27   器 winter 201	(小实验) 理解编译原理:一个四则运算的解释 9-03-21
的知识,如这里设计了	(00:00)  讲述: winter 大小: 11.90M  起中, 我在 JavaScript 和 CSS 的部分,多次提到了编译原理相关的知识。这一部分果我们从编译原理"龙书"等正规的资料中学习,就会耗费掉不少的时间,所以我在一个小实验,帮助你快速理解编译原理相关的知识。  比较特殊,我们来做一段详细的代码实验,详细的代码我放在了文章里,如果你正在可以点击文章查看详情。
<b>分析</b> 按照编译原 • 定义四则 • 词法分析 • 语法分析 • 解释执行	理相关的知识,我们来设计一下工作,这里我们分成几个步骤。 运算:产出四则运算的词法定义和语法定义。 : 把输入的字符串流变成 token。 : 把 token 变成抽象语法树 AST。 : 后序遍历 AST,执行得出结果。
1 <b>1 + 2</b> 2 首先我们来	<b>自</b> 复制代码
<ul><li>Opera</li><li>Whitesp</li><li>LineTern</li><li>这里我们对</li></ul>	minator: <lf> <cr>空白和换行符没有任何的处理,所以词法分析阶段会直接丢弃。</cr></lf>
JavaScript	来定义语法,语法定义多数采用 BNF,但是其实大家写起来都是乱写的,比如标准里面就是一种跟 BNF 类似的自创语法。 义的核心思想不会变、都是八种结构的组合产生一个新的结构,所以语法定义也叫语像有优光级,所以我们可以认为加法是由若干个乘法再由加号或者减号连接成的:
2	ession> ::= AdditiveExpression> <eof>  tiveExpression&gt; ::= MultiplicativeExpression&gt; <additiveexpression>&lt;-&gt;<multiplicativeexpression> <additiveexpression>&lt;-&gt;<multiplicativeexpression> <additiveexpression>&lt;-&gt;<multiplicativeexpression></multiplicativeexpression></additiveexpression></multiplicativeexpression></additiveexpression></multiplicativeexpression></additiveexpression></eof>
1 <b><multi< b=""> 2 <b><n< b=""> 3 <b> </b>&lt;</n<></b></multi<></b>	成乘法的一种特例了。  □ 复制代码  iplicativeExpression> ::=  Number> <multiplicativeexpression>&lt;*&gt;<number> <multiplicativeexpression><number></number></multiplicativeexpression></number></multiplicativeexpression>
<b>词法分析</b> : 词法分析部 表达式,它	是四则运算的定义了。 : <b>状态机</b> 分,我们把字符流变成 token 流。词法分析有两种方案,一种是状态机,一种是正常们是等效的,选择你喜欢的就好,这里我都会你介绍一下状态机。 我们可能产生四种输入元素,其中只有两种 token,我们状态机的第一个状态就是机
据第一个输。  1 var to 2 const 3 if 4 5 6 7 8 9 9 10 11 12 13 ) 14 15 16 } 17 if 18 19 20 21 ) 22 23 24 } 17 25 if 26 27 } 18 29 30 ) 31 32 } 33 } 34 const 35 if 36 37 38 39 40 41 42 43 44 45 ) 46 47	次字符来判断进入了哪种状态:
48 } 49 50 51 52 } 53 }	<pre>else {   emmitToken("Number", token.join(""));   token = [];   return start(char); // put back char</pre>
这里我的状值表示下一下面我们来的 To a standard standa	ion emmittoken(tyre, value) { Insole.lug(value);  nput = \"1022\frac{1}{2} \frac{1}{2} \f
13 state( 14 15	(Symbol('EOF')) 发现输出如下: <b> </b>
2 + 3 2 4 * 5 256 6 2 这是我们想	
做完了词法 们来写好函 1 functi 2 3 4 }	分析,我们开始进行语法分析,LL 语法分析根据每一个产生式来写一个函数,首先:
为了便于理	解,我们就不做流式处理了,实际上一般编译代码都应该支持流式处理。 设 token 已经都拿到了:
2 ty 3 va 4 }, { 5 ty 6 va 7 }, { 8 ty 9 va 10 }, { 11 ty 12 va 13 }, { 14 ty	ype:"Number", alue: "1024"  ype:"+" alue: "+"  ype:"Number", alue: "2"  ype:"*" alue: "*"  ype:"Number", alue: "256"
况: 1 <addit 2="" 3<="" <n="" td=""><td>对应着一个函数,例如:根据产生式,我们的 AdditiveExpression 需要处理三种情 □ 复制代码 tiveExpression&gt; ::= MultiplicativeExpression&gt; <additiveexpression>&lt;+&gt;<multiplicativeexpression> <additiveexpression>&lt;-&gt;<multiplicativeexpression></multiplicativeexpression></additiveexpression></multiplicativeexpression></additiveexpression></td></addit>	对应着一个函数,例如:根据产生式,我们的 AdditiveExpression 需要处理三种情 □ 复制代码 tiveExpression> ::= MultiplicativeExpression> <additiveexpression>&lt;+&gt;<multiplicativeexpression> <additiveexpression>&lt;-&gt;<multiplicativeexpression></multiplicativeexpression></additiveexpression></multiplicativeexpression></additiveexpression>
AdditiveEx	itveExpression 中就要写三个 if 分支,来处理三种情况。  Expression 的写法是根传入的节点,利用产生式合成新的节点  ion AdditiveExpression(source){ f(source[0].type === "MultiplicativeExpression") {    let node = {       type:"AdditiveExpression",
11 12 13 14 15 16 17 }	<pre>children:[source[0]] } source[0] = node; return node;  f(source[0].type === "AdditiveExpression" &amp;&amp; source[1].type === "+") {   let node = {      type:"AdditiveExpression",      operator:"+",      children:[source.shift(), source.shift(), MultiplicativeExpression(source)] } source.unshift(node);  f(source[0].type === "AdditiveExpression" &amp;&amp; source[1].type === "-") {   let node = {      type:"AdditiveExpression",      operator:"-",      children:[source.shift(), source.shift(), MultiplicativeExpression(source)] } source.unshift(node);</pre>
