编译实践技术报告

1. 背景与现状分析

编译器是将一种语言（高级语言）翻译为另一种语言（低级语言）的程序。此次我们实现的编译器将会对一种类似于C的语言进行词法、语法分析，并将其结果可视化。

1. 相关技术介绍

我们的项目采用java web技术，用java进行后端编译阶段的的执行，前端用jsp，以html和css,js进行编译结果的展示。项目包含两部分，一是：编译部分（包括词法和语法），二是：展示部分（Symbol表，LL表，词法出错提示，语法出错提示，以及语法树的展示（借助外部工具））。词法主要为处理文件或文本输入，以DFA实现正则匹配进行处理。语法主要实现的是first算法,follow算法,以及分析表的构建。

1. 功能需求分析

1).词法:去除注释行，分析标识符，如不符合词法报错出具体位置，对输入返回token用于语法分析

2).语法:对输入的语句进行分析，如果遇到错误，则会提示错误所在的行数，以及错误的原因。输出语法分析表。

3）输入:以相对友好的方式进行源文件输入

4).输出显示:

a.词法:词法错误报错信息

b.语法:动态地显示在分析语句时，符号栈的变化，当前输入的token，以及输出的动作。

1. 功能设计

a).词法

各功能地实现如下:

1. .自动机DFA来实现正则表达式分析:

对于DFA的实现，采用的核心数据结构为HashMap<StateTrans,Integer>;

其中StateTrans为自定义类，其中包括两个属性，state[int],input[char]。该数据结构主要实现这样一种思想，对于当前状态，对于给定的某种输入，到达另一种状态。

构造DFA函数原型定义如下：

static int constructDfa(String regex,Map<StateTrans,Integer> dfa)

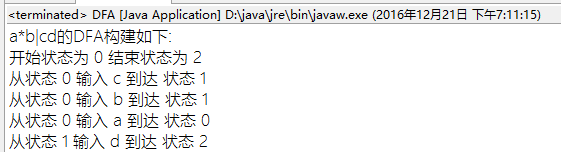
起始状态为0，该函数返回值为终止状态状态号。

判断是否符合某正则表达式的函数原型定义如下：

static boolean match(String str,String regex)；

该函数内部调用constructDfa,然后遍历str,从0状态开始做起始条件，若最终能够到达终止状态则说明匹配，若中间过程出现当前状态和当前输入在构造的hashMap中不存在的情况，则说明匹配错误，返回false;

该算法构造dfa的简单例子的过程输出如下



在整个编译项目中，对标识符，int,real及注释行判断是否匹配皆采用这种方式。

1. .扫描与分析设计:

项目中处理两种形式的输入，一是文件形式，二是直接输入文本；

对于文件输入，使用Scanner，以行为单位读取文件；对于文本输入，用换行符分割输入文本。也即是两者的共同部分是处理一行的内容。

对于行的处理如下：

若是注释行则不做处理

调用separate(String line)函数，该函数用于分割字符串，除了简单的依据空格进行分割，还处理了较为复杂的情况，比如**{if(i==1)** ,可以分割为 **{ if ( i == 1** ，这里采用了较为巧妙的解决方法，对于一个字符串str，倒序扫描str，直到i位置使得str 0到i所组成的字符串是标识符或关键字或num或括号等，然后str以i位置为分水岭一分为二，前半部分已满足要求，再对后半部分做如上相同的处理，如此递归处理则可完成。不过此时需处理一种特殊情况，如2i,如照此规则，则分割为2 和 i ,并没有词法错误，实际上这是一个错误的标识符，因此需对这类情况做特殊处理，即不做分割。

然后对每一个已经分割过的str进行处理，首先判断是否为关键字或括号或花括号或分号或逗号等，若是tokenName设置为其相应的名称，然后将设置行号等加到List<Token>中，该str处理完毕，进入下个str的处理；上述条件不符合的情况下判断是否是num（包括int和real），若是将tokenName设置为num,其它如上；上述条件仍不符合的情况下判断是否是标识符，若是将tokenName设置为id，其它如上；如上述条件都不满足，则该str有此法错误，记录错误信息用于报错。

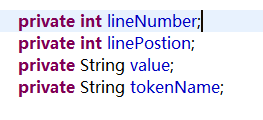
1. .词法返回值

词法分析核心函数原型（已处理文件输入为例）如下：

public List<Token> getTokenListFromFile(String fileName,List<LexError> errors)

