



编译原理实验报告

|  |  |
| --- | --- |
| 专 业 班 级 | 计算机科学与技术18-3班 |
| 学生姓名及学号 | 2018211991 余梓俊 |
| 任 课 教 师 | 李宏芒老师 |
| 实验指导教师 | 李宏芒老师 |
| 实验地点 | 家中 |
| 2019 ~2020 学年第二学期 | |

## 实验一 词法分析设计

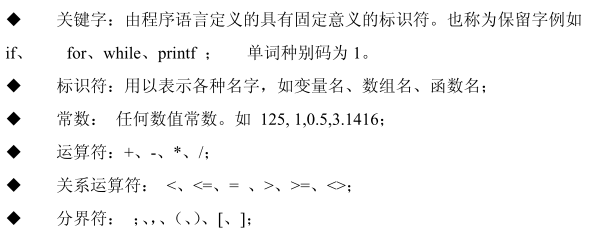
**一、实验目的**

通过本实验的编程实践，使学生了解词法分析的任务，掌握词法分析程序设计的原理和构造方法，使学生对编译的基本概念、原理和方法有完整的和清楚的理解，并能正确地、熟练地运用。

1. **功能描述**

根据输入的关键字和输入程序进行词法分析。

所能识别的单词种类与实验要求相同。



**三、程序结构描述**

程序最初由Java编写，由于实验二、三需要界面，决定使用JavaScript编写，用HTML来实现页面。在实现实验二、三后，发现实验一的输入和输出的形式同实验二、三类似，故将Java代码重写，迁移到了JavaScript代码。

以下仅介绍JavaScript版本的代码结构。

重要变量：

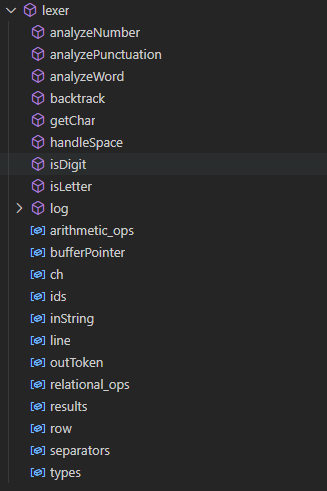
inString，bufferPointer，row，line，ch，outToken

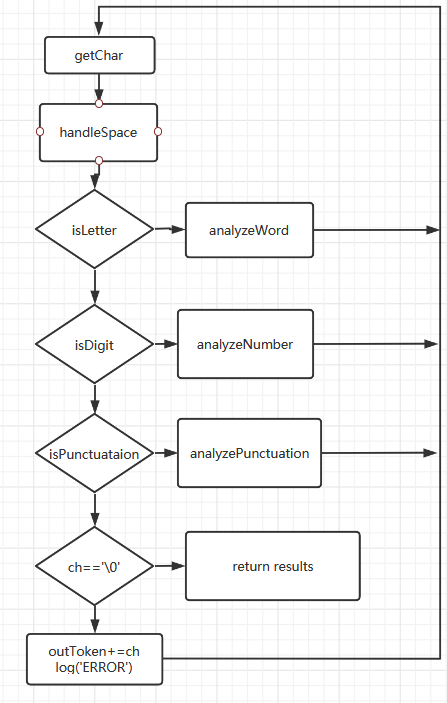
主要函数（主函数部分在流程图及算法描述）：

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称及调用格式返回类型 | 参数 | 返回值 | 函数功能 |
| getChar() |  |  | 读取一个字符，移动指针到下一个位置，line自增 |
| handleSpace() |  |  | 判断当前ch是否为空白类字符（空格、换行、制表），并进行处理 |
| analyzeWord() |  |  | 分析接下来的一个单词 |
| analyzeNumber() |  |  | 分析接下来的一个常数 |
| analyzePunctuation() |  |  | 分析接下来的一个标点符号 |
| backtrack() |  |  | 回退指针并根据需要调整row和line |

辅助函数：

|  |  |  |  |
| --- | --- | --- | --- |
| 函数名称及调用格式返回类型 | 参数 | 返回值 | 函数功能 |
| isLetter() |  | 布尔 | 字面功能 |
| isDigit() |  | 布尔 | 字面功能 |
| log() | type：输出的类型 |  | 根据参数type以及当前outToken和row、line等信息添加一条输出至结果数组，并将outToken置空 |





**四、详细的算法描述**

function getChar() {

    if (ch !== '\0') {

        ch = inString[bufferPointer ++];

        ++ line;

    }

}

function handleSpace() {

    while (ch === ' ' || ch === '\t' || ch === '\n' || ch === '\r') {

        if (ch === '\n') {

            ++ row;

            line = 1;

        }

        if (ch === '\r') {

            line = 1;

        }

        getChar();

    }

}

注意GetChar（）只是将下一个字符赋给ch后将指针后移，不会将ch添加到outToken中，而handleSpace（）则根据当前ch来判断是否需要处理空白符。

AnalyzeWord（），analyzeNumber（）和analyzePunctuation（）算法相似，以analyzeWord（）作为实例。

function analyzeWord() {

    while (isLetter() || isDigit()) {

        outToken += ch;

        getChar();

    }

    backtrack();

    if (keywords.has(outToken)) log('KEYWORD');

    else log('ID');

}

注意由于analyzeWord的只可能在isLetter为真后被调用，因此可以在第一次就直接循环判断isLetter()||isDigit()。

上文提到getChar并没有将ch加到outToken里，这里看到这一步操作是放在了各个analyze函数中，同时当结束循环时应当调用backTrack（）来回退一次指针。

function backtrack() {

    if (ch !== '\0') {

        -- bufferPointer;

    }

    /\* Put this expression out of the above if scope

    to keep the last outToken's position correct. \*/

    -- line;

