软件分析

程序设计语言基础知识

熊英飞
北京大学
2021

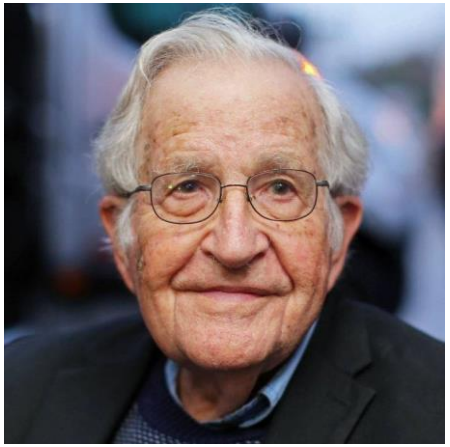
程序设计语言的两大要素

- 语法(Syntax)——程序的静态书写方式
  - 如：可以写a=a+2但不能写2 a =+ 2
- 语义(Semantics)——程序的动态执行行为
  - 如：a=a+2在运行时会把a的值加上2并赋值给a

- 也是自然语言的要素

如何描述语法?

- 诺姆·乔姆斯基
  - 现代语言学的奠基人
  - 提出了产生式文法、乔姆斯基分类等重要概念



文法

- 一个文法包括:
  - 一组非终结符，用大写字母表示，如S, A, B
  - 一组终结符，用小写字母表示，如a, b
  - 一个非终结符作为开始符号，用字母S表示
  - 一组推导规则，表示把一个字符串中的左边部分替换成右边部分，如
    - S->AB
    - aA->aaAb
    - A→ϵ （ϵ表示空串）

文法生成的例子

- G2 = ({S, A}, {a, b}, S, {S → aAb, aA → aaAb, A → ε })
  - S ⇒ aAb　　应用S → aAb
    ⇒ aaAbb　　应用aA → aAb
    ⇒ aaaAbbb　　应用aA → aaAb
    ⇒ aaabbb　　应用A → ε
- 所有从S开始，应用任意多次推导规则可以生成的所有终结符序列构成了文法对应语言
  - 即一个编程语言的语法所允许的所有程序

文法和自动机

- 给定文法，能否构造一个程序，判断所给字符串是否符合文法要求?

- 自动机：根据输入序列改变内部状态的机器
  - 状态转移函数：$\delta: Q \times \Sigma \to Q$
    - Q是（可能无限的）状态集合，包括起始态和终止态
    - Σ是输入字符集合

- 文法被自动机识别：
  - 字符串Str符合文法 ⇔自动机在输入Str上达到终止态

文法的类型—乔姆斯基分类

文法类型	语言类型	自动机
任意文法	递归可枚举语言	图灵机
上下文有关文法	上下文有关语言	线性有界自动机
上下文无关文法	上下文无关语言	下推自动机
正则文法	正则语言	有限状态自动机

- 正则文法：形如$A \to a$或者$A \to aB$或者$S \to \epsilon$
- 上下文无关文法：形如$A \to \gamma$
- 上下文有关文法：形如$\alpha A \beta \to \alpha \gamma \beta$
  - A, B为任意非终结符，a为任意终结符
  - α, β, γ是任意终结符和非终结符组成的字符串

程序设计语言常用文法

- 现代程序设计语言通常采用正则文法描述程序基本单元，称为token
  - Number := 0|[1-9][0-9]*
  - Iden := [a-zA-Z_][0-9a-zA-Z_]*
- 再将token作为终结符，采用上下文无关文法描述整个语言
  - Expr -> Expr + Expr | Expr – Expr | Number | Iden

如何描述语义?

