12 语义分析（下）：如何做上下文相关情况的处理?

我们知道，词法分析和语法分析阶段，进行的处理都是上下文无关的。可仅凭上下文无关的处理，是不能完成一门强大的语言的。比如先声明变量，再用变量，这是典型的上下文相关的情况，我们肯定不能用上下文无关文法表达这种情况，所以语法分析阶段处理不了这个问题，只能在语义分析阶段处理。**语义分析的本质，就是针对上下文相关的情况做处理。**

我们之前讲到的作用域，是一种上下文相关的情况，因为如果作用域不同，能使用的变量也是不同的。类型系统也是一种上下文相关的情况，类型推导和类型检查都要基于上下文中相关的AST节点。

本节课，我们再讲两个这样的场景：**引用的消解、左值和右值，**然后再介绍上下文相关情况分析的一种方法：**属性计算。**这样，你会把语义分析就是上下文处理的本质掌握得更清楚，并掌握属性计算这个强大的方法。

我们先来说说引用的消解这个场景。

语义分析场景：引用的消解

在程序里使用变量、函数、类等符号时，我们需要知道它们指的是谁，要能对应到定义它们的地方。下面的例子中，当使用变量a时，我们需要知道它是全局变量a，还是fun()函数中的本地变量a。因为不同作用域里可能有相同名称的变量，所以必须找到正确的那个。这个过程，可以叫引用消解。

```
/*
scope.c
测试作用域
 */
#include <stdio.h>

int a = 1;

void fun()
{
    a = 2;      //这是指全局变量a
    int a = 3; //声明一个本地变量
    int b = a; //这个a指的是本地变量
```

```
    printf("in func: a=%d b=%d \n", a, b);
}
```

在集成开发环境中，当我们点击一个变量、函数或类，可以跳到定义它的地方。另一方面，当我们重构一个变量名称、方法名称或类名称的时候，所有引用它的地方都会同步修改。这是因为IDE分析了符号之间的交叉引用关系。

函数的引用消解比变量的引用消解还要更复杂一些。

它不仅要比对函数名称，还要比较参数和返回值（可以叫函数原型，又或者叫函数的类型）。我们在把函数提升为一等公民的时候，提到函数类型（FunctionType）的概念。两个函数的类型相同，需要返回值、参数个数、每个参数的类型都能匹配得上才行。

在面向对象编程语言中，函数引用的消解也很复杂。

当一个参数需要一个对象的时候，程序中提供其子类的一个实例也是可以的，也就是子类可以用在所有需要父类的地方，例如下面的代码：

```
class MyClass1{}      //父类
class MyClass2 extends MyClass1{}  //子类

MyClass1 obj1;
MyClass2 obj2;

function fun(MyClass1 obj){}       //参数需要父类的实例

fun(obj2);   //提供子类的实例
```

在C++语言中，引用的消解还要更加复杂。

它还要考虑某个实参是否能够被自动转换成形参所要求的类型，比如在一个需要double类型的地方，你给它传一个int也是可以的。

命名空间也是做引用消解的时候需要考虑的因素。

像Java、C++都支持命名空间。如果在代码前头引入了某个命名空间，我们就可以直接引用里面的符号，否则需要冠以命名空间。例如：

```
play.PlayScriptCompiler.Compile()   //Java语言
play::PlayScriptCompiler.Compile()  //C++语言
```

而做引用消解可能会产生几个结果：

- 解析出了准确的引用关系。

- 重复定义（在声明新的符号的时候，发现这个符号已经被定义过了）。
- 引用失败（找不到某个符号的定义）。
- 如果两个不同的命名空间中都有相同名称的符号，编程者需要明确指定。

在playscript中，引用消解的结果被存到了AnnotatedTree.java类中的symbolOfNode属性中去了，从它可以查到某个AST节点引用的到底是哪个变量或函数，从而在运行期正确的执行，你可以看一下代码，了解引用消解和使用的过程。

了解完引用的消解之后，接下来，我们再讲一个很有意思的场景：左值和右值。

语义分析场景：左值和右值

在开发编译器或解释器的过程中，你一定会遇到左值和右值的问题。比如，在playscript的ASTEvaluate.java中，我们在visitPrimary节点可以对变量求值。如果是下面语句中的a，没有问题，把a变量的值取出来就好了：

```
a + 3;
```

可是，如果针对的是赋值语句，a在等号的左边，怎么对a求值呢？

```
a = 3;
```

假设a变量原来的值是4，如果还是把它的值取出来，那么成了3=4，这就变得没有意义了。所以，不能把a的值取出来，而应该取出a的地址，或者说a的引用，然后用赋值操作把3这个值写到a的内存地址。**这时，我们说取出来的是a的左值（L-value）。**

左值最早是在C语言中提出的，通常出现在表达式的左边，如赋值语句的左边。左值取的是变量的地址（或者说变量的引用），获得地址以后，我们就可以把新值写进去了。

与左值相对应的就是右值（R-value），右值就是我们通常所说的值，不是地址。

在上面这两种情况下，变量a在AST中都是对应同一个节点，也就是primary节点。那这个节点求值时是该返回左值还是右值呢？这要借助上下文来分析和处理。如果这个primary节点存在于下面这几种情况中，那就需要取左值：

