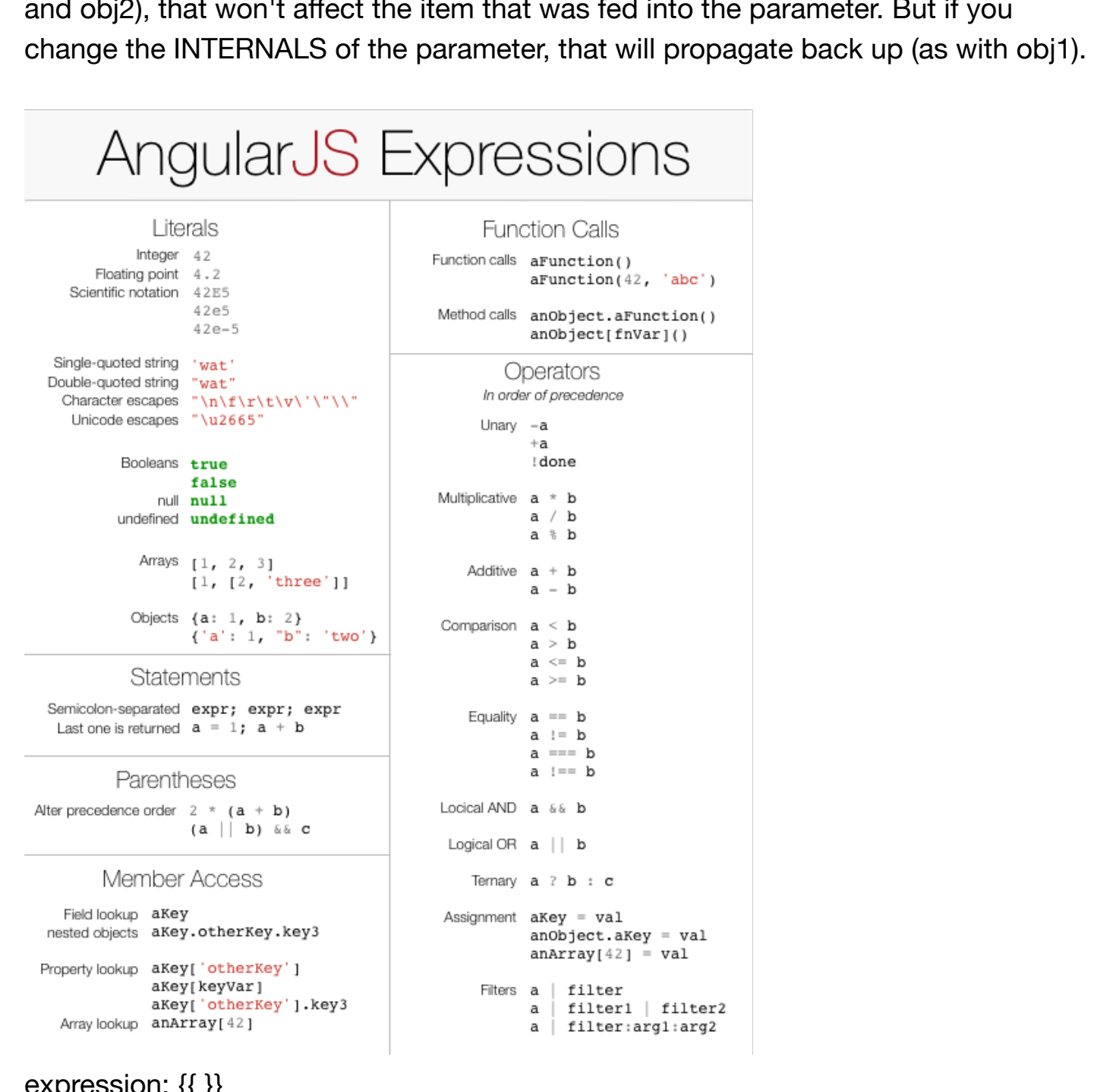
Javascript是值传递还是引用传递?
http://stackoverflow.com/questions/518000/is-javascript-a-pass-by-reference-or-pass-by-value-language?lq=1
the situation is that the item passed in is passed by value. But the item that is passed by value is itself a reference. Technically, this is called call-by-sharing. 在函数内，如果改参数变量本身没有效果，但是如果改参数的INTERNAL property，会改变参数内部的值。
In practical terms, this means that if you change the parameter itself (as with num and obj2), that won't affect the item that was fed into the parameter. But if you change the INTERNALS of the parameter, that will propagate back up (as with obj1).



