



下载APP



04 | 如何让我们的语言支持变量和类型？

2021-08-16 宫文学

《手把手带你写一门编程语言》

课程介绍 >



讲述：宫文学

时长 18:55 大小 17.33M



你好，我是宫文学。

到目前为止，我们的语言已经能够处理语句，也能够处理表达式，并且都能够解释执行了。不过，我们目前程序能够处理的数据，还都只是字面量而已。接下来，我们要增加一个重要的能力：**支持变量**。

在程序中声明变量、对变量赋值、基于变量进行计算，是计算机语言的基本功能。只有支持了变量，我们才能实现那些更加强大的功能，比如，你可以用程序写一个计算物体下落的速度和位置，如何随时间变化的公式。这里的时间就是变量，通过给时间变量赋予不同的值，我们可以知道任意时间的物体速度和位置。



这一节，我就带你让我们手头上的语言能够支持变量。在这个过程中，你还会掌握语义分析的更多技能，比如类型处理等。

好了，我们已经知道了这一节的任务。那么第一步要做什么呢？你可以想到，我们首先要了解与处理变量声明和初始化有关的语法规则。

与变量有关的语法分析功能

在 TypeScript 中，我们在声明变量的时候，可以指定类型，这样有助于在编译期做类型检查：

```
1 let myAge : number = 18;
```

[复制代码](#)

如果编译成 JavaScript，那么类型信息就会被抹掉：

```
1 var myAge = 18;
```

[复制代码](#)

不过，因为我们的目标是教给你做类型分析的方法，以后还要静态编译成二进制的机器码，所以我们选择的是 TypeScript 的语法。

此外，在上面的例子中，变量在声明的时候就已经做了初始化。你还可以把这个过程拆成两步。第一步的时候，只是声明变量，之后再给它赋值：

```
1 let myAge : number;  
2 myAge = 18;
```

[复制代码](#)

知道了如何声明变量以后，你就可以试着写出相关的语法规则。我这里给出一个示范的版本：

```
1 variableDecl : 'let' Identifier typeAnnotation? ('=' singleExpression)?;
```

[复制代码](#)

```
2 typeAnnotation : ':' typeName;
```

学完了前面 3 节课，我相信你现在应该对阅读语法规则越来越熟悉了。接下来，就要修改我们的语法分析程序，让它能够处理变量声明语句。这里有什么关键点呢？

这里你要注意的，我们采用的语法分析的算法是 LL 算法。而在 02 讲中，我们知道 LL 算法的关键是计算 First 和 Follow 集合。

首先是 First 集合。在变量声明语句的上一级语法规则（也就是 statement）中，要通过 First 集合中的不同元素，准确地确定应该采用哪一条语法规则。由于变量声明语句是用 let 开头的，这就使它非常容易辨别。只要预读的 Token 是 let，那就按照变量声明的语法规则来做解析就对了。

接下来是 Follow 集合。在上面的语法规则中你能看出，变量的类型注解和初始化部分都是可选的，它们都使用了？号。

由于类型注解是可选的，那么解析器在处理了变量名称后，就要看一下后面的 Token 是什么。如果是冒号，由于冒号是在 typeAnnotation 的 First 集合中，那就去解析一个类型注解；如果这个 Token 不是冒号，而是 typeAnnotation 的 Follow 集合中的元素，就说明当前语句里没有 typeAnnotation，所以可以直接略过。

那 typeAnnotation 的 Follow 集合有哪些元素呢？我就不直说了，你自己来分析一下吧。

再往后，由于变量的初始化部分也是可选的，还要计算一下它的 Follow 集合。你能看出，这个 Follow 集合只有；号这一个元素。所以，在解析到变量声明部分的时候，我们可以通过预读准确地判断接下来该采取什么动作：

如果预读的 Token 是 = 号，那就是继续做变量初始化部分的解析；

如果预读的 Token 是；号，那就证明该语句没有变量初始化部分，因此可以结束变量声明语句的解析了；

如果读到的是 = 号和；号之外的任何 Token 呢，那就触发语法错误了。

相关的实现很简单，你参考一下这个示例代码：

```
1 let t1 = this.scanner.peek();
2 //可选的类型标注
3 if (t1.text == ':'){
4     this.scanner.next();
5     t1 = this.scanner.peek();
6     if (t1.kind == TokenKind.Identifier){
7         this.scanner.next();
8         varType = t1.text;
9         t1 = this.scanner.peek();
10    }
11    else{
12        console.log("Error parsing type annotation in VariableDecl");
13        return null;
14    }
15 }
16
17 //可选的初始化部分
18 if (t1.text == '='){
19     this.scanner.next();
20     init = this.parseExpression();
21 }
22
23 //分号，结束变量声明
24 t1 = this.scanner.peek();
25 if (t1.text == ';'){
26     this.scanner.next();
27     return new VariableDecl(varName, varType, init);
28 }
29 else{
30     console.log("Expecting ; at the end of varaible declaration, while we meet");
31     return null;
32 }
```