其中errors为判断是否出错，调用该函数后若erros为空则说明无语法错误，否则进行出错处理

返回值为List<Token>,其中Token为自定义类，包含属性如下



b).语法

数据结构上，用HashMap<String,String>LLmap来保存该语言的产生式，用于生成语法分析树，其key为非终结符，其value为非终结符所对应的产生式。

在LLmap基础上用一个HashBasedMap<String,String,String>LLanaTable(HashBasedMap为guava拓展包内容，可以同时使用2个值作为key)构造该言的first,follow并保存其结果，结果即为语法分析时的分析。

在LLanaTable的基础上，我们构造了一个predicTable用于语法分析时的紧急错误恢复。

1).语法分析表存储；LLmap。

1. .分析堆栈构建与初始化:

Stack<String>strstack=**new** Stack<String>();

strstack.push("$");

strstack.push("program");//非终结符栈

对输入的语句逐个token的进行分析，分析时可能情况如下:

栈顶为终结符时:

Stack<String>strstack=**new** Stack<String>();

strstack.push("$");

strstack.push("program");//非终结符栈

对输入的语句逐个token的进行分析，分析时可能情况如下:

栈顶为终结符时:

**if**(!predicLLanaTable.containsRow(nonterminal))//栈顶有匹配的终结符时

(1).匹配，输出匹配动作，弹出栈顶无素，继续下一个token的分析。

**if**(nonterminal.equals(analysisElem))

{

matchaction(analysisElem);//匹配动作

analysisId++;//下一个token位置

analysisElem=analysisstr.get(analysisId);

strstack.pop();

}

(2).不匹配，弹出栈顶元素，并报错输入缺少该终结符。继续当前的token分析。

strstack.pop();

jumptopTerminal(nonterminal);//跳过栈顶终结符

i**nt** linepos=tokenlinepos.get(analysisId);

errors.add("第"+linepos+"行缺少"+nonterminal);

**continue**;

栈顶为非终结符时:

Map<String,String>terminalAndproduction=predicLLanaTable.row(nonterminal); //得到非终结符可以推导出的terminal和其对应的产生式。

Set<Entry<String,String>>tp=terminalAndproduction.entrySet();

Iterator<Entry<String,String>>tpit=tp.iterator();

HashMap<String,String>cur\_terminal\_pro=**new** HashMap<String,String>();

**while**(tpit.hasNext())//遍历当前的非终结符的产生式

(1).当前token匹配，弹出栈顶元素，输出匹配动作并将其对该元素所对应的产生式(不为”synch”,若该产生式为synch，则弹出栈顶元素，继续当前token分析)压入栈中。

**if**(!analysisElem.equals(terminal))...

else

{

String rightstr[]=**new** String[productionelem.length()];

rightstr=productionelem.split(" ");

**for**(**int** l=rightstr.length;l>0;l--)

{

**if**(rightstr[l-1].equals("ε"))**continue**;

**else** strstack.push(rightstr[l-1]);

}

outputaction(production);//输出动作

}

(2).当前token不匹配时，则处理情况又有如下两种:

1).此token不为合法的终结符，则跳过，并报错，输入错误，继续下一个token的分析。

**if**(!terminals.contains(analysisElem))

{

error(tokenlinepos.get(analysisId),analysisElem,errors);

analysisId++;

analysisElem=analysisstr.get(analysisId);

}

2).此token为合法的终结符时，则我们认为本次输入缺少一个终结符，且该终结符就在栈中，此时，从栈顶开始往下遍历，直到找到一个终结符。当找到终结符以后，就报错，输入缺少该终结符，并将该终结符加入到当前的输入中向下分析。(此时需要将栈的状态做一个备份，因为可能输入并不缺少)。当分析到栈为空时，如果此时输入还没跳过上次加入的字符的位置，则认为此输入是非法的，跳过这个非法的输入字符，并恢复栈的状态，继续进行下一个token的分析。

**for**(**int** x=strstack.size()-1;x>0;x--)

{

stackelem=strstack.get(x);

**if**(!predicLLanaTable.containsRow(stackelem)&&!stackelem.equals(analysisElem))

{ analysisstr.add(analysisId, stackelem);

**break**;

}

}

**if**(cur\_terminal\_pro.containsKey(stackelem)&&!cur\_terminal\_pro.get(stackelem).equals("synch"))//不能为synch也是为了保证栈顶元素不会一直跳过

{

**if**(!recover)

{//有可能在添加后一直跳不过去，此时就要恢复栈，并向前进来分析一个

jumpon.put(analysisElem, analysisId);//jumpon用来保存当前输入字符信息，以便以后恢复

oldstrstack=(Stack<String>) strstack.clone();

recover=**true**;