expression: {{ }}

Chapter 5: Literal expressions

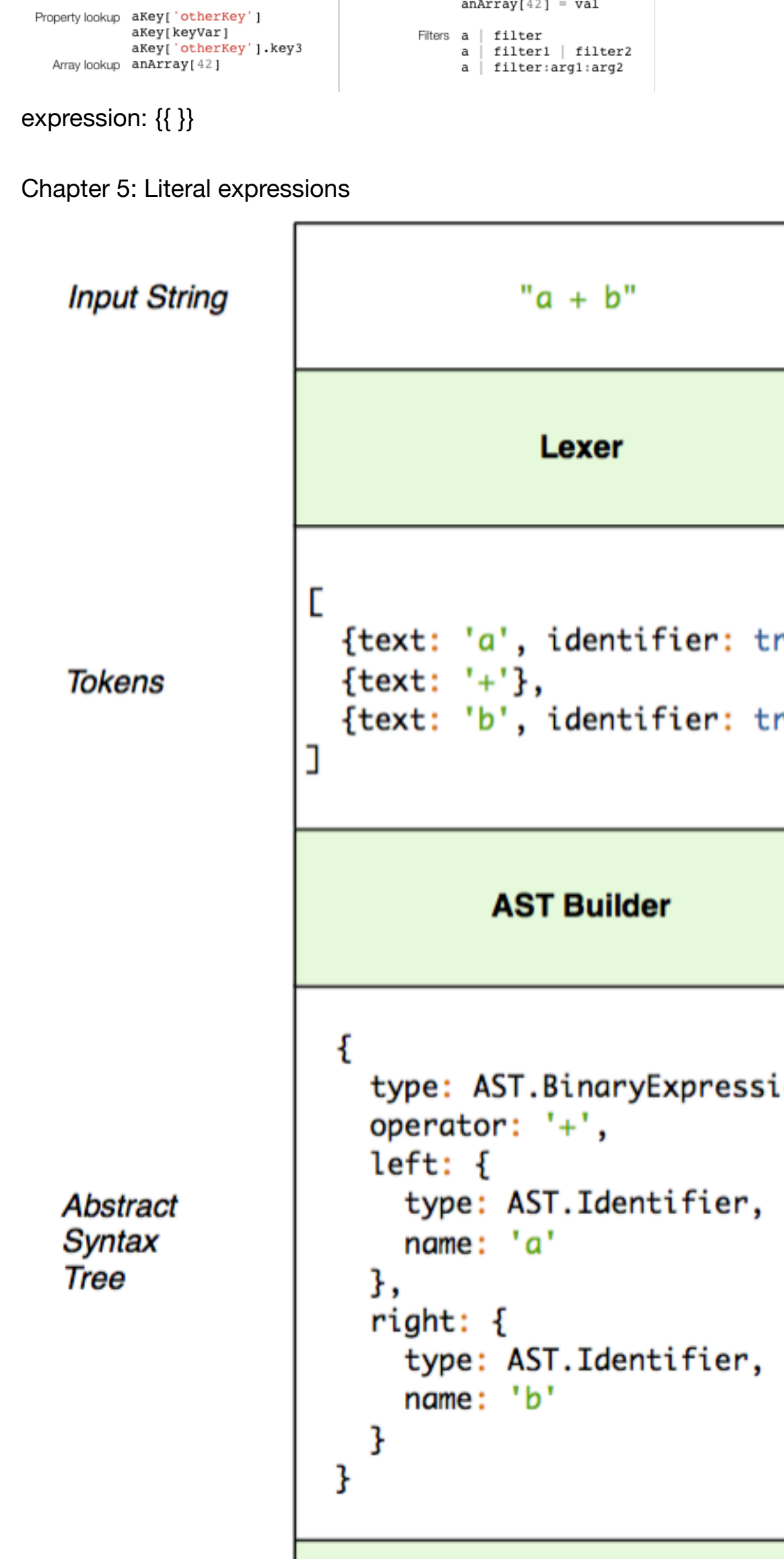


Figure 5.1: The expression parsing pipeline

Lexer: expression string => tokens parsed from the string, 如 a+b 变为"a, +, b"
AST Builder: tokens => Abstract Syntax Tree, 例如:

```
{
  type: AST.BinaryExpression,
  operator: '+',
  left: {
    type: AST.Identifier,
    name: 'a'
  },
  right: {
    type: AST.Identifier,
    name: 'b'
  }
}
```