采用增强后的语法分析程序，去解析 “let myAge:number = 18;” 这条语句，就会形成下面的 AST，这表明我们的解析程序是有效的：

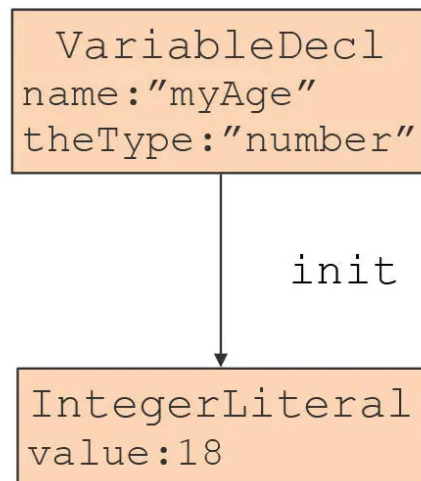


图1：变量声明语句对应的AST

在这个 AST 中，我们用一个 `VariableDecl` 节点代表一个变量声明语句。如果这个变量声明语句带有初始化部分，那么我们就用一个单独的节点来表示用于初始化的表达式。

好了，生成了这个 AST，说明现在我们已经支持变量声明和初始化语句了。那么如何支持变量赋值语句呢？其实，在大多数语言中，我们把赋值运算看作是跟加减乘除一样性质的运算。`myAge=18` 被看作是一个赋值表达式。你还可以把 `=` 换成 `+=`、`-=` 等运算符，形成像 `myAge += 2` 这样的表达式。如果这些表达式后面直接跟一个 `;` 号，那就变成了一个表达式语句了。

[复制代码](#)

```
1 expressionStatement : singleExpression ';' ;
```

而在上一节，我们已经能够用运算符优先级算法来解析各种二元表达式。不过，**赋值表达式跟加减乘数等表达式有一点不同：它是右结合的**。比如，对于 `a=b=3` 这个表达式，是先把 3 赋值给 `b`，再把 `b` 赋值给 `a`。所以呢，对应的语法解析程序也要做一下调整，从而生成体现右结合的 AST。

具体我们可以参照解析器的 `parseAssignment()` 方法的代码。由于赋值表达式的优先级比较低，按照自顶向下的解析原则，可以先解析赋值表达式。而赋值表达式的每个子节点，就是一个其他的二元表达式。所以，我们的语法规则可以大致修改成下面的样子：

[复制代码](#)

```
1 singleExpression : assignment;  
2 assignment : binary (AssignmentOp binary)*  
3 binary: primary (BinaryOp primary)*
```

那现在，我们采用修改完毕的解析器，试着解析一下 “`a=b=c+3;`” 这个语句，会打印出下面这个 AST：

```
ExpressionStatement  
  Binary:=  
    Variable: a, resolved  
  Binary:=  
    Variable: b, resolved  
  Binary:+  
    Variable: c, resolved  
    3
```

图2：解析赋值语句`a=b=3;`形成的AST

讲到这里，其实我们与变量有关的语法分析工作就完成了。接下来的工作是什么呢？是实现一些有关的语义分析功能。

语义分析：引用消解和符号表

不知道你还记不记得，在第 1 节，我们曾经接触过一点语义分析功能。那个时候我们主要是在函数调用和函数声明之间做了链接。这个工作叫做“引用消解”（Reference Resolve），或者“名称消解”。

对于变量，我们也一样要做这种消解工作。比如下面的示例程序 “`myAge + 2`” 这个表达式中，必须知道这个 `myAge` 是在哪里声明的。

```
1 let myAge : number = 18;  
2 let yourAge : number;  
3 yourAge = myAge + 2;
```

请你回忆一下，在 [01 讲](#) 里，我们是怎么实现引用消解的呢？是在 AST 的不同节点之间直接做了个链接。这个实现方式比较简单，但实际上不太实用。为什么呢？因为你每次在程序里遇到一个函数或者变量的话，都要遍历 AST 才能找到它的定义，这有点太麻烦了。