}

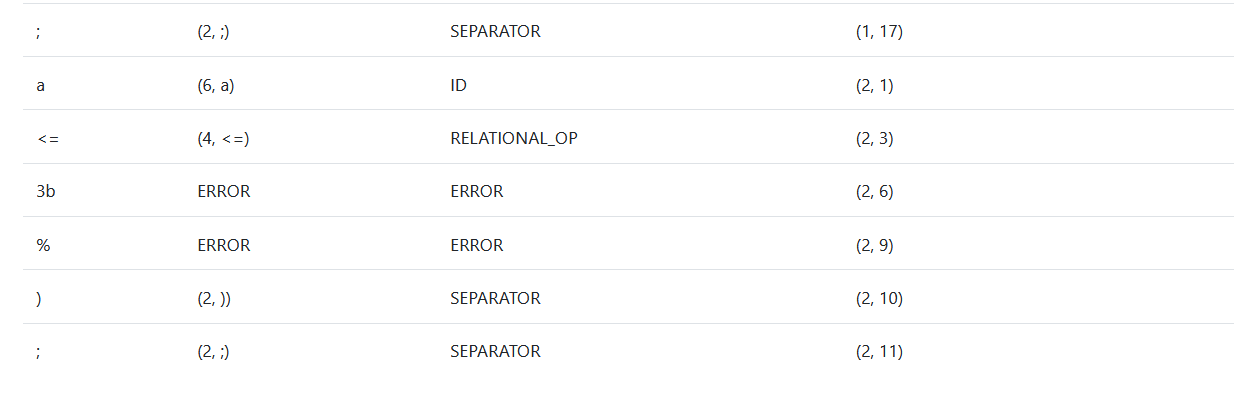
BackTrack（）函数将指针回退一位，但仅当ch！==‘\0’才回退，否则程序将无限循环。

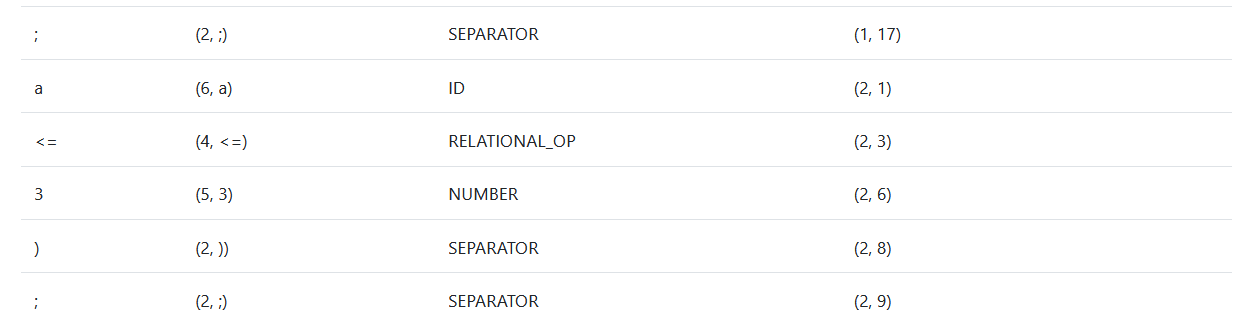
同时注意，无论ch！==‘\0’与否，都应将line减一，否则在输出程序最后一个单词的位置时line将比实际的大1，这是因为结束时line已经指向了’\0’的后面一位，用line – outToken.length计算便会导致大1。

**五、给出软件的测试方法和测试结果**

同样的，此处也只给出JavaScript版本的测试结果（Java版本的结果相同，区别仅为Java版本为控制台输出）。



特色功能：

若分析结果存在某个单词的识别结果为ERROR，分析按钮将变为红色；去掉输入程序中的错误后再点击分析，分析按钮将变为绿色。

不足之处：

对无符号整数的判断并不完美，常熟开头可以为0，如012将会被接受为NUMBER类型（算不上错误，但由于精力有限，没有增加额外的判断逻辑）。

**六、实验总结**

1. 由于在写完实验二、三后回过头来将实验一由Java迁移到了JavaScript，除去界面以外的核心代码由200行变为了130行，体会到了动态弱类型语言的方便。
2. 即使已经实现过了一遍，在迁移过程中仍然遇到了bug，相比于实验二三，实验一的难点不在于数据结构和算法架构的设计，而在于指针等精细的流程控制（例如backTrack（）将—line放在if外面等等）。

## 实验二 LL(1)分析法

1. **实验目的**

通过完成预测分析法的语法分析程序，了解预测分析法和递归子程序法的区

别和联系。使学生了解语法分析的功能，掌握语法分析程序设计的原理和构造方

法，训练学生掌握开发应用程序的基本方法。有利于提高学生的专业素质，为培

养适应社会多方面需要的能力。

1. **写出 LL（1）分析法的思想及写出符合 LL（1）分析法的文法**

对文法G的句子进行确定的自顶向下语法分析的充分必要条件是，G的任意两个具有相同左部的产生式A—>α|β 满足下列条件：

（1）如果α、β均不能推导出ε，则 FIRST(α) ∩ FIRST(β) = ∅。

（2）α 和 β 至多有一个能推导出 ε。

（3）如果 β \*═> ε，则 FIRST(α) ∩ FOLLOW(A) = ∅。

将满足上述条件的文法称为LL(1)文法。

**First构造：**

对每一文法符号X∈VT∪VN构造FIRST(X)，连续使用下面的规则，直至每个集合FIRST不再增大为止：

1. 若X∈VT，则FIRST(X)＝{X}。

2. 若X∈VN，且有产生式X→a…，则把a加入到FIRST(X)中；若X→ε也是一条产生式，则把ε也加到FIRST(X)中。

**Follow构造：**

对于文法G的每个非终结符A构造FOLLOW(A)的办法是，连续使用下面的规则，直至每个FOLLOW不再增大为止：

1. 对于文法的开始符号S，置＃于FOLLOW(S)中；

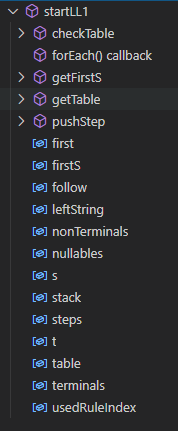
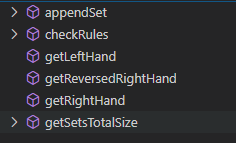
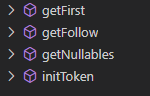
2. 若A→αBβ是一个产生式，则把FIRST(β)\{ε}加至FOLLOW(B)中；