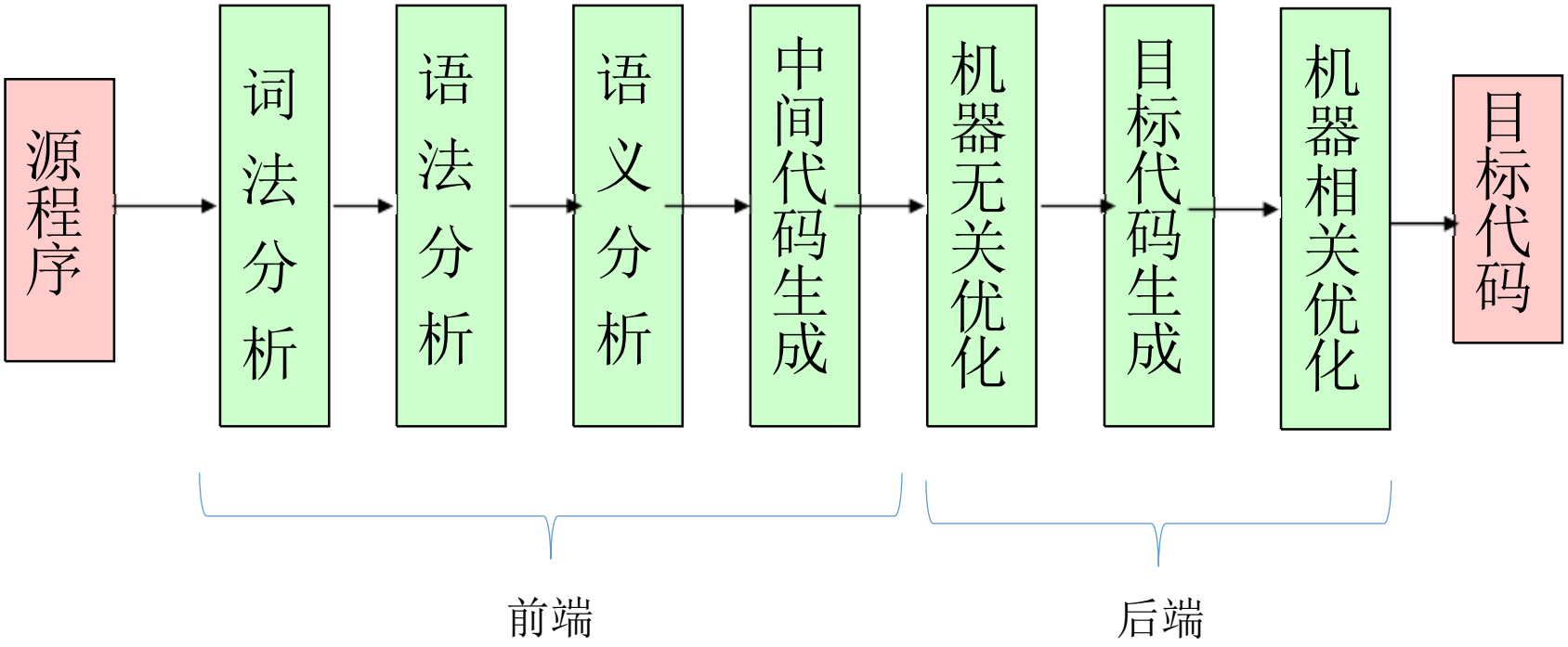
- 操作语义
  - 将程序元素解释为计算步骤
- 指称语义
  - 将程序元素解释为数学中严格定义的对象，通常为函数
- 操作语义 vs 指称语义
  - 在现代程序语义学的教材中，二者通常是等价的，只是惯用符号不同
  - 操作语义常采用逻辑推导规则描述，指称语义常采用集合定义描述
- 公理语义
  - 将程序元素解释为前条件和后条件，并可用霍尔逻辑推导

编译与程序分析

- 在保证语义不变的情况下，将程序从一种语法变换到另一种语法的技术
  - 通常用于将高级语言的程序变换到低级语言
- 程序分析：输入程序语法，回答关于程序语义的问题
- 程序分析和编译都依赖程序语法的解析
- 编译器依赖程序分析检查和推导类型、查找常见错误和保证优化的安全性