我有一个效率更高的方法，就是建立一个符号表（Symbol Table），用来保存程序中的所有符号。那什么是符号呢？符号就是我们在程序中自己定义对象，如变量、函数、类等。它们通常都对应一个标识符，作为符号的名称。

采用符号表以后，我们可以在做语义分析的时候，每当遇到函数声明、变量声明，就把符号加到符号表里去。这样，如果别的地方还要使用该符号，我们就直接到符号表里去查就行了。

那这个符号表里要保存哪些信息呢？其实就是我们声明的那些对象的定义信息，比如：

- 名称，也就是变量名称、类名称和函数名称等；


- 符号种类，也就是变量、函数、类等；

- 其他必要的信息，如函数的签名、变量的类型、类的成员，等等。

你不要小看了这个符号表，符号表在整个编译过程中都有着重要的作用，它的生命周期可以横跨整个编译过程。有的编译器，在词法分析的时候就会先形成符号表，因为每个标识符肯定会对应一个符号。


不过，大部分现代的编译器，都是在语义分析阶段开始建立符号表的。在 Java 的字节码文件里，也是存在符号表的，也就是各个类、类的各个成员变量和方法的声明信息。对于 C/C++ 这样的程序，如果要生成可调试的目标代码，也需要用到符号表。所以，在后面的课程中，我们会不断的跟符号表打交道。

初步了解了符号表以后，我们再回到引用消解的任务上来。首先，我们要建立符号表，这需要对 AST 做第一次遍历：

 复制代码

```
1  /**
2   * 把符号加入符号表。
3   */
4  export class Enter extends AstVisitor{
5      symTable : SymTable;
6      constructor(symTable:SymTable){
7          super();
8          this.symTable = symTable;
9      }
10
11     /**
12      * 把函数声明加入符号表
13      * @param functionDecl
14      */
15     visitFunctionDecl(functionDecl: FunctionDecl):any{
16         if (this.symTable.hasSymbol(functionDecl.name)){
17             console.log("Duplicate symbol: "+ functionDecl.name);
18         }
19         this.symTable.enter(functionDecl.name, functionDecl, SymKind.Function)
20     }
21
22     /**
23      * 把变量声明加入符号表
24      * @param variableDecl
25      */
26     visitVariableDecl(variableDecl : VariableDecl):any{
27         if (this.symTable.hasSymbol(variableDecl.name)){
28             console.log("Duplicate symbol: "+ variableDecl.name);
29         }
30         this.symTable.enter(variableDecl.name, variableDecl, SymKind.Variable)
31     }
32 }
```

然后我们要基于符号表做引用消解，需要对 AST 再做第二次的遍历：

 复制代码

```
1  /**
2   * 引用消解
3   * 遍历AST。如果发现函数调用和变量引用，就去找它的定义。
4   */
5  export class RefResolver extends AstVisitor{
6      symTable:SymTable;
7      constructor(symTable:SymTable){
8          super();
9          this.symTable = symTable;
10     }
11 }
```




```
12 //消解函数引用
13 visitFunctionCall(functionCall:FunctionCall):any{
14     let symbol = this.symTable.getSymbol(functionCall.name);
15     if (symbol != null && symbol.kind == SymKind.Function){
16         functionCall.decl = symbol.decl as FunctionDecl;
17     }
18     else{
19         if (functionCall.name != "println"){ //系统内置函数不用报错
20             console.log("Error: cannot find declaration of function " + fu
21         }
22     }
23 }
24
25 //消解变量引用
26 visitVariable(variable: Variable):any{
27     let symbol = this.symTable.getSymbol(variable.name);
28     if (symbol != null && symbol.kind == SymKind.Variable){
29         variable.decl = symbol.decl as VariableDecl;
30     }
31     else{
32         console.log("Error: cannot find declaration of variable " + variab
33     }
34 }
35 }
```

好了，现在我们通过新建符号表升级了我们的引用消解功能。有了符号表的支持，我们在程序中使用变量时，就可以直接从符号表里知道变量的类型，调用一个函数时，也能够直接从符号表里找到该函数的代码，也就是函数声明中的函数体。不过，还有一个语义分析功能也最好现在就实现一下，就是与类型处理有关的功能。

语义分析：类型处理

相比 JavaScript，TypeScript 的一个重要特性，就是可以清晰地指定类型，这能够在编译期进行类型检查，减少程序的错误。比如，在下面的程序中，myAge 是 number 类型的。这时候，如果你把一个字符串赋值给 myAge，TypeScript 编译器就会报错：

