你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+`、`-`、`*`、`/` 之一
- Whitespace: `<sp>`
- LineTerminator：`<LF>` `<CR>`

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>

<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
```

```
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
```

```
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];

    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
```

```
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。
- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。

你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+` 、`-`、 `*`、 `/` 之一
- Whitespace: `<sp>`
- LineTerminator： `<LF>` `<CR>`

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>

<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
```

```
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
```

```
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
```

```
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];

    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
```

```
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。
- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。

你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+` 、`-`、 `*`、 `/` 之一
- Whitespace: `<sp>`

- LineTerminator：<LF> <CR>

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>

<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
```

```
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
```

```
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><->><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];

    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
```

```
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。
- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。

你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+` 、`-`、 `*`、 `/` 之一
- Whitespace: `<sp>`
- LineTerminator: `<LF>` `<CR>`

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>

<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
```

```
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];
```

```
    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。

- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。

你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+`、`-`、`*`、`/` 之一
- Whitespace: `<sp>`
- LineTerminator：`<LF>` `<CR>`

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>

<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
```

```
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];

    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。
- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。

你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+` 、`-`、 `*`、 `/` 之一
- Whitespace: `<sp>`
- LineTerminator： `<LF>` `<CR>`

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>
```

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];

    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。
- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。

你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+`、`-`、 `*`、 `/` 之一
- Whitespace: `<sp>`
- LineTerminator：`<LF>` `<CR>`

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>

<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
```

```
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
```

```
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];

    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
```

```
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。
- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。

你好，我是winter。

在前面的课程中，我在JavaScript和CSS的部分，多次提到了编译原理相关的知识。这一部分的知识，如果我们从编译原理“龙

书”等正规的资料中学习，就会耗费掉不少的时间，所以我在这里设计了一个小实验，帮助你快速理解编译原理相关的知识。

今天的内容比较特殊，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，如果你正在收听音频，可以点击文章查看详情。

分析

按照编译原理相关的知识，我们来设计一下工作，这里我们分成几个步骤。

- 定义四则运算：产出四则运算的词法定义和语法定义。
- 词法分析：把输入的字符串流变成token。
- 语法分析：把token变成抽象语法树AST。
- 解释执行：后序遍历AST，执行得出结果。

定义四则运算

四则运算就是加减乘除四种运算，例如：

```
1 + 2 * 3
```

首先我们来定义词法，四则运算里面只有数字和运算符，所以定义很简单，但是我们还要注意空格和换行符，所以词法定义大概是下面这样的。

- Token
  - Number: `1 2 3 4 5 6 7 8 9 0` 的组合
  - Operator: `+`、`-`、 `*`、 `/` 之一
- Whitespace: `<sp>`
- LineTerminator： `<LF>` `<CR>`

这里我们对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。

接下来我们来定义语法，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JavaScript标准里面就是一种跟BNF类似的自创语法。

不过语法定义的核心思想不会变，都是几种结构的组合产生一个新的结构，所以语法定义也叫语法产生式。

因为加减乘除有优先级，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：

```
<Expression> ::=
    <AdditiveExpression><EOF>

<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

这种BNF的写法类似递归的原理，你可以理解一下，它表示一个列表。为了方便，我们把普通数字也得当成乘法的一种特例了。

```
<MultiplicativeExpression> ::=
    <Number>
    |<MultiplicativeExpression><*><Number>
    |<MultiplicativeExpression></><Number>
```

好了，这就是四则运算的定义了。

词法分析：状态机

词法分析部分，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，一种是正则表达式，它们是等效的，选择你喜欢的就好，这里我都会你介绍一下状态机。

根据分析，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根据第一个输入字符来判断进入了哪种状态：

```
var token = [];
const start = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
```

```
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    }
    if(char === '+'
        || char === '-'
        || char === '*'
        || char === '/'
    ) {
        emmitToken(char, char);
        return start
    }
    if(char === ' ') {
        return start;
    }
    if(char === '\r'
        || char === '\n'
    ) {
        return start;
    }
}
const inNumber = char => {
    if(char === '1'
        || char === '2'
        || char === '3'
        || char === '4'
        || char === '5'
        || char === '6'
        || char === '7'
        || char === '8'
        || char === '9'
        || char === '0'
    ) {
        token.push(char);
        return inNumber;
    } else {
        emmitToken("Number", token.join(""));
        token = [];
        return start(char); // put back char
    }
}
```