}//

............//当发现分析到栈为空时

**if**(strstack.empty())

{

**if**(recover)

{

**if**(jumpon.containsKey(analysisElem))

{

analysisId++;

analysisElem=analysisstr.get(analysisId);

strstack=(Stack<String>) oldstrstack.clone();

recover=**false**;

System.***out***.println("recover the stack");

errors.add("恢复栈状态，同时跳过当前字符!"+'\n');

**continue**;

}

**else** **break**;

}

**break**;

}

3).采用LL(1)算法的分析结构:LL(1)中最重要的就是first与follw算法

first与follow均采用迭代的方式实现。

first(String start,String production,String father)

如果一条语句为a->b | c ， 则start为a，production为b | c

father的作用体现在当b或c为非终结符时，first进入迭代，开始寻找此非终结的fisrt，此时用father记录这场迭代的起点表达式（a->b | c） 最终的结果会保存在LLanaMap中，比如此次迭代发现a的fisrt有c,d，LLanaMap的

a行c，d列便会储存（a->b | c）

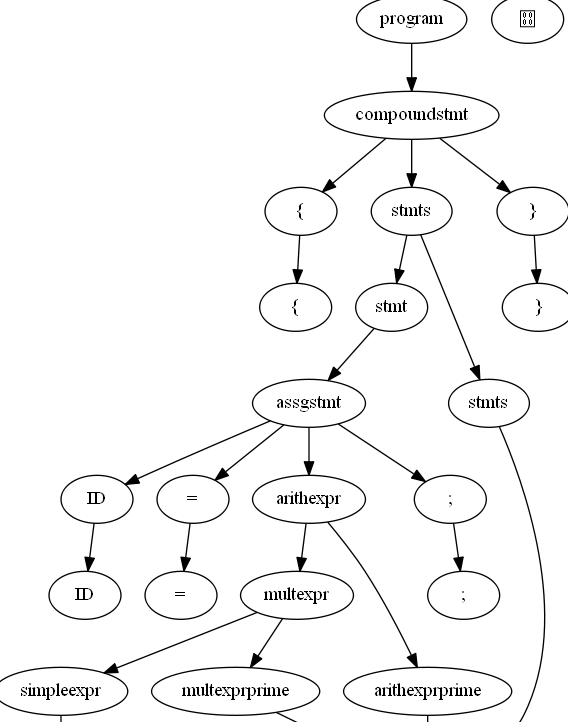
follow (String str,String mother)

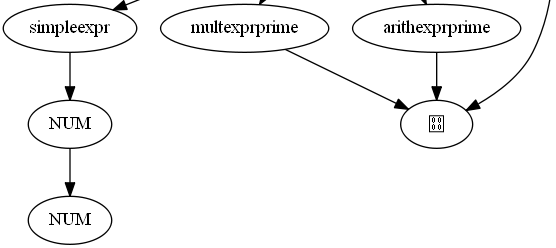
follow的迭代思路与first基本一致，values记录所有语句，但将-->改为“一个空格（a b | c），用来遍历查看str是否出现，以及记录每次迭代的源头（first的father不会改变，记录最底部的源头表达式，而在follow中mother实现这个功能，这次迭代最初的开端，father变为记录每次follow的左半部分的临时变量），在follow中因为后面为空时需要求最左边非终结符的follow，因此可能出现循环求follow的情况，mothers用于记录所有已经求过follow的非终结符，避免死循环。

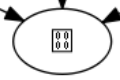
1. .输出语法树的构建:输出动作。

关于语法树的绘制，我们采用了GVEdit进行绘制。

对于输出语法树的构建，我们有一个单独的名为Graphic的类，这个类在接受了所有的输出语句后，首先在当前文件夹目录生成名为LLtree的文档，LLtree包含了根据输出语句写入的结点与线的信息，使用exec函数调用程序，传入LLtree的地址，便会生成一个语法树，如下图即为{int a=3;}该函数块的语法树:



接上图

这种标记是ε,该软件无法绘制，因此没有对ε进行区分（很显然例如ID->ID这根枝两个ID的node号不同），所以会指向一个单元

c).符号表

1).符号表的结构:

2).符号表的查询、添加、修改等操作:

e).出错处理:

错误我们把它分为两类：

1.第一类是输入错误，即类似于int a=b.其中b并不是一个Int型变量，只是一个特殊的符号，对于此类错误，我们采取的是直接跳过。

2.第二类是输入符号的缺失，如if(a>3 ,其中a是一个int型变量，可以看到if条件判断缺少了一个右括号，此时将会给予”第xxx行缺少xxxx”的错误提示。

简单地概括其处理情况即为:若分析时遇到的字符不为合法的终结符，则为第一种情况直接跳过;若分析时遇到的字符是合法的的终结符，若对此终结符的分析无法进行，那很有可能是因为缺少了一个终结符，而此终结符往往是在分析栈中，因此从栈中到该终结符，并将此终结符添加到原有的输入上分析，此时即会报错缺少.......，倘若分析到栈为空时，当前的输入还没有跳过原先无法分析的字，则视为该字符是错误输入，跳过并恢复栈状态，继续分析。

1).出错信息:

(1). **protected** **void** error(**int** i ,String str)

//第"+i+"行"+str+" 出错，跳过当前字符!!!!!!!!!!!!!

(2). **protected** **void** jumptopTerminal(String str)

//弹出栈顶终结符，并报错缺少相应的终结符

(3).errors.add("恢复栈状态，同时跳过当前字符!"+'\n');

//恢复栈状态，同时跳过当前字符

(4).errors.add("第"+linepos+"缺少"+stackelem+'\n');

//第xxxx行缺少xxxxx

2).出错定位

**int** linepos=tokenlinepos.get(analysisId);

Public List<String>analysis(List<String>analysisstr,

List<Integer>tokenlinepos)