AST Compiler: AST syntax tree => Javascript function，最终得到一个Javascript函数
Parser负责合并上面描述的这些操作的底层步骤

Lexer, AST Builder, AST Compiler, Parse： constructor function，prototype添加function

1，解析一个字符串为一个数："42"为42
tokenizer需要text 和 numeric value, 得到tokens:

```
{
  text: '42',
  value: 42
}
```

AST Builder将tokens解析为一个AST：

```
{
  type: AST.Program,
  body: {
    type: AST.Literal,
    value: 42
  }
}
```

AST Compiler将tree转换成一个function，用一个递归的recurse方法来遍历整个树。this.state = {body:[]}，将遍历到的value转化成return语句压到数组里
最后返回一个return new Function(this.state.body.join(''));
最后变成了 function(){return 42;} 完成解析
2，解析小数点，例如4.2，.42（0被省略）
修改readNumber方法，让小数点也能够被认识为value即可，javascript自带支持小数点（似乎没考虑两个点的情况）
3，解析科学计数法，如42e3，或者.42e3，4200e-2, .42e+2, .42E2等，这个时候修改readNumber方法，能够解析+,-,e这几个符号也就可以过测试了，但是太草率，因为同时对于invalid notation需要抛出错误，所以没那么简单。
步骤如下：1，引入exponent operator，e后面可以跟一个+,-或者数字；2，readNumber，考虑各种情况
4，解析字符串，如"'abc'"
解析成function的时候会变成function(){ return abc; }，需要使用escape方法，给abc加引号。
考虑首尾的引号需要对应，否则抛出 unmatched quote
对于escape character的解析：包括\n, \f, \r, \t, \v, \', \", 以及unicode如\u00A0

```
ASTCompiler.prototype.escape = function(value){
    if (_.isString(value)){
        return '\'' +
value.replace(this.stringEscapeRegex,
this.stringEscapeFn) + '\'';
    }else{
        return value;
    }
};
ASTCompiler.prototype.stringEscapeRegex = /[^ a-zA-Z0-9]/g;
ASTCompiler.prototype.stringEscapeFn = function(c){
    return '\\u'+('0000'+
c.charCodeAt(0).toString(16)).slice(-4);
    //get the numeric unicode value, and convert it
into corresponding hexadecimal unicode escape sequence
};
```

5，解析true, false, null
这些单词称为identifier tokens
6，解析空白，几乎全部空白都可以忽略
7，解析数组，空白数组解析为[],
Lexer直接作为text解析成token，将[,],逗号隔开的元素push进tokens
AST增加expect函数，因为现在需要处理多个tokens，expect检查下一个token是不是我们所期望的，如果是的话就return，并且从this.tokens中移除该token，方便向下解析。
增加arrayDeclaration函数，调用consume方法吸收和数组有关的tokens并且构造对应的AST节点。consume函数和expect函数差不多，但是如果一个对应的token没找到的话consume函数会抛出一个异常。
增加一个新的节点type: AST.ArrayExpression，然后交给Compiler。
8，解析object
和解析数组一个道理，先读出所有的key, value，然后压到一个properties数组里，组成一个type: AST.ObjectExpression的对象，交给Compiler。Compiler取出每个key, value对，key作escape，value做recurse，返回 key+":"+value;

总结：

1. expression parser的三个阶段： Lexing, AST building, AST compilation
2. parsing的结果是一个javascript function
3. 如何解析各个对象

第六章：Lookup and function call expressions
给Angular expression language增加accessing data on scopes和manipulating data的功能，包括access and assign attributes, call functions, security measures等等

```
it('looks up an attribute from the scope', function()
{
    var fn = parse('aKey');
    expect(fn({aKey: 42})).toBe(42);
    expect(fn({})).toBeUndefined();
});
```

