0);

//判断结果，输出是否可满足

if (F) {

System.out.print("命题集可满足！在化简后的基础上，各项取值如下即可满足:\n");//

for (int j = 0; j < max; j++) {

System.out.print(newsub[j] + "==" + preValue.get(newsub[j])

+ " ");

}

System.out.println("\n");

} else {

System.out.println("命题集不可满足！！");

return;

}

}

在main函数中调用以下语句即可判断是否可满足：

judge();

1. 显示语法树

新建一个子线程，把输入的公式集的语法树显示出来，代码如下：

Thread myThread = **new** Thread(){

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

**super**.run();

caiParser parser1 = *getContext*(input) ;

parser1.setBuildParseTree(**true**);

Trees.*inspect*(parser1.program(), parser1);

}

};

myThread.start();

**运行测试测试结果：**

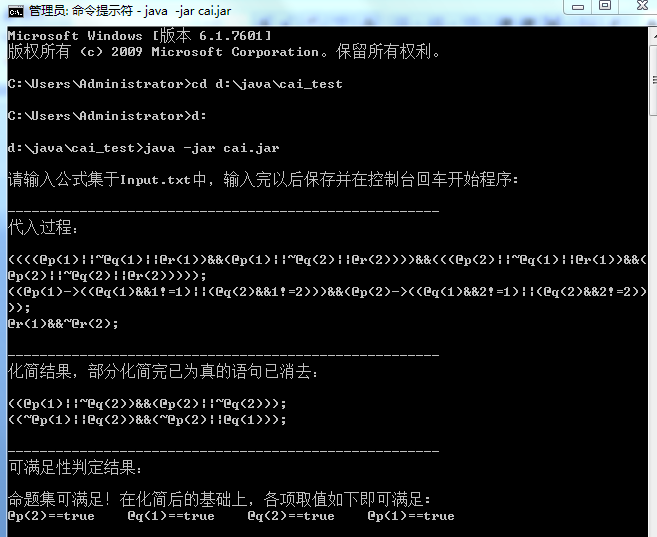
1、可满足测试集：

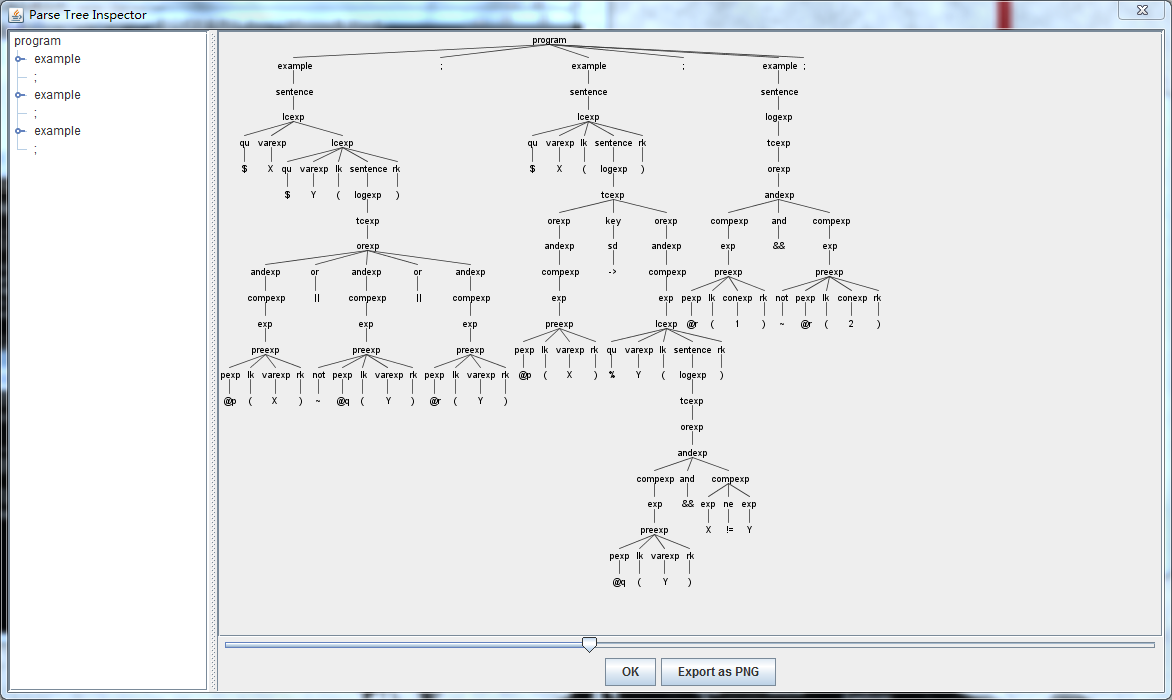
$X$Y(@p(X)||~@q(Y)||@r(Y));

$X(@p(X)-> %Y (@q(Y) && X != Y));

@r(1)&&~@r(2);

测试结果：





测试结果正确。

2、不可满足测试集：

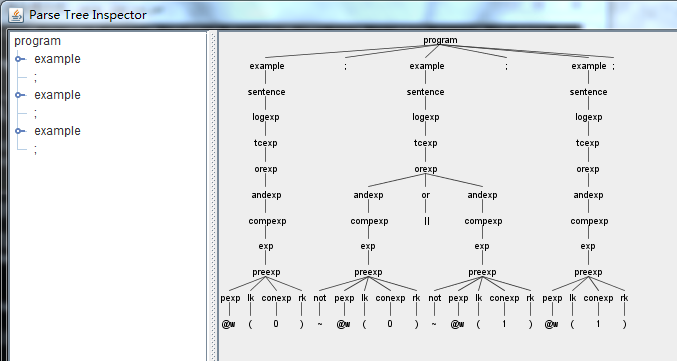
@w(0);

~@w(0)||~@w(1);

@w(1);

测试结果：

![C:\Users\Administrator\AppData\Roaming\Tencent\Users\525749092\QQ\WinTemp\RichOle\74IB7UD0`BS](_%8T75%C}I.png](data:image/png;base64,)



测试结果正确。