3. 若A→αB是一个产生式，或A→αBβ是一个产生式而β→ε (即ε∈FIRST(β))，则把FOLLOW(A)加至FOLLOW(B)中。

1. **程序结构描述**

**变量常量与函数结构：**

common.js helper.js ll1.js



**具体解释：**

重要常量：

/\*\*

 \*  @param {Array<String>} rules

 \*  @param {String} inputString

 \*/

    const [nonTerminals, terminals] = initToken(rules);

    const nullables = getNullables(rules);

    const leftString = inputString.split('');

    const stack = [rules[0][0]];

const steps = [];

    const first = getFirst(rules, nonTerminals, terminals, nullables);

    const follow = getFollow(rules, nonTerminals, terminals, nullables, first);

    const firstS = getFirstS();

    const table = getTable();

common.js中定义的函数（实验二三均用到）

/\*\*

 \* @param {Array<String>} rules

 \* @returns {[Set<String>, Set<String>]} [nonTerminals, terminals]

 \*/

function initToken(rules) {

    let nonTerminals = new Set();

    let terminals = new Set();

    terminals.add('#');

。。。省略

函数功能：根据rules产生终结符和非终结符集合

    return [nonTerminals, terminals]

}

function getNullables(rules) {

    let nullables = new Set();

。。。省略

函数功能：根据rules产生可以推导出的epsilon的非终结符

以便于后续产生first、follow集合

    return nullables;

}

function getFirst(rules, nonTerminals, terminals, nullables) {

    const first = {};

。。。省略

产生各个非终结符的first集

    return first;

}

function getFollow(rules, nonTerminals, terminals, nullables, first) {

    const follow = {};

。。。省略

产生各个非终结符的follow集

    return follow;

}

helper.js辅助函数（实验二、三均用到，getReversedRightHand()仅实验二用到）

（函数参数、功能、返回值均可以直观地看出）

function getLeftHand(rule) {

    return rule.substring(0, rule.indexOf('->'));

}

function getRightHand(rule) {

    return rule.substring(rule.indexOf('->') + 2);

}

/\*\*

 \* @returns {Array} the reversed Array of the right hand of the rule

 \*/

function getReversedRightHand(rule) {

    return getRightHand(rule).split('').reverse();

}

function checkRules(rules) {

    return rules.findIndex(rule => !rule.includes('->')) !== -1 ?

        false : true;

}

function getSetsTotalSize(sets) {

    return Object.values(sets)

        .map(set => set.size)

        .reduce((prev, curr) => prev + curr);

}

function appendSet(setA, setB) {

    setB.forEach(item => {setA.add(item)});

}

主要函数（主函数部分在算法描述）：

function getFirstS() {

    const firstS = {};

参数：rules, nonTerminals, terminals, nullables, first, follow

功能：产生每条规则右部符号串的first集

返回：firstS（对象/字典）,属性为每条规则对应的序号，属性值为该条规则的first集

    return firstS;

}

function getTable() {

    const table = {};

参数：rules, nonTerminals, terminals, firstS

功能：产生LL1分析表

    return table;

}

function checkTable() {

功能：检查LL1分析表是否有冲突（上一步产生的分析表每个的格子是一个数组，如果数组长度大于一则说明由冲突），若有冲突，返回false，否则把每个表格有数组变为单个字符，然后返回true。

}

/\*\*

 \* @param {Array} stack

 \* @param {Array} leftString

 \* @param {number} usedRuleIndex -1 ~ rules.length - 1

 \* -1 means no rule is used

 \* @param {String} action if usedRuleIndex !== -1,

 \* action should be 'pop, push(reverse of the right hand)'

 \* @returns {Object} return an Object that contain 4 strings

 \*/

function pushStep(stack, leftString, usedRuleIndex, action) {

。。。省略

    steps.push({

        。。。省略

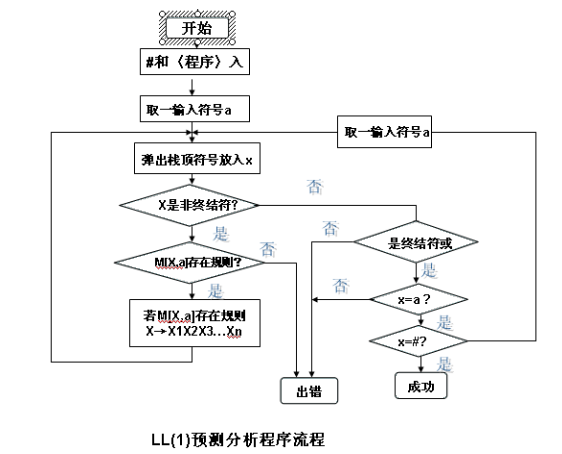
});

功能：产生当前动作的详细信息，加到steps数组中，steps数组最终会用来在界面上输出分析过程。

}

1. **详细的算法描述**

主程序的算法：



主程序的算法代码：

。。。部分常量初始化在前面已提到，剩余的初始化（如调用检查分析表冲突）省略

pushStep(stack, leftString, -1, 'initialize');

while (stack.length) {

    const t = stack.pop();

    if (nonTerminals.has(t)) {

        const usedRuleIndex = table[t][leftString[0]];

        if (usedRuleIndex === null || usedRuleIndex === undefined)

            return ['fail', steps];

        getReversedRightHand(rules[usedRuleIndex]).forEach(token => {

            stack.push(token);

        });

        pushStep(stack, leftString, usedRuleIndex);

    }

    else {

        const s = leftString.shift();

        if (t !== s) return ['fail', steps];

        pushStep(stack, leftString, -1, 'get next');

    }

}

return [undefined, steps];

产生nullables集合，first集，follow集的算法：

采用不动点算法，在一轮循环开始时和结束前，分别计算集合的size，若size不再变化，则退出循环，否则就一次次的循环遍历每条规则进行集合的计算。

1. **给出软件的测试方法和测试结果。**

准备了以下两个文法：

默认用例：

E->TG

G->+TG

G->-TG

G->

T->FS

S->\*FS

S->/FS

S->

F->(E)