这里传给fn的是一个object，实际应用中几乎肯定是一个scope对象。'aKey'被解析成一个identifier token，然后变成一个Identifier AST node。现在我们要添加attribute lookup的支持。
compile中new Function添加一个参数s，变成return (s).xxx;，这样通过上面这个unit test。
AngularJS对于undefined比较宽容，引用不存在的object attribute一般不会抛出exception，所以lookup之前需要先判断是否是undefined。在解析的时候会引入临时变量，变成类似'var v1;if(s){v1=(s).anObject;}return v1;'的形式
var hoisting: declaring a variable anywhere in the code is equivalent to declaring it at the top. 变量声明的位置不重要。所以可以一次性声明完解析过程中所需要的全部变量。

到解析object为止，AST节点的Type:
AST.Program = 'Program';
AST.Literal = 'Literal';
AST.ArrayExpression = 'ArrayExpression';
AST.ObjectExpression = 'ObjectExpression';
AST.Property = 'Property';
AST.Identifier = 'Identifier';
AST.ThisExpression = 'ThisExpression';
AST.MemberExpression = 'MemberExpression';
解析A.B: 关于.符号

```
if (this.expect('.')) { primary = {
type: AST.MemberExpression, object: primary,
property: this.identifier()
}; }
return primary;
```

复用primary node作为新的primary node的object member，实现关于.的look up
如果有三个以上的点怎么办？例如A.B.C，trick：改成while循环，while (this.expect('.'))，重复地复用。
最后被解析出来的形式类似于:

```
function(s) {
var v0, v1, v2, v3; if (s) {

v3 = s.aKey; }

if (v3) {

  v2 = v3.secondKey;
}

if (v2) {

  v1 = v2.thirdKey;
}

if (v1) {

  v0 = v1.fourthKey;
}


return v0; }
```

到目前为止parse函数只有一个参数scope 's'，现在添加一个argument叫做locals 'l'，parsed expressions应该先从locals object进行look up，然后再去scope找。这样做的作用是可以使得一些属性对于表达式可用，但不用把它们和scope进行绑定。例如ngClick directive，是和$event绑定的

```
it('uses locals instead of scope when there is a matching key', function() {
    var fn = parse('aKey');
    var scope = {aKey: 42};
    var locals = {aKey: 43};
    expect(fn(scope, locals)).toBe(43);
```

解析了之后现在'l'中找有无该属性，如果没有则去's'中找。下面是拼出代码的实现：

```
case AST.Identifier:
    var intoId = this.nextId();
    this.if_(this.getHasOwnProperty('l', ast.name),
        this.assign(intoId,
this.nonComputedMember('l', ast.name))); //if (s){
v0=...; } return v0;
    this.if_(this.not(this.getHasOwnProperty('l',
ast.name)) + ' && s',
        this.assign(intoId,
this.nonComputedMember('s', ast.name)));
    return intoId;
```

关于computed attribute lookup:
non-computed lookup: access attributes on scoped using dot operator
computed attribute lookup: using square bracket notation，aKey[anotherKey], 即这里的attribute的名字是它自己本身被look up from the scope or computed in some other way，中括号内可以是变量
computed attribute lookup解析出来的函数是：
LOG:
'var v1,v2,v3,v4;if(l&&('lock' in l)){v2=(l).lock;}if(!(l&&('lock' in l)) &&s){v2=(s).lock;}if(l&&('keys' in l)){v4=(l).keys;}if(!(l&&('keys' in l)) &&s){v4=(s).keys;}if(v4){v3=(v4)['aKey'];}if(v2){v1=(v2)[v3];}return v1;'

关于Function call：
1，寻找function，2，添加括号来调用函数，其实与object lookup并无明显区别
Function call需要支持参数，先解析括号之间的参数，recurse每个参数并将结果放到数组里，然后返回 callee + && + callee + (+ args.join(,) +) ;

method call：When a function is attached to an object as an attribute and invoked by first dereferencing it from the object using a dot or square brackets, the function is invoked as a method. 类似于object.method()（noncomputed）或者anObject["anMethod"]()（computed）
相关的解析工作在AST compiler中做，在recurse函数中增加CallExpression分支，使得this指向原先Angular expression指向的对象。
引入一个"call context"对象，保存了这个method call相关的上下文信息。
recurse函数增加context参数，传入method call之后，context会增加三个属性：1，context，方法的owning object，最终称为this，2，name，该方法的属性名，3，computed，是否是computed property