 复制代码

```
1 let myAge:number;
2 myAge = "Hello";
```

运行 TypeScript 编译器，报错信息如下：

```
example02.ts:5:1 - error TS2322: Type 'string' is not assignable to type 'number'.  
5 myAge = "Hello";  
   ~~~~~  
  
Found 1 error.
```

图3：TypeScript编译器的类型检查功能

那么，如何检查程序中的类型是否匹配呢？这需要用到一种叫做属性计算的技术。它其实就是给 AST 节点附加一些属性，比如类型等。然后通过一些 AST 节点的属性，去计算另一些节点的属性。

其实，对表达式求值的过程，就可以看做是属性计算的过程，这里的属性就是表达式的值。我们通过遍历 AST，可以通过叶子节点的值逐步计算更上层节点的值。

类似的技术还可以用于计算类型。比如，对于 `myAge = "Hello"` 这个表达式对应的 AST 节点，我们可以设置两个属性：一个属性是 `type_req`，也就是赋值操作中，左边的变量所需要的类型；另一个属性是 `type`，也就是 `=` 号右边的表达式实际的类型。

所谓类型检查，就是检查这两个类型是否匹配就可以了。其中，`type_req` 可以通过查符号表获得，也就是在声明 `myAge` 时所使用的类型。而 `=` 号右边表达式的 `type` 属性，可以像计算表达式的值一样，自底向上逐级计算出来。

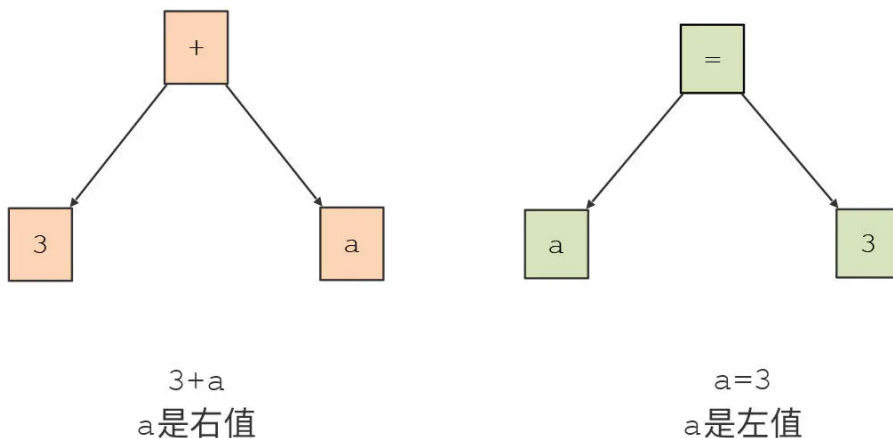
对于当前例子来说，我们一下子就能知道 “Hello” 字面量是字符串型的。如果是一个更复杂一点的表达式，比如 `"Hello" + 3 * 5`，它的类型就需要自底向上的逐级计算得到。这种自下而上逐级计算得到的属性，我们把它叫做综合属性（Synthesized Attribute）。

还有一个与综合属性相对应的概念，叫继承属性（Inherited Attribute），它是指从父节点或者兄弟节点计算出来的属性，这里我举一个例子来帮助你理解一下。

其实在解释器执行 `"a+3"` 和 `"a=3"` 这两个表达式的时候，对这两个变量 `a` 的操作是不一样的。对于 `"a+3"`，只需要取出 `a` 的值就行了。而对于 `"a=3"`，则需要给 `a` 赋一个新的值。`a` 如果在赋值符号的左边，我们就叫它左值，其他情况就叫右值。

为了让解释器顺利地运行，我们在遍历 AST 的时候，需要知道当前的这个变量是左值还是右值。所以，我就给表达式类型的 AST 节点添加了一个 `isLeftValue` 的属性。这个属性

呢，是一个典型的继承属性，因为它的值是通过上级节点计算得到的。当变量是赋值运算符的第一个子节点的时候，它是个左值。



现在再回到类型计算。知道了类型检查的思路，我们其实还可以再进一步，进行类型的推断。

在类型声明的语法规则中，我们会发现 `typeAnnotation` 是可选的。当你不显式规定类型的情况下，其实 TypeScript 可以根据变量初始化部分的类型，来进行类型推论的。比如下面的例子中，`myAge` 的类型可以被自动推断出来是 `number`，这样第二个赋值语句就会被报错。

