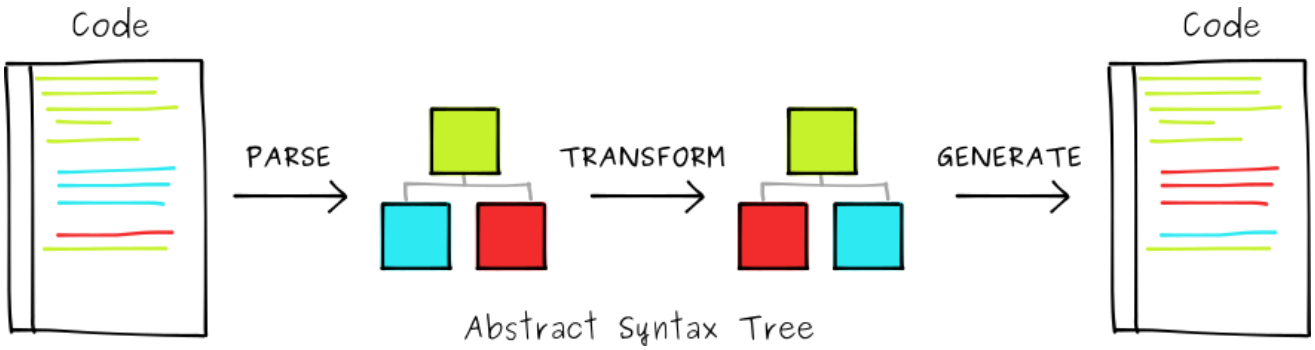
抽象语法树(AST)及Vistor设计模式

刚才赵一杰同学先讲了一些面向对象语言的语法特性，下面我来介绍一下针对面向对象语言的抽象语法树的设计思想以及一个简单的设计模式—Vistor模式。

1. 什么是抽象语法树?

抽象语法树相信大家已经很熟悉了，它是在词法分析和语法分析阶段生成的一种高级的中间表示，树上的每个结点都表示了源代码中的一种结构，那为什么它又是抽象的呢？是因为抽象语法树并不会表示出真实语法出现的每一个细节，比如嵌套的括号之类的，它只显示关键信息，能够很好地体现出源程序的语法结构。

> 在计算机科学中，抽象语法树（abstract syntax tree或者缩写为AST），或者语法树（syntax tree），是源代码的抽象语法结构的树状表现形式，这里特指编程语言的源代码。



大家都知道中间代码的表示有很多种，那么为什么本次实验采用了AST的形式呢?

2. 为什么是抽象语法树?

我们首先知道，当编译器在进行语法分析时，它是在相应语言的语法规则指导下进行的，那这个规则我们是怎么表示的呢？一般是上下文无关文法和BNF范式。上下文无关文法大家也已经很熟悉了，我们这学期学到的有这么几类**LL(1)，LR(0)，LR(1)， LR(k)，LALR(1)**等。

每一种文法还都有不同的要求，比如LL(1)要求文法无二义性和不存在左递归。当把一个文法改为LL(1)文法时，就需要引入一些额外的文法符号与产生式去消除左递归。

```
//含有左递归的文法
S -> Sa
S -> b

//消除左递归后的文法
S -> bS'
S'-> aS'|$
```

现在假如我们要写一个编译器，并且我们选择了LL(1)文法去表示这个语言的语法规则：

但是随着语言中加入的特性越来越多，用LL(1)文法描述时，感觉限制很大，并且编写文法时很吃力，这个时候我们采用更好的LR(1)文法。

那么我们需要先把编译器前端改生成LR(1)语法树，但在这个时候，你会发现以前编译器的后端是专门针对LL(1)语法树进行处理的，所以你还得修改后端的代码，简直是噩梦！

- 不依赖于具体的文法

  无论是LL(1)文法，还是LR(1)文法，都要求在语法分析时候，构造出相同的语法树，这样可以给编译器后端提供了清晰，统一的接口。即使是前端采用了不同的文法，都只需要改变前端代码，而不用连累到后端。即减少了工作量，也提高的编译器的可维护性。