对于standalone function来说，this指的是scope，对于局部变量绑定的function，this指的是locals，也需要解析出来。这个是在recurse中的Identifier分支，context要么是'l'要么是's'

assigning values：解析assignments，也就是put data on scopes。assign不仅可赋值给identifiers，也可以给computed and non computed properties。
Assignment不是之前的"primary" AST node，新建一个函数assignment。先解析等号左边的token，这是一个primary node，然后解析右边的primary node。

```
AST.prototype.assignment = function() { var left = this.primary();
if (this.expect('=')) {

var right = this.primary();

return {type: AST.AssignmentExpression, left: left, right: right}; }


return left; };
```

因为assignment当没有等号的时候等效于primary，所以将一些可能出现assignment的地方，原来的primary()替换成assignment()
ASTCompiler的解析：

```
case AST.AssignmentExpression:
    var leftContext = {};
    this.recurse(ast.left, leftContext);
    var leftExpr;
    if (leftContext.computed){
        leftExpr =
this.computedMember(leftContext.context,
leftContext.name);
    }
    else{
        leftExpr =
this.nonComputedMember(leftContext.context,
leftContext.name);
    }
    return this.assign(leftExpr,
this.recurse(ast.right));
```

Angular expressions的nested assignment的一个有趣特性：如果赋值给路径中不存在的object，则这些对象会被创建。解决办法：recurse添加一个新的布尔值参数，决定是否创建可能有的新的对象。
安全性保证：
1, Member access
Function的 constructor attribute有可能被攻击者利用，例如aFunction.constructor("return window;")()。以及其他一些可能被利用的issues，如__proto__, __defineGetter__, __lookupGetter__, __defineSetter__, and __lookupSetter__等等。Angular需要disallow这些属性在expression中的访问。
对于non-computed member access在AST Compiler里面注意name，如果碰到非法name直接抛出错误。而对于computed member access，因为在parse time我们不知道属性名字，所以需要在runtime的时候调用ensureSafeMemberName函数。
2, Ensuring safe objects
一些object调用也比较危险，如 window 和 DOM元素、以及function constructor、和Object 对象，调用window的alias也应当被抛出错误，如anObject.wnd应该抛出错误。并且unsafe object也不应当允许作为参数使用。

静态parse time检查：

```
function ensureSafeMemberName(name){
    if (name==='constructor' || name === '__proto__'
||
        name === '__defineGetter__' || name ===
'__defineSetter__'||
        name === '__lookupGetter__' || name ===
'__lookupSetter__') {
        throw 'Attempting to access a disallowed field
in Angular expressions!';
    }
}

function ensureSafeObject(obj){
    if (obj){
        if (obj.window === obj){
            throw 'Referencing window in Angular
expressions is disallowed!';
        } else if (obj.children && (obj.nodeName ||
(obj.prop && obj.attr && obj.find))){
            throw 'Referencing DOM nodes in Angular
expressions is disallowed!';
        } else if (obj.constructor === obj){
            throw 'Referencing Function in Angular
expressions is disallowed!';
        }else if (obj === Object){
            throw 'Referencing Object in Angular
expressions is disallowed!';
        }
    }
    return obj;
}
```

动态runtime检查：

```
ASTCompiler.prototype.addEnsureSafeMemberName =
function(expr){

this.state.body.push('ensureSafeMemberName('+expr+');'
);
};

ASTCompiler.prototype.addEnsureSafeMemberName =
function(expr){

this.state.body.push('ensureSafeObject('+expr+');');
};
```

3，Ensuring Safe Functions
Angular希望阻止rebind，即访问call和apply（也就是bind）方法。这样可以防止injection attack

```
var CALL = Function.prototype.call;
var APPLY = Function.prototype.apply;
var BIND = Function.prototype.bind;
function ensureSafeFunction(obj){
    if (obj){
        if (obj.constructor === obj){
            throw 'Referencing Function in Angular
expressions is disallowed!';
        } else if (obj===CALL || obj===APPLY ||
obj===BIND){
            throw 'Referencing call, apply, or bind in
Angular expressions is disallowed!' ;
        }
    }
    return obj;
}
```

总结：

1. computed and non-computed attribute的lookup
2. locals，可以覆盖scope attributes
3. function calls的解析
4. this context的解析
5. 安全措施，包括Function constructor, potentially dangerous objects, function call
6. 关于assignment的解析，simple attribute和nested attribute
7. Angular expression的forgiving nature，missing attributes不抛出exception，而是新建这个attribute