```
1 let myAge = 18;  
2 myAge = "Hello";
```

[复制代码](#)

好了，现在我们已经做了引用消解和类型处理这两项关键的语义分析工作。不过，要保证一个程序正确，还要做很多语义分析工作。

更多的语义分析工作

语义分析的工作其实比较多和杂的。语言跟语言的差别，很多情况下也体现在语义方面。对于我们当前的语言功能，还需要去做的语义分析功能包括：

在赋值语句中，= 号左边必须是可以被赋值的表达式，我们把它叫做左值。变量就可以作为左值，而字面量就不可以；

字符串可以用 + 号跟数值进行连接运算，但不可以参与 - 号、* 号和 / 号的运算等等。

实际上，在 JavaScript 语言的标准（[ECMAScript 语言规格 2015 版](#)）中，大量的内容都是对语义的规定，比如下面的截图中，就是对 + 号运算符的语义规定。

12.7.3 The Addition operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

12.7.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be [GetValue](#)(*lref*).
3. [ReturnIfAbrupt](#)(*lval*).
4. Let *rref* be the result of evaluating *MultiplicativeExpression*.
5. Let *rval* be [GetValue](#)(*rref*).
6. [ReturnIfAbrupt](#)(*rval*).
7. Let *lprim* be [ToPrimitive](#)(*lval*).
8. [ReturnIfAbrupt](#)(*lprim*).
9. Let *rprim* be [ToPrimitive](#)(*rval*).
10. [ReturnIfAbrupt](#)(*rprim*).
11. If [Type](#)(*lprim*) is String or [Type](#)(*rprim*) is String, then
 - a. Let *lstr* be [ToString](#)(*lprim*).
 - b. [ReturnIfAbrupt](#)(*lstr*).
 - c. Let *rstr* be [ToString](#)(*rprim*).
 - d. [ReturnIfAbrupt](#)(*rstr*).
 - e. Return the String that is the result of concatenating *lstr* and *rstr*.
12. Let *lnum* be [ToNumber](#)(*lprim*).
13. [ReturnIfAbrupt](#)(*lnum*).
14. Let *rnum* be [ToNumber](#)(*rprim*).
15. [ReturnIfAbrupt](#)(*rnum*).
16. Return the result of applying the addition operation to *lnum* and *rnum*. See the Note below [12.7.5](#).

图4：JavaScript中加法运算的语义描述

不过，这些语义分析和处理工作通常并不复杂，大部分都可以通过遍历 AST 来实现。我也给出了参考实现，你可以查阅相关的代码。

做完语义分析工作以后，我们基本上就能够保证程序的正确性了。接下来，我们要对解释器做一下提升，让它也能够支持变量声明和赋值。

增强解释器的功能

让目前的解释器支持变量，其实比较简单，我们只需要有一个机制能够保存变量的值就可以了。

在我的参考实现里，我用了一个 map 的数据结构来保存每个变量的值。解释器可以从 map 里查找一个变量的值，也可以更新某个变量的值，具体你可以看下面这些代码：

[复制代码](#)

```
1  /**
2   * 遍历AST，并执行。
3   */
4  class Interpreter extends AstVisitor{
5      //存储变量值的区域
6      values:Map<string, any> = new Map();
7
8      /**
9       * 变量声明
10      * 如果存在变量初始化部分，要存下变量值。
11      */
12      visitVariableDecl(variableDecl:VariableDecl):any{
13          if(variableDecl.init != null){
14              let v = this.visit(variableDecl.init);
15              if (this.isLeftValue(v)){
16                  v = this.getVariableValue((v as LeftValue).variable.name);
17              }
18              this.setVariableValue(variableDecl.name, v);
19              return v;
20          }
21      }
22
23      /**
24       * 获取变量的值。
25       * 这里给出的是左值。左值既可以赋值（写），又可以获取当前值（读）。
26       * @param v
27       */
28      visitVariable(v:Variable):any{
29          return new LeftValue(v);
30      }
31      //获取变量值
32      private getVariableValue(varName:string):any{
33          return this.values.get(varName);
34      }
35      //设置变量值
36      private setVariableValue(varName:string, value:any):any{
37          return this.values.set(varName, value);
38      }
39      //检查是否是左值
40      private isLeftValue(v:any):boolean{
41          return typeof (v as LeftValue).variable == 'object';
```