	我们就把解析好的 token 传给我们的顶层处理函数 Expression。
接下来,我 我们 Expres	又供本品信:
程在编译原 1 funct 2 i 3 4 5 6 7 8 9 }	我们就需要对产生式的首项层层展开,根据所有可能性调用相应的处理函数,这个这理中称为求 "closure"。  □ 每刺代码 cion Expression(source){ if(source[0].type === "AdditiveExpression" && source[1] && source[1].type === "EOF" let node = {     type:"Expression",     children:[source.shift(), source.shift()] } source.unshift(node); return node; chidditiveExpression(source); children:Expression(source);
14 i 15 16 17 18 19 20 21 }	<pre>cion AdditiveExpression(source){ if(source[0].type === "MultiplicativeExpression") {     let node = {         type:"AdditiveExpression",         children:[source[0]]     }     source[0] = node;     return AdditiveExpression(source);  if(source[0].type === "AdditiveExpression" &amp;&amp; source[1] &amp;&amp; source[1].type === "+")     let node = {         type:"AdditiveExpression",         operator:"+",</pre>
36 37 38 39 40 41 42 43 44 45 46 47 48 i 49 50 M 51 r 52 }	<pre>if(source[0].type === "AdditiveExpression" &amp;&amp; source[1] &amp;&amp; source[1].type === "-")   let node = {         type:"AdditiveExpression",         operator:"-",         children:[]   }   node.children.push(source.shift());   node.children.push(source.shift());   MultiplicativeExpression(source);   node.children.push(source.shift());   source.unshift(node);   return AdditiveExpression(source);</pre>
55 56 57 58 59 60 61	<pre>if(source[0].type === "Number") {     let node = {         type:"MultiplicativeExpression",         children:[source[0]]     }     source[0] = node;     return MultiplicativeExpression(source); } if(source[0].type === "MultiplicativeExpression" &amp;&amp; source[1] &amp;&amp; source[1].type ===     let node = {         type:"MultiplicativeExpression",         operator:"*",         children:[]     }     node.children.push(source.shift());</pre>
75 76 77 78 79 80 81 82 83 84 85	<pre>if(source[0].type === "MultiplicativeExpression"&amp;&amp; source[1] &amp;&amp; source[1].type ===     let node = {         type:"MultiplicativeExpression",         operator:"/",         children:[]     }     node.children.push(source.shift());     node.children.push(source.shift());     node.children.push(source.shift());     source.unshift(node);     return MultiplicativeExpression(source);</pre>
96 t 97 v 98 }, { 99 t 100 v 101 }, { 102 t 106 v 107 }, { 108 t 109 v 110 }, { 111 t 112 v 113 }, { 114 t	return MultiplicativeExpression(source);  type:"*",  value: "*"  type:"H",  value: "2"  type:"*",  value: "*"  Value: "*"  type:"Number",  value: "5"  type:"E0=  type:E0=  type
<b>解释执行</b> 得到了 AST 了,那么接	ole.log(ast); 一之后,最困难的一步我们已经解决了。这里我们就不对这颗树做任何的优化和精简 下来,直接进入执行阶段。我们只需要对这个树做遍历操作执行即可。
1 2 functi 3 if 4 5 } 6 if 7 8 9	<pre>f(node.type === "AdditiveExpression") {    if(node.operator === '-') {       return evaluate(node.children[0]) - evaluate(node.children[2]);    }    if(node.operator === '+') {</pre>
11 12 13 14 } 15 16 17 18 19 20 21 22 23 }	<pre>return evaluate(node.children[0]) + evaluate(node.children[2]); } return evaluate(node.children[0])  f(node.type === "MultiplicativeExpression") {    if(node.operator === '*') {       return evaluate(node.children[0]) * evaluate(node.children[2]);    }    if(node.operator === '/') {       return evaluate(node.children[0]) / evaluate(node.children[2]);    }    return evaluate(node.children[0])  f(node.type === "Number") {    return Number(node.value); }</pre>
<b>总结</b> 在这个小实 JavaScript 通过实验, 最后留给你 • 补全 emi	
	同学留言给我。
	每天 10 分钟,重构你的前端知识体系 winter 程劢非 前手机淘宝前端负责人  新版升级:点击「 <a>請朋友读」,10位好友免费读,邀请订阅更有现金奖励。</a>