- 赋值表达式的左边；
- 带有初始化的变量声明语句中的变量；
- 当给函数形参赋值的时候；
- 一元操作符： ++和--。

- 其他需要改变变量内容的操作。

在讨论primary节点在哪种情况下取左值时，我们可以引出另一个问题：**不是所有的表达式，都能生成一个合格的左值。**也就是说，出现在赋值语句左边的，必须是能够获得左值的表达式。比如一个变量是可以的，一个类的属性也是可以的。但如果是一个常量，或者2+3这样的表达式在赋值符号的左边，那就不行。所以，判断表达式能否生成一个合格的左值也是语义检查的一项工作。

借上节课讲过的S属性和I属性的概念，我们把刚才说的两个情况总结成primay节点的两个属性，你可以判断一下，这两个属性是S属性还是I属性？

- 属性1：某primary节点求值时，是否应该求左值？
- 属性2：某primary节点求值时，能否求出左值？

你可能发现了，这跟我们类型检查有点儿相似，一个是I属性，一个是S属性，两个一比对，就能检查求左值的表达式是否合法。从这儿我们也能看出，处理上下文相关的情况，经常用属性计算的方法。接下来，我们就谈谈如何做属性计算。

如何做属性计算

属性计算是做上下文分析，或者说语义分析的一种算法。按照属性计算的视角，我们之前所处理的各种语义分析问题，都可以看做是对AST节点的某个属性进行计算。比如，针对求左值场景中的primary节点，它需要计算的属性包括：

- 它的变量定义是哪个（这就引用到定义该变量的Symbol）。
- 它的类型是什么？
- 它的作用域是什么？
- 这个节点求值时，是否该返回左值？能否正确地返回一个左值？
- 它的值是什么？

从属性计算的角度看，对表达式求值，或运行脚本，只是去计算AST节点的Value属性，Value这个属性能够计算，其他属性当然也能计算。

属性计算需要用到属性文法。在词法、语法分析阶段，我们分别学习了正则文法和上下文无关文法，在语义分析阶段我们要了解的是**属性文法（Attribute Grammar）**。

属性文法的主要思路是计算机科学的重要开拓者，高德纳（Donald Knuth）在《The Genesis of Attribute Grammers》中提出的。它是在上下文无关文法的基础上做了一些增强，使之能够计算属性值。下面是上下文无关文法表达加法和乘法运算的例子：

```
add → add + mul
add → mul
mul → mul * primary
mul → primary
primary → "(" add ")"
primary → integer
```

然后看一看对value属性进行计算的属性文法：

```
add1 → add1 + mul [ add1.value = add2.value + mul.value ]
add → mul [ add.value = mul.value ]
mul1 → mul2 * primary [ mul1.value = mul2.value * primary.value ]
mul → primary [ mul.value = primary.value ]
primary → "(" add ")" [ primary.value =  add.value ]
primary → integer [ primary.value = strToInt(integer.str) ]
```

利用属性文法，我们可以定义规则，然后用工具自动实现对属性的计算。有同学曾经问：“我们解析表达式2+3的时候，得到一个AST，但我怎么知道它运算的时候是做加法呢？”

因为我们可以在语法规则的基础上制定属性文法，在解析语法的过程中或者形成AST之后，我们就可以根据属性文法的规则做属性计算。比如在Antlr中，你可以在语法规则文件中插入一些代码，在语法分析的过程中执行你的代码，完成一些必要的计算。

总结一下属性计算的特点：它会基于语法规则，增加一些与语义处理有关的规则。

所以，我们也把这种语义规则的定义叫做语法制导的定义（Syntax directed definition，SDD），如果变成计算动作，就叫做语法制导的翻译（Syntax directed translation，SDT）。

属性计算，可以伴随着语法分析的过程一起进行，也可以在做完语法分析以后再进行。这两个阶段不一定完全切分开。甚至，我们有时候会在语法分析的时候做一些属性计算，然后把计算结果反馈回语法分析的逻辑，帮助语法分析更好地执行（这是在工程实践中会运用到的一个技巧，我这里稍微做了一个延展，帮大家开阔一下思路，免得把知识学得太固化了）。

那么，在解析语法的时候，如何同时做属性计算呢？我们知道，解析语法的过程，是逐步建立AST的过程。在这个过程中，计算某个节点的属性所依赖的其他节点可能被创建出来了。比如在递归下降算法中，当某个节点建立完毕以后，它的所有子节点一定也建立完毕了，所以S属性就可以计算出来了。同时，因为语法解析是从左向右进行的，它左边的兄弟节点也都建立起来了。

如果某个属性的计算，除了可能依赖子节点以外，只依赖左边的兄弟节点，不依赖右边的，这种属性就叫做L属性。它比S属性的范围更大一些，包含了部分的I属性。由于我们常用的语法分析的算法都是从左向右进行的，所以就很适合一边解析语法，一边计算L属性。