一次典型的编译过程

- 编译由一系列的程序变换构成



编译示例


```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n", a);
}
```

词法分析

基于正则文法识别出token序列

<保留字，void>
<标识符，cal>
<分隔符，(>
<保留字，float>
……
<标识符，a>
<算符 ，= > <标识符，b> < 算符，+ >
<标识符，c> <算符，* > <常数 ，60>
……
<字符串， “%f\n”>
<分隔符，， >
<标识符，a>
……

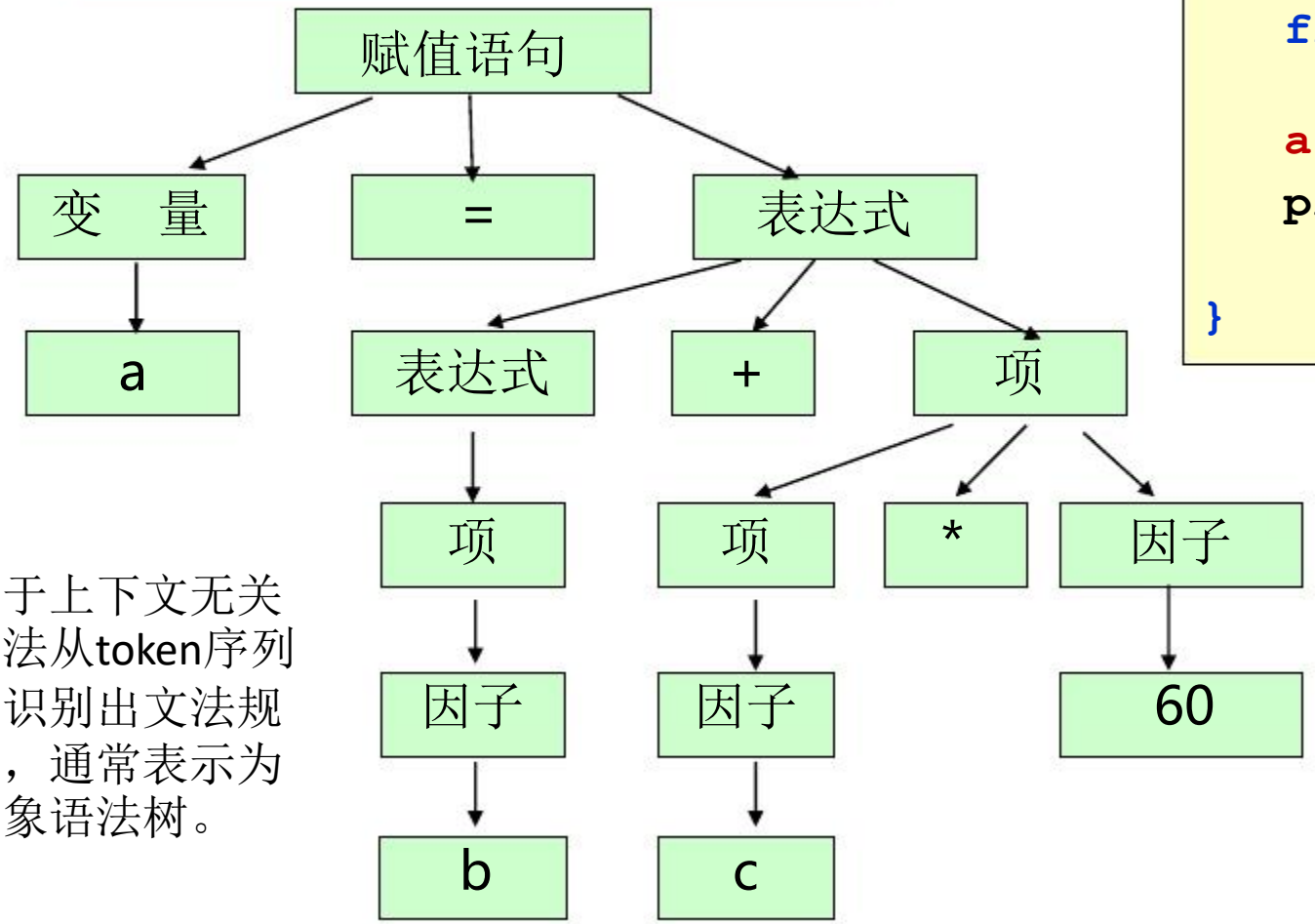
```
void cal(float b,
         float c){
   float a;

   a = b+c*60;
   printf("%f\n",
          a);
}
```