将输入的字符与其所在的位置一一映射，同过tokenlinepos.get(x)，得到x位置的字符在输入中的位置。

5.功能实现

基本的输入、输出界面；

输入界面如下：



可以直接选择本地文件系统中文件或在编辑框中输入

提交之后看到分析结果如下



若有词法错误则不进行后续的语法分析

没有错误时如下：

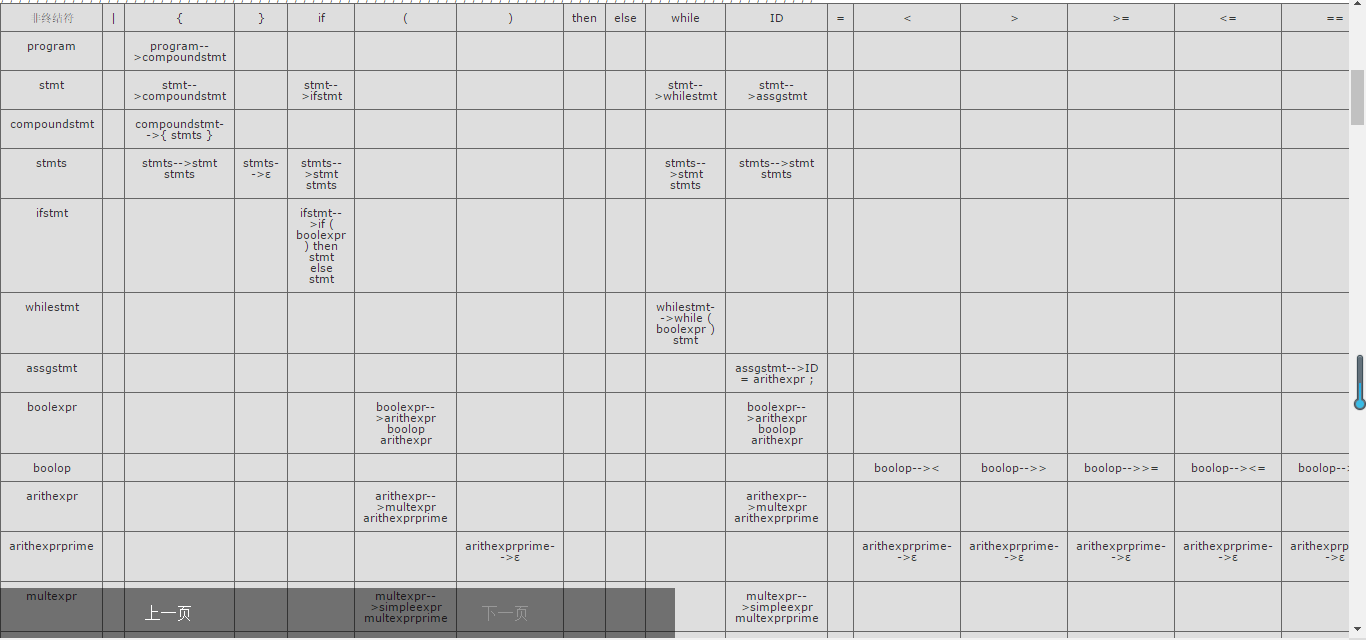


右边显示符号表

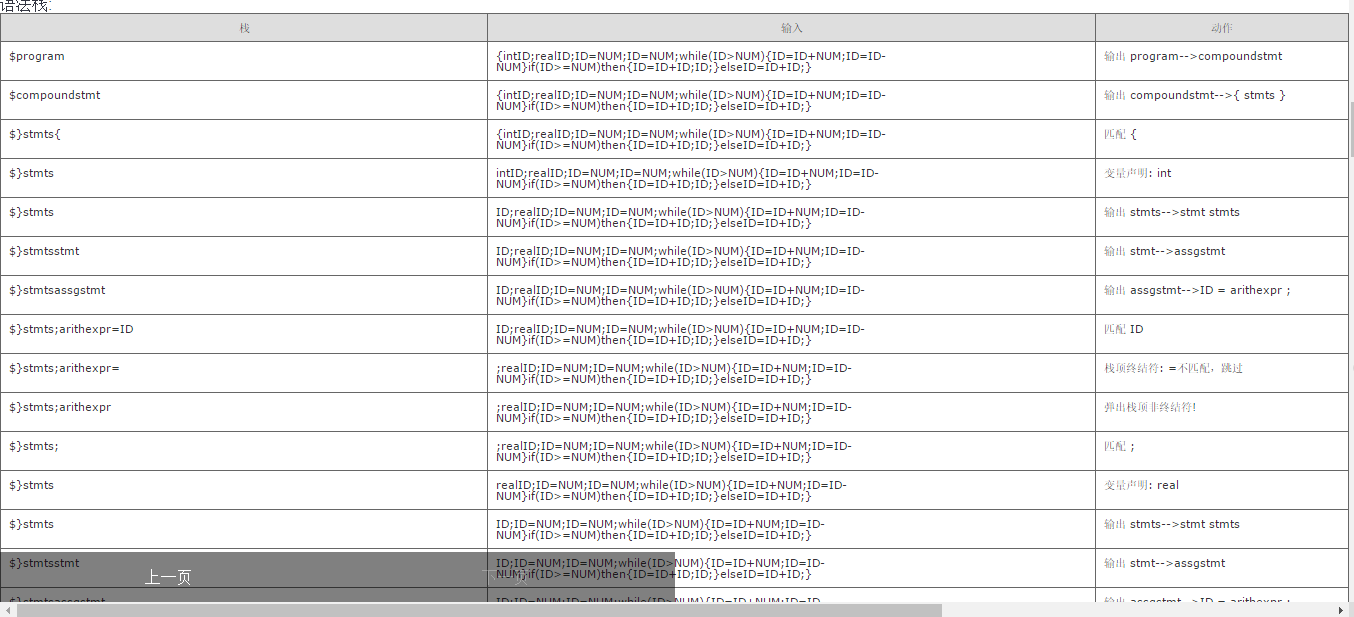
点击下一页查看语法分析情况



滚动往下可查看LL表和语法栈



（LL表，大小原因，只能截取到部分）



(语法栈 大小原因，只截取部分)

其他界面:

6.测试

自测用例

{ int a = 1 ; int b = 2 ; real c = 3.0 ;

{

a = a + 1;

b = b \* a ;

if ( a < b ) then c = c / 2 ; else c = c / 4 ;

}

}

{

}

检查用例

{

int i;

real j;

i=10;

j=100.00;

while (i>0) {n=n+1;i=i-1}

if (j>=50) then {sum=sum+i;n;} else sum=sum+j;

}

**7**.总结

我们的编译器除了对项目要求的语言进行实现外，还可以推广到其他的语言，即使该语言存在着左递归，该编译器也可以通过simplify()消除其直接左递归，从而进行相关的处理。对其他语言的处理只需要在LL的初始化中，将LLmap.put()改成需要的即可。

词法方面，自定义算法实现根据正则表达式构造DFA；文本分析阶段，对字符串分割算法的实现比较精细。

语法方面，first和follow的实现为迭代实现。

错误处理，此法阶段报告错误标识符（不符合标识符定义规范）的错误，语法阶段则是将一个token与其所在的行数绑定，以此进行相关的报错处理。

用户界面上，采用HTML+CSS+JavaScript的形式，较为友好

分工:

8.参考文献