比如，C语言和Java语言进行类型分析，都可以用L属性的计算来实现。因为这两门语言的类型要么是从下往上综合出来的，属于S属性。要么是在做变量声明的时候，由声明中的变量类型确定的，类型节点在变量的左边。

```
2+3;       //表达式类型是整型
float a;   //a的类型是浮点型
```

那问题来了，Go语言的类型声明是放在变量后面的，这意味着类型节点一定是在右边的，那就不符合L属性文法了：

```
var a int = 10
```

没关系，我们没必要在语法分析阶段把属性全都计算出来，等到语法分析完毕后，再对AST遍历一下就好了。这时所有节点都有了，计算属性也就不是难事了。

在我们的playscript语言里，就采取了这种策略，实际上，为了让算法更清晰，我把语义分析过程拆成了好几个任务，对AST做了多次遍历。

第1遍：类型和作用域解析（TypeAndScopeScanner.java）。

把自定义类、函数和和作用域的树都分析出来。这么做的好处是，你可以使用在前，声明在后。比如你声明一个Mammal对象，而Mammal类的定义是在后面才出现的；在定义一个类的时候，对于类的成员也会出现使用在声明之前的情况，把类型解析先扫描一遍，就能方便地支持这个特性。

在写属性计算的算法时，计算的顺序可能是个最重要的问题。因为某属性的计算可能要依赖别的节点的属性先计算完。我们讨论的S属性、I属性和L属性，都是在考虑计算顺序。像使用在前，声明在后这种情况，就更要特殊处理了。

第2遍：类型的消解（TypeResolver.java）。

把所有出现引用到类型的地方，都消解掉，比如变量声明、函数参数声明、类的继承等等。做完消解以后，我们针对Mammal m;这样语句，就明确的知道了m的类型。这实际上是对I属性的类型的计算。

第3遍：引用的消解和S属性的类型的推导（RefResolver.java）。

这个时候，我们对所有的变量、函数调用，都会跟它的定义关联起来，并且完成了所有的类型计算。

第4遍：做类型检查（TypeChecker.java）。

比如当赋值语句左右两边的类型不兼容的时候，就可以报错。

第5遍：做一些语义合法性的检查（SematicValidator.java）。

比如break只能出现在循环语句中，如果某个函数声明了返回值，就一定要有return语句，等等。

语义分析的结果保存在AnnotatedTree.java类里，意思是被标注了属性的语法树。注意，这些属性在数据结构上，并不一定是AST节点的属性，我们可以借助Map等数据结构存储，只是在概念上，这些属性还是标注在树节点上的。

AnnotatedTree类的结构如下：

```
public class AnnotatedTree {
    // AST
    protected ParseTree ast = null;

    // 解析出来的所有类型，包括类和函数
    protected List<Type> types = new LinkedList<Type>();

    // AST节点对应的Symbol
    protected Map<ParserRuleContext, Symbol> symbolOfNode = new HashMap<ParserRuleConte

    // AST节点对应的Scope，如for、函数调用会启动新的Scope
    protected Map<ParserRuleContext, Scope> node2Scope = new HashMap<ParserRuleContext,

    // 每个节点推断出来的类型
    protected Map<ParserRuleContext, Type> typeOfNode = new HashMap<ParserRuleContext,

    // 命名空间，作用域的根节点
    NameSpace nameSpace = null;

    ...
}
```

我建议你看看这些语义分析的代码，了解一下如何保证语义分析的全面性。

课程小结

本节课我带你继续了解了语义分析的相关知识：

- 语义分析的本质是对上下文相关情况的处理，能做词法分析和语法分析所做不到的事情。
- 了解引用消解，左值和右值的场景，可以增加对语义分析的直观理解。
- 掌握属性计算和属性文法，可以使我们用更加形式化、更清晰的算法来完成语义分析的任务。

在我看来，语义分析这个阶段十分重要。因为词法和语法都有很固定的套路，甚至都可以工具化的实现。但语言设计的核心在于语义，特别是要让语义适合所解决的问题。比如SQL语言针对的是数据库的操作，那就去充分满足这个目标就好了。我们在前端技术的应用篇中，也会复盘讨论这个问题，不断实现认知的迭代升级。

如果想做一个自己领域的DSL，学习了这几讲语义分析的内容之后，你会更好地做语义特性的设计与取舍，也会对如何完成语义分析有清晰的思路。

一课一思

基于你熟悉的语言，来说说你觉得在语义分析阶段还有哪些上下文处理工作要做？需要计算出哪些属性？它们是I属性还是S属性？起到什么作用？这个思考练习很有意思，欢迎在留言区分享你的发现。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的朋友。

本节课相关的示例代码放在文末，供你参考。

- playscript-java（项目目录）： 码云 GitHub
- PlayScript.g4（语法规则）： 码云 GitHub
- TypeAndScopeScanner.java（类型和作用域扫描）： 码云 GitHub
- TypeResolver.java（消解变量声明中引用的类型）： 码云 GitHub
- RefResolver.java（变量和函数应用的消解，及S属性的类型推断）： 码云 GitHub
- TypeChecker.java（类型检查）： 码云 GitHub

上一页 下一页

© 2019 - 2023 Liangliang Lee. Powered by gin and hexo-theme-book.