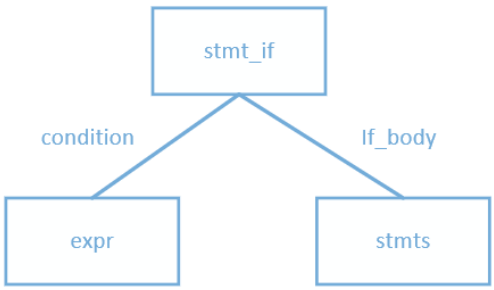
- 不依赖于语言的细节

  相信大家都听过gcc，gcc不止可以编译器c语言，还能用于c++，java，object-C等语言程序，它就是对不同的语言进行词法，语法分析和语义分析后，产生一个统一结构的抽象语法树，交给后端处理。

  比如对于下面这两种语言的条件语句，都可以构造出相同的语法树：

```
if(condition){
    //TODO
}
```

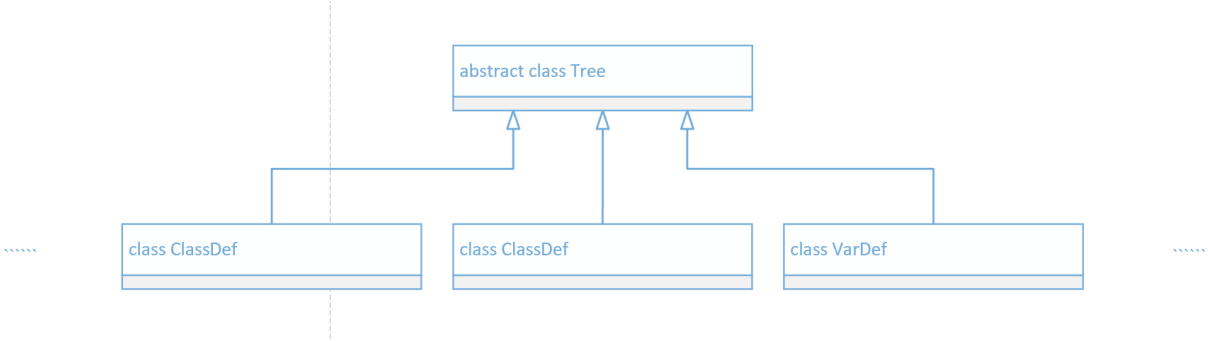
```
If condition then
    !TODO
end if
```



3. 怎么设计抽象语法树?

抽象语法树是编译器前后端的桥梁，它在设计的时候就必须考虑到如何能让其它模块更高效地使用这棵树，在整个编译器中需要对它进行多遍扫描，有的是获取信息，有的是在上面填充信息。