语法分析

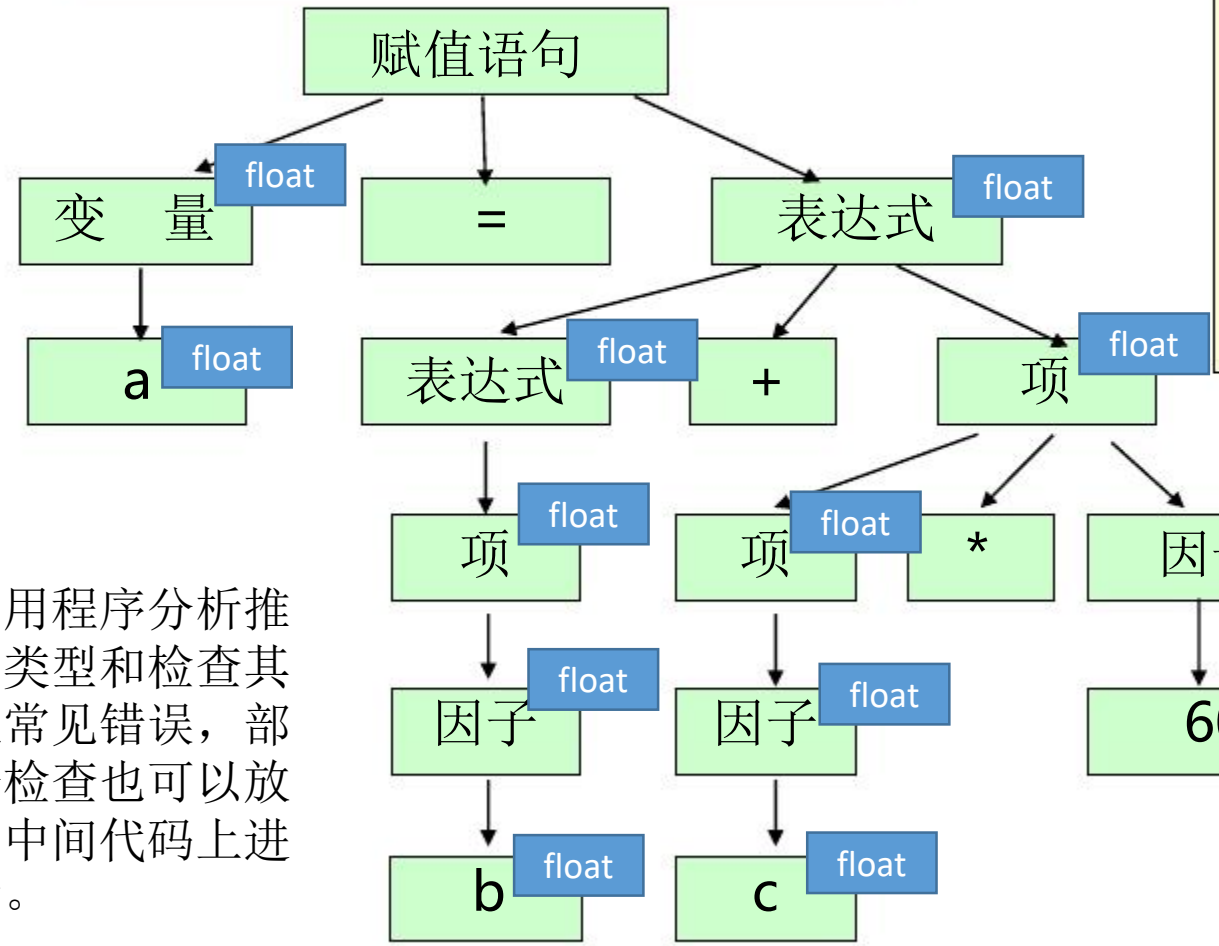
```
void cal(float b,
         float c){
    float a;

    a = b+c*60;
    printf("%f\n",
           a);
}
```