这个状态机非常简单，它只有两个状态，因为我们只有Number不是单字符的token。

这里我的状态机实现是非常经典的方式：用函数表示状态，用if表示状态的迁移关系，用return值表示下一个状态。

下面我们来运行一下这个状态机试试看：

```
function emmitToken(type, value) {
    console.log(value);
}

var input = "1024 + 2 * 256"

var state = start;

for(var c of input.split(''))
    state = state(c);

state(Symbol('EOF'))
```

运行后我们发现输出如下：

```
1024
+
2
*
256
```

这是我们想要的答案。

语法分析：LL

做完了词法分析，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，首先我们来写好函数名：

```
function AdditiveExpression( ){


}
function MultiplicativeExpression(){


}
```

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。

所以我们假设token已经都拿到了：

```
var tokens = [{
    type:"Number",
    value: "1024"
}, {
    type:"+"
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*"
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
```

每个产生式对应着一个函数，例如：根据产生式，我们的AdditiveExpression需要处理三种情况：

```
<AdditiveExpression> ::=
    <MultiplicativeExpression>
    |<AdditiveExpression><+><MultiplicativeExpression>
    |<AdditiveExpression><-><MultiplicativeExpression>
```

那么AddititveExpression中就要写三个if分支，来处理三种情况。

AdditiveExpression的写法是根传入的节点，利用产生式合成新的节点

```
function AdditiveExpression(source){
    if(source[0].type === "MultiplicativeExpression") {
        let node = {
            type:"AdditiveExpression",
            children:[source[0]]
        }
        source[0] = node;
        return node;
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "+") {
        let node = {
            type:"AdditiveExpression",
            operator:"+",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
    if(source[0].type === "AdditiveExpression" && source[1].type === "-") {
        let node = {
            type:"AdditiveExpression",
            operator:"-",
            children:[source.shift(), source.shift(), MultiplicativeExpression(source)]
        }
        source.unshift(node);
    }
}
```

那么下一步我们就把解析好的token传给我们的顶层处理函数Expression。

```
Expression(tokens);
```

接下来，我们看Expression该怎么处理它。

我们Expression收到第一个token，是个Number，这个时候，Expression就傻了，这是因为产生式只告诉我们，收到了

AdditiveExpression 怎么办。

这个时候，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，这个过程在编译原理中称为求“closure”。

```
function 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;
    }
    AdditiveExpression(source);
    return Expression(source);
}
function 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" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression" && source[1] && 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);
    }
    if(source[0].type === "AdditiveExpression")
        return source[0];
    MultiplicativeExpression(source);
    return AdditiveExpression(source);
}
function MultiplicativeExpression(source){
    if(source[0].type === "Number") {
        let node = {
            type:"MultiplicativeExpression",
            children:[source[0]]
        }
        source[0] = node;
        return MultiplicativeExpression(source);
    }
    if(source[0].type === "MultiplicativeExpression" && source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression"&& source[1] && 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);
    }
    if(source[0].type === "MultiplicativeExpression")
        return source[0];

    return MultiplicativeExpression(source);
};

var source = [{
    type:"Number",
    value: "3"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "300"
}, {
    type:"+",
    value: "+"
}, {
    type:"Number",
    value: "2"
}, {
    type:"*",
    value: "*"
}, {
    type:"Number",
    value: "256"
}, {
    type:"EOF"
}];
var ast = Expression(source);

console.log(ast);
```

解释执行

得到了AST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了，那么接下来，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。

我们根据不同的节点类型和其它信息，写if分别处理即可：

```
function evaluate(node) {
    if(node.type === "Expression") {
        return evaluate(node.children[0])
    }
    if(node.type === "AdditiveExpression") {
        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])
    }
    if(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])
    }
    if(node.type === "Number") {
        return Number(node.value);
    }
}
```

总结

在这个小实验中，我们通过一个小实验学习了编译原理的基本知识，小实验的目的是帮助你理解JavaScript课程中涉及到的编译原理基本概念，它离真正的编译原理学习还有很大的差距。

通过实验，我们了解了产生式、词法分析、语法分析和解释执行的过程。

最后留给你一些挑战，你可以根据自己的水平选择：

- 补全emmitToken，使得我们的代码能完整工作起来。
- 为四则运算加入小数。
- 引入负数。
- 添加括号功能。

欢迎写好的同学留言给我。