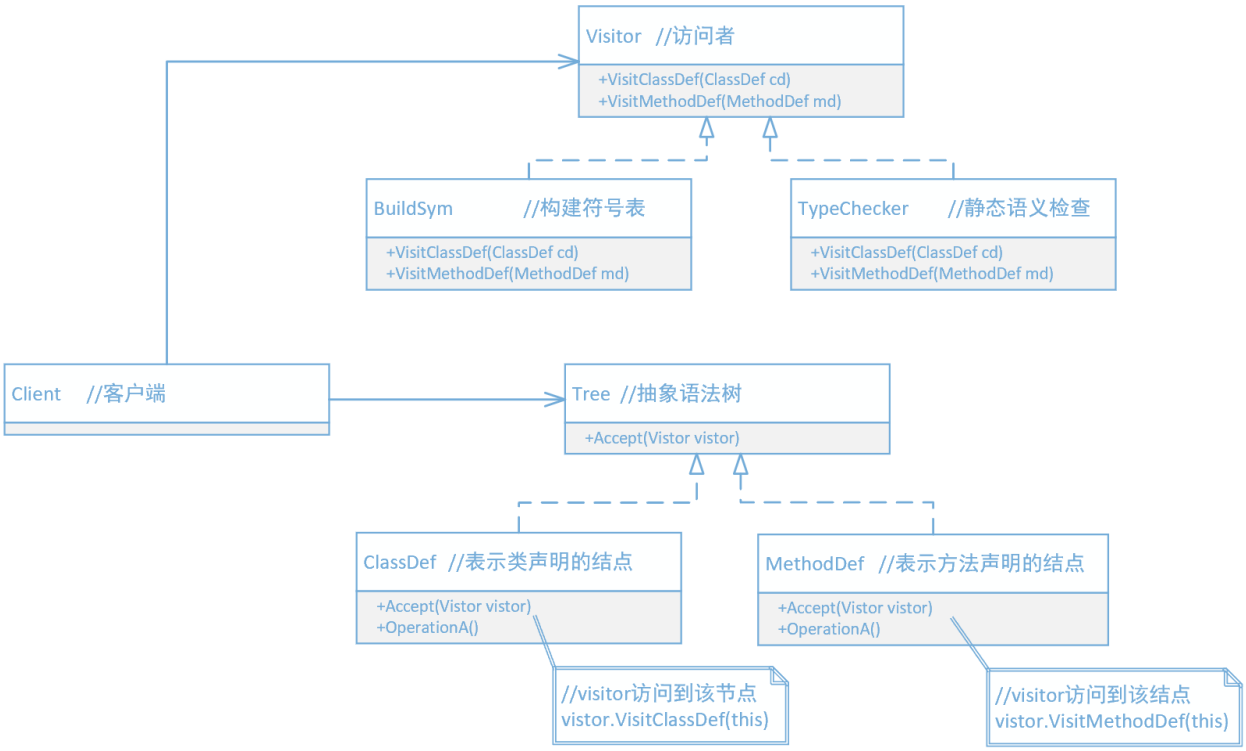
下面是我们常规的建树模式，树上的每个结点都对应着语法规则，并且都继承了抽象类Tree



就这种多遍扫描的方式来说，虽然每个结点都是不同的，但是在扫描过程中，每个结点都会经过同类的处理，只是处理到不同结点时有不同的行为，如果我们对于每个结点上的不同遍历方式都专门写一个方法，代码就会显得很冗余，而且代码量也很大，并且如果出现新的遍历方式，其实对于这种问题，有一种设计模式可以解决——访问者模式(Visitor)。

下面简单介绍一下这种设计模式：

> **访问者（Visitor）模式**：提供一个作用于某对象结构中的各元素的操作表示，它使得可以在不改变各元素的类的前提下定义作用于这些元素的新操作。访问者模式是一种对象行为型模式。



- Visitor（抽象访问者）：声明一个访问操作，从这个操作的名称或参数类型可以清楚知道需要访问的具体元素的类型，具体访问者则需要实现这些操作方法，定义对这些元素的访问操作。
- BuildSym/TypeChecker（具体访问者）：具体访问者实现了抽象访问者声明的方法，每一个操作作用于访问对象结构中一种类型的元素。
- Tree（抽象元素）：是一个抽象类，定义一个Accept方法，该方法以一个抽象访问者（Visitor）作为参数。
- ClassDef/MethodDef（具体元素）：具体元素实现了Accept方法，在Accept方法中调用访问者的访问方法以便完成一个元素的操作。

大家仔细阅读过代码之后会发现，每个结点大致都是这样的结构，完全是按照上述的设计模式设计的：

```
    public static class XXX extends Tree {
        //一些属性

        //构造函数
        public XXX() {
            //TODO
        }
        //根据不同的访问者调用
        @Override
        public void accept(Visitor v) {
            v.visitXXX(this);
        }

        //打印自己以及孩子结点的信息（P302 以缩进格式体现父子关系）
        @Override
        public void printTo(IndentPrintWriter pw) {
        }
    }
```