F->i

测试用例：

Z->d

Z->XYZ

Y->c

Y->

X->Y

X->a

其中默认用例是LL(1)文法，而测试用例的LL(1)分析表存在冲突，不是LL(1)文法

**阶段测试1**：终结符集（terminals），非终结符集（nonTerminals），可空集（nullables）

默认用例结果：



测试用例结果：



结果均正确

**阶段测试2**：FISRT集，FOLLOW集，FIRST\_S集，LL(1)分析表

默认用例结果：（分析表较长，只展开第一行）



测试用例结果：



结果均正确（两个文法的LL1分析表结果均与手算结果相同，可以看到文法1分析表不存在冲突，是LL1文法，而文法2分析表存在冲突，不是LL1文法）

**最终测试：**

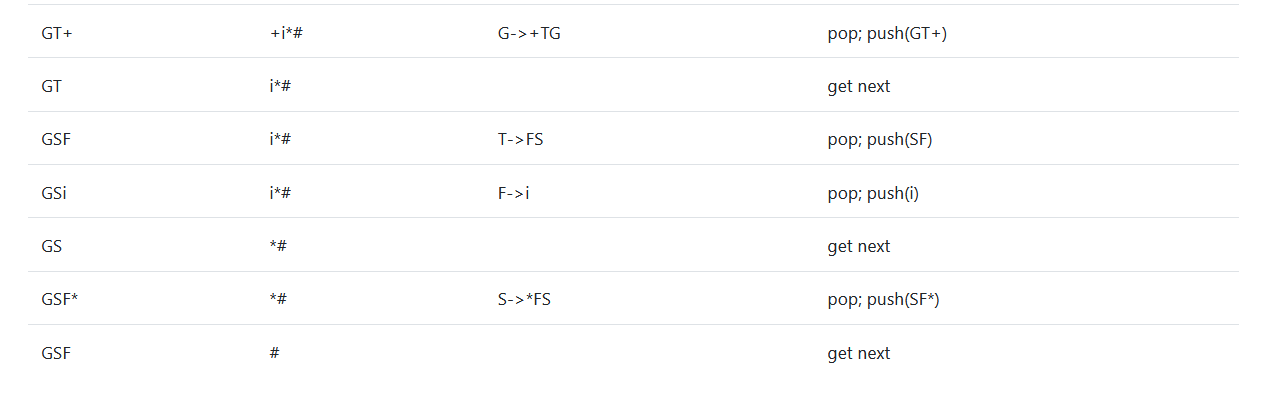
1. 输入一个错误的格式，可以看到如下提示



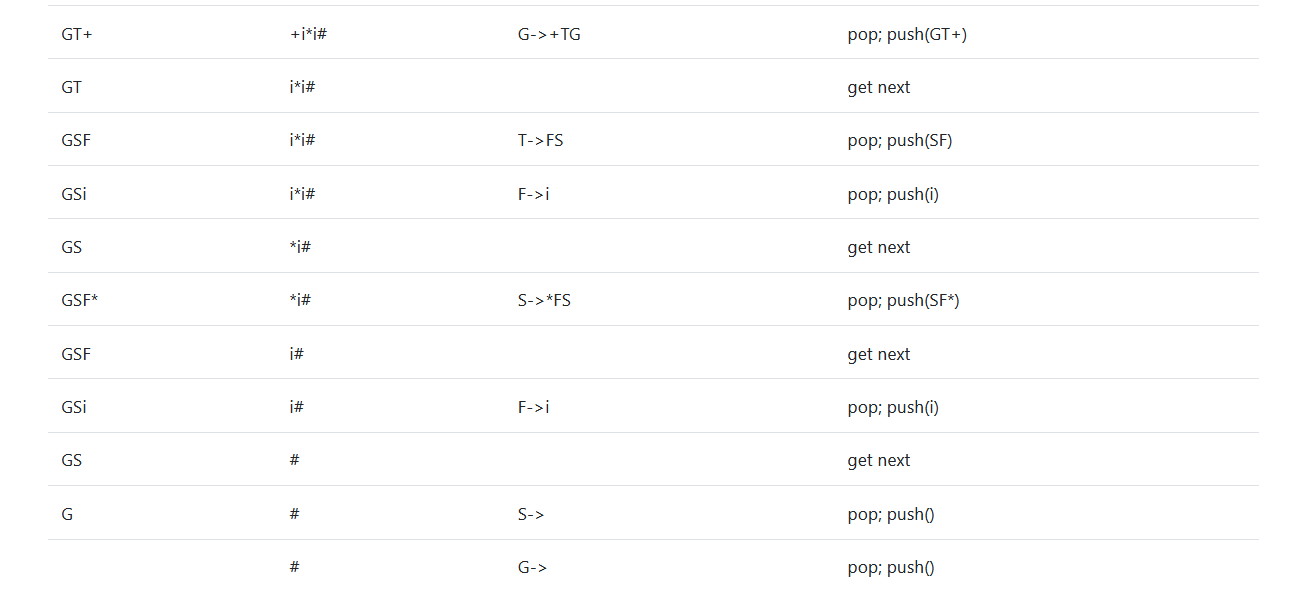
1. 输入非LL1文法，即测试用例，可以看到如下提示



1. 输入LL1文法规则，但输入串不是该文法的句子，可以看到分析过程将显示出错前的所有步骤，并且按钮将由蓝色/绿色变为红色

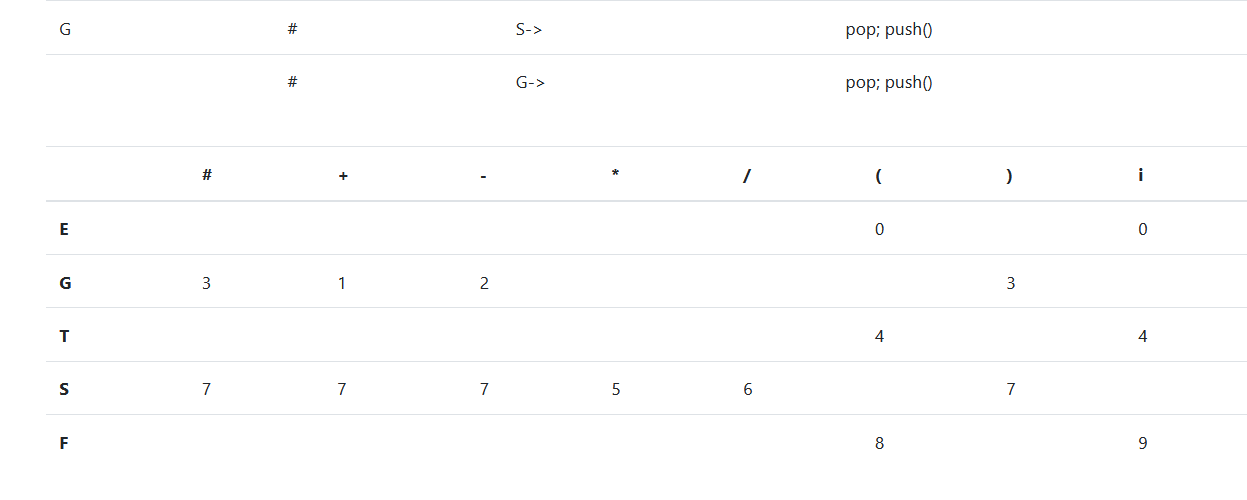
 

1. 输入LL1文法规则，并且输入是该文发的句子，可以看到分析过程将显示所有步骤，并且按钮将由蓝色/红色变为绿色

1. **实验总结**
2. 起初没有注意到，JavaScript的强制转换函数Number()如果会把null转换为0，导致当栈顶为非终结符且查分析表应报错时没有报错，而是用第0条规则进行pop、push，进而导致当输入不可被接受的串时分析栈最终会无线增长。
3. 与实验一相比，虽然语法分析器的复杂度上升了不少，小的模块比较多，但是只要理解了产生分析表的过程，包括first、follow集的产生方法，其流程以及算法的细节并不是太难实现。
4. **验收后改进**

运行界面添加展示LL1分析表：



注：考虑到表格中放推导串过于拥挤，此处用数字来指代具体的规则，0即为输入的第一条规则，1即为第二条规则，以此类推。

## 实验三 LR(1)分析法

1. **实验目的**

构造 LR(1)分析程序，利用它进行语法分析，判断给出的符号串是否为该文法识别的句子，了解 LR（K）分析方法是严格的从左向右扫描，和自底向上的语法分析方法。

1. **描述 LR(1)语法分析程序的设计思想**

**1.提取所有有效识别活前缀的式子：**

形式上我们说一个LR(1)项目[A→α·β, a]对于活前缀γ是有效的，如果存在规范推导其中，

1) γ＝δα；

2) a是ω的 第一个符号，或者a为#而ω为ε。

[A→α·Bβ, a]对活前缀γ＝δα是有效的，则对于每个形如B→ξ的产生式， 对任何b∈FIRST(βa)，[B→·ξ, b]对γ也是有效的。

**2.项目集I 的闭包CLOSURE(I)构造：**

1. I的任何项目都属于CLOSURE(I)。
2. 若项目[A→α·Bβ, a]属于CLOSURE(I)，B→ξ 是一个产生式，那么，对于FIRST(βa) 中的每个终结符b，如果[B→·ξ, b]原来不在CLOSURE(I)中，则把它加进去。
3. 重复执行步骤2，直至CLOSURE(I)不再增大为止。

**3.GO构造：**

令I是一个项目集，X是一个文法符号，函数GO(I，X)定义为：GO(I，X)＝CLOSURE(J)其中J＝{任何形如[A→αX·β, a]的项目| [A→α·Xβ, a]∈I}

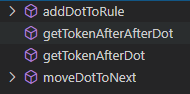