```
42     }  
43     //省略其他部分...  
44 }
```

这样，通过引入一个简单的 map，我们的程序在每一次引用变量时，都能获得正确的值。当然，这个机制目前是高度简化的。我们在后面会持续演化，引入栈帧等进一步的概念。

好了，现在我们的功能又得到了提升，你可以编写几个程序试一下它的能力。我编写了下面的一个程序，你可以试试看它的运行结果是什么。

```
1  /**  
2   * 示例程序，由play.js解析并执行。  
3   * 特性：对变量的支持。  
4   */  
5  
6  //那年才18  
7  let myAge:number = 18;  
8  
9  //转眼10年过去  
10 myAge = myAge + 10;  
11  
12 println("myAge is: ");  
13 println(myAge);
```

[复制代码](#)

课程小结

今天这节课，我们为了让程序支持变量，分别升级了语法分析、语义分析和解释器。通过这种迭代开发方式，我们可以让语言的功能一点点的变强。

在今天的旅程中，我希望你能记住下面这几个关键点：

首先，我给你演示了如何增加新的语法规则，在这里你要继续熟悉如何用 EBNF 来书写语法规则。然后，我也给你说明了如何通过计算 First 和 Follow 集合来使用 LL 算法，你要注意 EBNF 里所有带? 号或者 * 号、+ 号的语法成分，都需要计算它的 Follow 集合，以便判断到底是解析这个语法成分，还是跳过去。通过这样一次次的练习，你会对 LL 算法越来越熟练。

在语义分析部分，我们引入了符号表这个重要的数据结构，并通过它简化了引用消解机制。我们在后面的课程还会深化对符号表的理解。

另外，与类型有关的处理也很重要。你可以通过属性计算的方式，实现类型的检查和自动推断。你还要记住，语义分析工作的内容是很多的，建立符号表、引用消解和类型检查是其中的重点工作，但还有很多其他语义分析工作。

最后，解释器如果要支持变量，就必须能够保存变量在运行时的值。我们是用了一个 map 数据结构来保存变量的值，在后面的课程里，我们还会升级这个机制。

不过，我们现在只能在顶层代码里使用变量。如果在函数里使用，还有一些问题，比如作用域和局部变量的机制、函数传参的机制、返回值的机制等等。我们会在下一节去进一步升级我们的语言，让它的功能更强大。

思考题

今天的思考题，我想问下你，在变量声明的语法规则中，你能否计算出 typeAnnotation 的 Follow 集合都包含哪些元素？欢迎在留言区给我留言。

感谢你和我一起學習，如果你觉得这节课讲得还不错，也欢迎分享给更多感兴趣的朋友。我是宫文学，我们下节课见。

资源链接

[🔗 这节课的示例代码在这里！](#)

分享给需要的人，Ta 订阅后你可得 **20 元现金奖励**

👍 赞 1 💡 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 支持表达式：解析表达式和解析语句有什么不同？

精选留言 (6)

写留言



leaf

2021-08-17

老师目前的语法规则我仔细看后觉得好像还不是很严谨

```
* prog = statementList? EOF;  
* statementList = (variableDecl | functionDecl | expressionStatement)+ ;  
* variableDecl : 'let' Identifier typeAnnotation ? ('=' singleExpression) ';'  
* typeAnnotation : ':' typeName;...
```

展开 ∨

1

1



R

2021-08-19

宫老师，04的代码是不是还不完整？我用go调试了好久函数总是没消解引用，调了ts的好像也是没有



leaf

2021-08-17

RefResolver.visitFunctionCall是否应该加一段对functionCall.parameters的visit, 如下:

```
visitFunctionCall(functionCall:FunctionCall):any{  
    let symbol = this.symTable.getSymbol(functionCall.name);  
    if (symbol != null && symbol.kind == SymKind.Function){  
        functionCall.decl = symbol.decl as FunctionDecl;...
```

展开 ∨



leaf

2021-08-17

老师, 看了各个Visitor, 有一个比较大的困惑, 就是基类应该如何写, 派生类应该override哪些方法, 分别有什么原则吗?

展开 ∨



R

2021-08-17

关于 typeAnnotation 的 follow 集合，根据文中的说法，冒号 ':' 是 First 集，那么 Follo

w 集就是各个类型关键字，这些关键字有：

- : boolean 布尔值
- : number 数字
- : string 字符串...

展开 ▾



leaf
2021-08-17

老师这节课的代码好像没有解决赋值语句的右结合问题吧, 比如对如下语句

```
a = b = c;
```

分析结果如下:

```
ExpressionStatement
  Binary:=...
```

展开 ▾