**4.分析表构造：**

令每个Ik的下标k为分析表的状态，令含有[S′→·S, #]的Ik的k为分析器的初态。

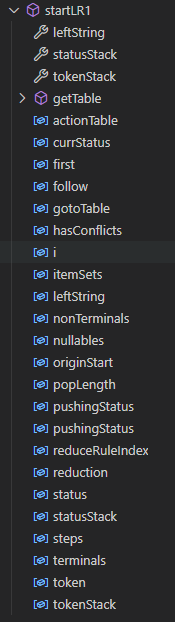
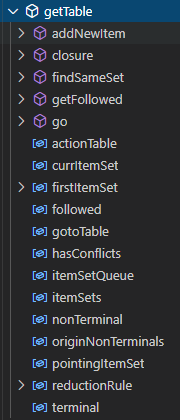
1. 若项目[A→α·aβ, b]属于Ik且GO(Ik, a)＝Ij， a为终结符，则置ACTION[k, a]为 “sj”。
2. 若项目[A→α·，a]属于Ik，则置ACTION[k, a]为 “rj”；其中假定A→α为文法G′的第j个产生式。
3. 若项目[S′→S·, #]属于Ik，则置ACTION[k, #]为 “acc”。
4. 若GO(Ik，A)＝Ij，则置GOTO[k, A]=j。
5. 分析表中凡不能用规则1至4填入信息的空白栏均填上“出错标志”。
6. **程序结构描述：**

**变量常量与函数结构：(common.js helper.js 见实验二)**

bottom-up-helper.js（自底向上分析专用的辅助函数，主要用来操作“点”符号）



lr1.js getTable函数展开

**具体解释：**

注：程序使用`符号来表示点符号。

重要常量：

/\*\*

 \*  @param {Array<String>} rules

 \*  @param {String} inputString

 \*/

const originStart = rules[0][0];

rules.unshift(`${originStart}'->${originStart}`);

const [nonTerminals, terminals] = initToken(rules);

const nullables = getNullables(rules);

const leftString = inputString.split('');

const statusStack = ['0'];

const tokenStack = ['#'];

const steps = [];

const first = getFirst(rules, nonTerminals, terminals, nullables);

const follow = getFollow(rules, nonTerminals, terminals, nullables, first);

const [actionTable, gotoTable, itemSets, hasConflicts] = getTable();

if (hasConflicts)

    return ['notLR1', steps];

项目集的数据结构：

        /\* itemSet format:

            {

                id: '0',

                items (Set): {

                    {

                        string: 'A->B`C',

                        followed: Set{'#'}

                    },

                    {

                        string: 'C->`a',

                        followed: Set{'#', 'b'}

                    },

                    ...

                }

            }

        \*/

终结符、非终结符、first、follow集计算方法在实验二提到的common.js中

辅助函数也均在实验二提到的helper.js中

主要函数（主函数部分在算法描述）：

function getTable() {

功能：根据rules，terminals，nonTerminals，first，follow产生项目集族与LR1分析，以及表中是否有冲突

return [actionTable, gotoTable, itemSets, hasConflicts];

}

以下函数均为getTable() 函数的内部子函数

function go(itemSet, token) {

参数：itemSet：原来的项目集；token：一个字符

功能：即课上所述的go函数，计算从原有项目集按某个字符引出的新项目集

返回：引出的新项目集（如已存在则返回已存在的项目集的引用），若为空则返回null，项目集的数据结构如前文所述。

}

/\*\*

 \* @param {Set} items Set of the defined format items.

 \* @returns {Set}

 \*/

function closure(items) {

参数：items，项目的集合（不是上面定义的项目集，是项目集中的items属性）

功能：计算items的闭包

返回：闭包（go函数调用该函数拿到闭包后就会查找是否已经存在一样的项目集）

}

function getFollowed(tokenAfterAfterDot, originFollowed) {



功能：在计算闭包的过程中被调用，即计算上图中的 b

返回：上图中的 b（一些终结符的集合）

}

/\*\*

 \* @param {String} string

 \* @param {Set} followed

 \* @param {Set} items

 \*/

function addNewItem(string, followed, items) {

    功能：添加一条项目到某个项目集（因为需要判断该项目是否已经存在，需要一定的判断逻辑，因此将该动作单独写成了一个函数）

}

/\*\*

 \* @param {Set} items

 \* @returns {number} itemSet id

 \*/

function findSameSet(items) {

    功能：根据项目的集合在项目集中查找是否已经存在有相同项目集合的项目集，返回项目集的id或返回-1，被调用情况前文已述

}

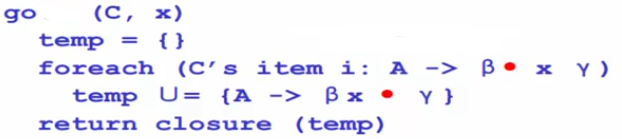
1. **详细的算法描述**

**Closure()函数算法：**

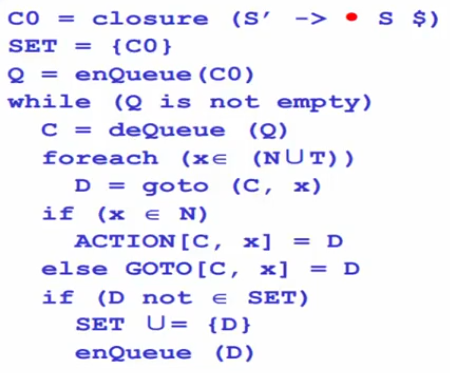
使用前面实验所述的不动点算法，在每次循环开始时和结束前计算项目的集合的size，不再变化则退出循环。

循环内部的步骤：遍历所有项目，对项目，添加到项目集（其中b由getFollowed函数计算）

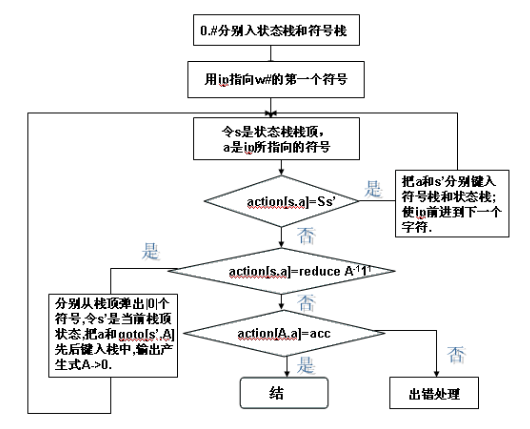
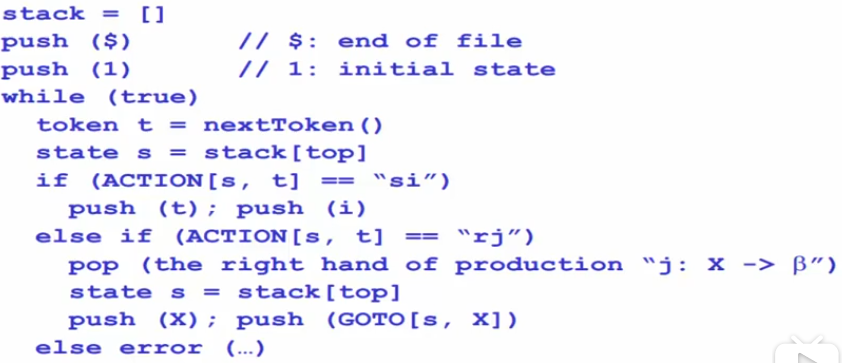
**Go()函数算法：**



**利用closure()和go()，getTable()函数的大致算法：**



**主函数大致算法（即根据分析表进行分析的过程）：**



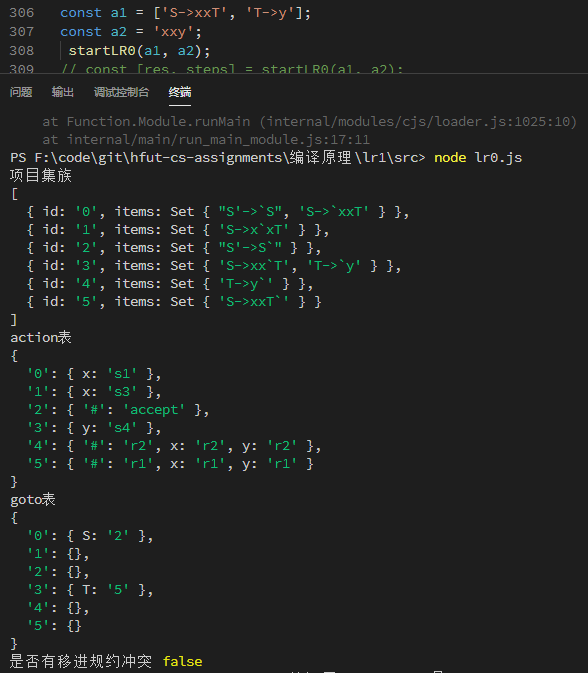
1. **给出软件的测试方法和测试结果。**

**先实现LR(0)：**

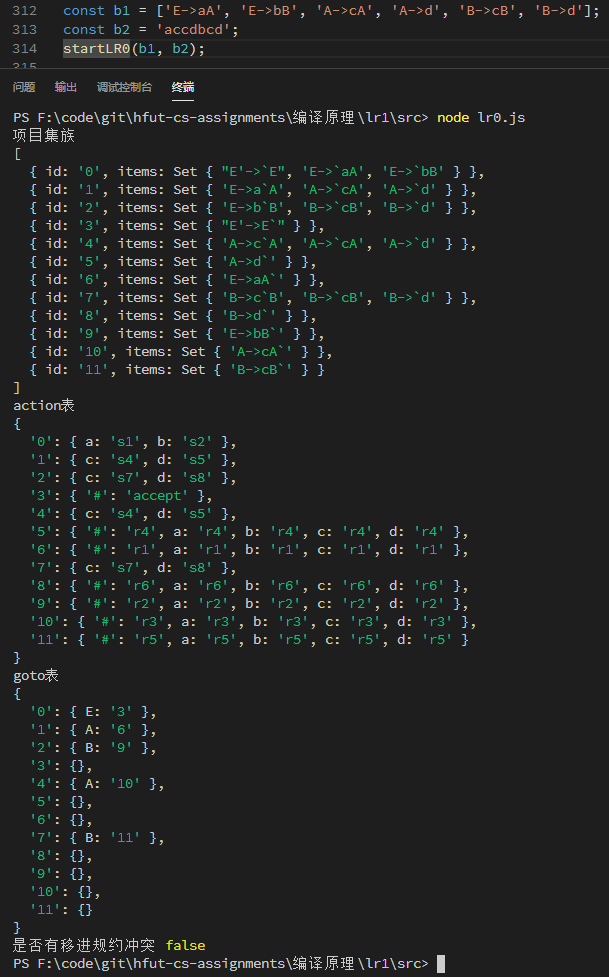
**LR(0)阶段测试1**：项目集族（itemSets），action表（actionTable），goto表（gotoTable）

(注：因代码修改而未重新截图，下图中“是否有移进规约冲突”应为“是否有冲突”)

测试用例1：



测试用例2：



两例结果均正确

**LR(0)阶段测试2**：移进规约过程

测试用例1：



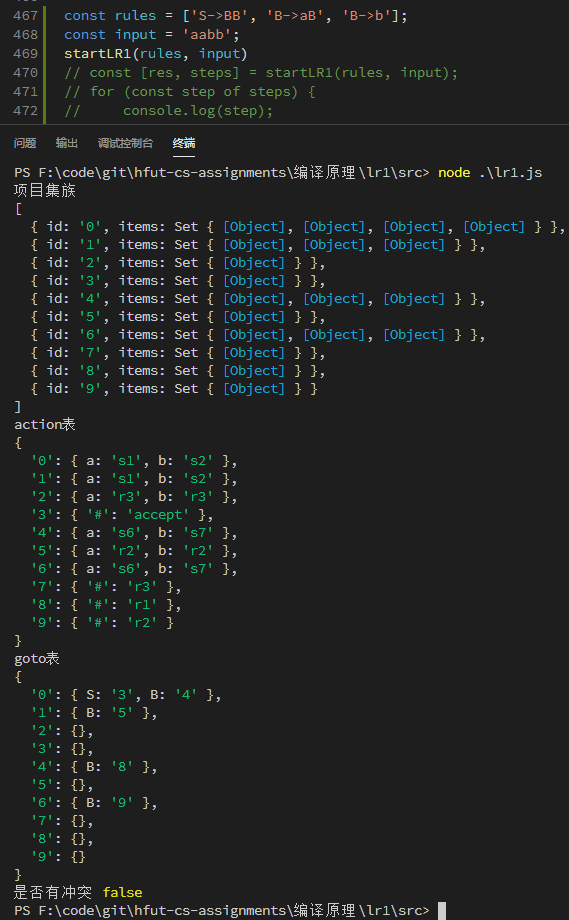
测试用例2：



**在LR(0)基础上实现LR(1)：**

**LR(1)阶段测试**：项目集族（itemSets），action表（actionTable），goto表（gotoTable）





**LR(1)最终测试：**

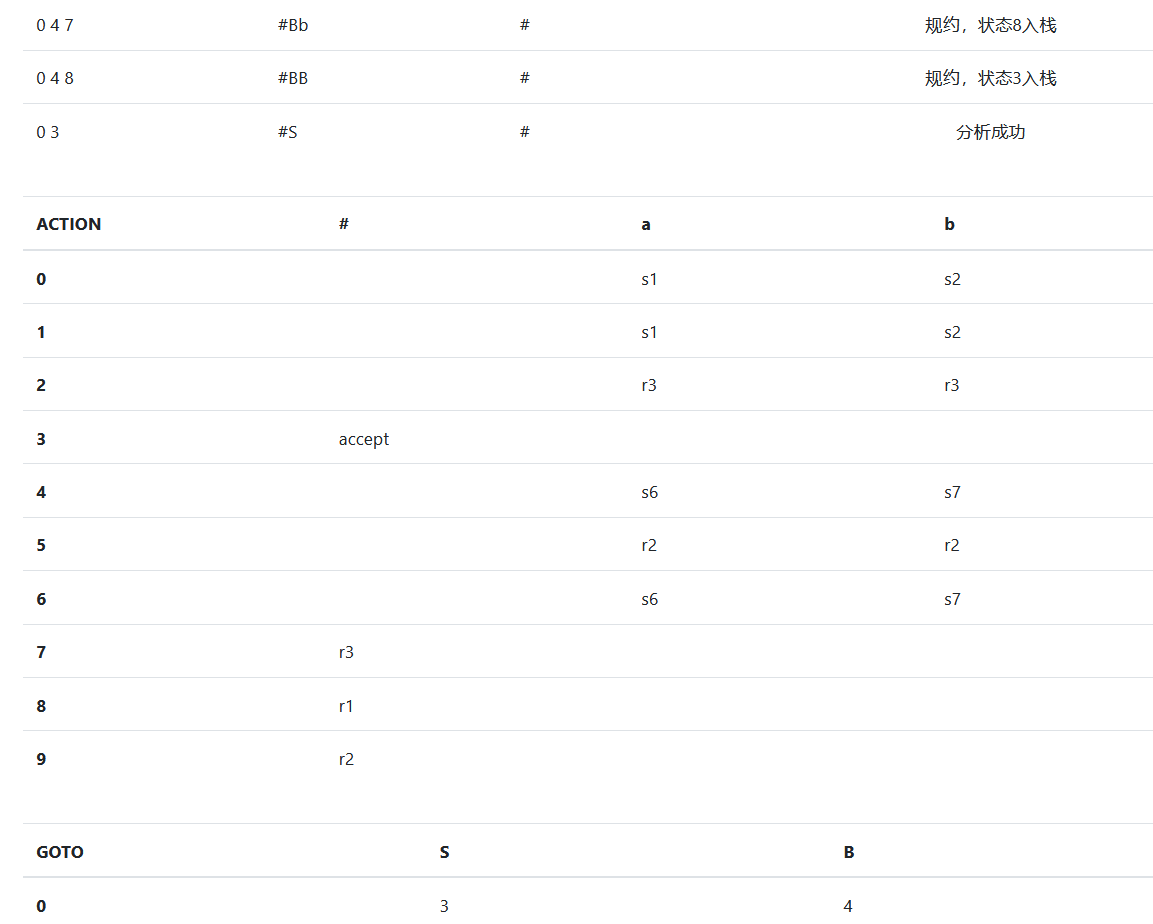
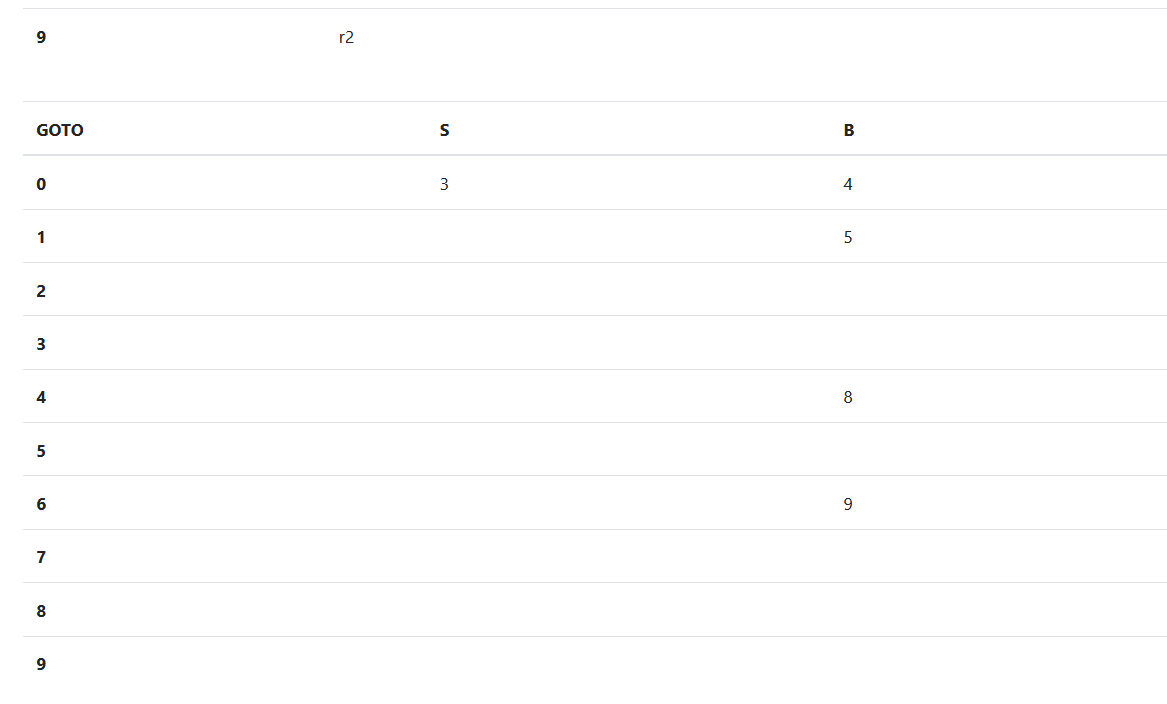




格式错误、非LR1文法等等提示的效果均和实验二相同，不再截图。

1. **实验总结**
2. LR1语法分析器是三个实验中最复杂的一个，但还好部分代码可以从实验二复用。
3. 好的数据结构设计、好的函数设计非常重要。鉴于LR1的复杂，实现过程中个人选择了先实现LR0分析器，再在LR0基础上实现LR1。
4. 在实现LR0的过程中，得益于函数的拆分，一些小的操作可以直接从实验二设计的helper.js中的函数。而得益于数据结构的设计，在LR0向LR1的转变中，除了项目中加入了followed终结符集合，其余数据结构几乎相同，在LR0基础上完成LR1仅花费了不到四个小时。
5. 自底向上的分析方法相比实验二的自顶向下复杂了不少，在设计项目集族的数据结构和具体的closure、go、getTable实现上花费了不少功夫。
6. 得益于JavaScript的动态弱类型，对设计的数据结构的实现和算法的实现的迭代速度非常地快，但动态弱类型也对代码编写时的严谨程度提出了更高的要求，在debug时遇到的类型问题也令人头疼。
7. 语法分析器的各个模块的思想都十分巧妙，即使实现了之后，在撰写报告的过程中仍然会停“笔”对代码结构和算法感到犹豫、进行思考，不得不感叹编译原理这些研究成果的思想的精妙之处。
8. **验收后改进**

运行界面添加展示LR1的action表和goto表：

代码已托管：

<https://github.com/wine99/hfut-cs-assignments>