基于上下文无关文法从token序列中识别出文法规则，通常表示为抽象语法树。

语义分析




```
void cal(float b,
         float c){
    float a;

    a = b+c*60;
    printf("%f\n",
           a);
}
```

利用程序分析推导类型和检查其他常见错误，部分检查也可以放到中间代码上进行。

中间代码生成

- 中间代码
  - 独立于机器
  - 通常易于分析
  - 易于翻译成汇编代码
- 不同编译器的中间代码差别很大，没有通用标准
- 常见中间代码
  - Java Bytecode
  - LLVM IR

```
@.str = private constant [13 x i8] c"Hello World!\00", align 1 ;

define i32 @main() ssp {
entry:
  %retval = alloca i32
  %0 = alloca i32
  %"alloca point" = bitcast i32 0 to i32
  %1 = call i32 @puts(i8* getelementptr inbounds ([13 x i8]*
@.str, i64 0, i64 0))
  store i32 0, i32* %0, align 4
  %2 = load i32* %0, align 4
  store i32 %2, i32* %retval, align 4
  br label %return
return:
  %retval1 = load i32* %retval
  ret i32 %retval1
}

declare i32 @puts(i8*)
```

LLVM编译器的中间代码

机器无关优化

- 常量传播
- 冗余消除
- 展开函数调用
- 将if替换成条件移动

- 部分高级语言相关优化也可以放到抽象语法树上进行。

目标代码生成

- 将中间代码转换成目标机器的代码

机器相关优化

- 利用指令级并行的优化
- 利用SSE指令的优化

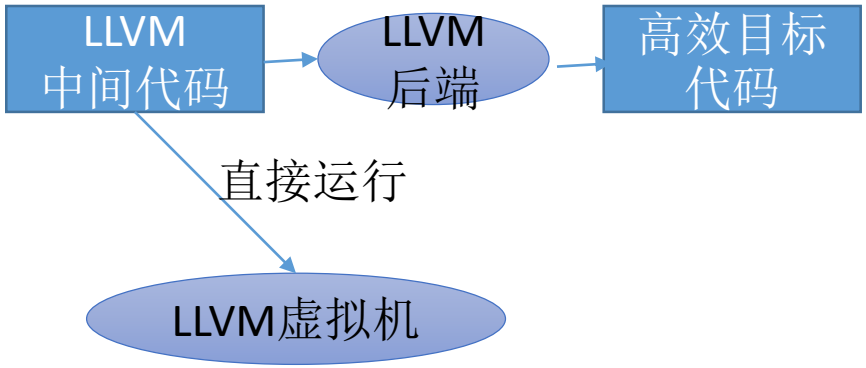
编译器 vs 解释器 vs JIT编译虚拟机

- 编译器：一次把源程序翻译成机器代码
- 解释器：每次读取一条语句
- JIT编译虚拟机：每次加载一个模块，或者运行一个函数之前一次性编译

- 编译器、解释器、JIT编译虚拟机，哪个速度最快?

LLVM

- LLVM同时支持编译和JIT编译虚拟机



- JIT编译虚拟机的运行速度有时能比编译快10倍以上!

LLVM

- 为什么JIT编译能比完全编译效率更高?
  - 机器级别优化在编译时无法完全实施
    - 不能预先知道对方机器的类型
    - CPU dispatching
      - 运行时根据CPU型号动态选择适当的函数版本
      - 增加空间、增加少量调用开销、需要修改大量代码
      - 实践中只在少量Intel和AMD发布的标准库中使用
  - 运行时优化在编译时无法实施
    - 如运行时知道两个指针不会指向同一地址
- 对于Python、Javascript等动态语言，JIT是唯一优化动态产生代码的方式

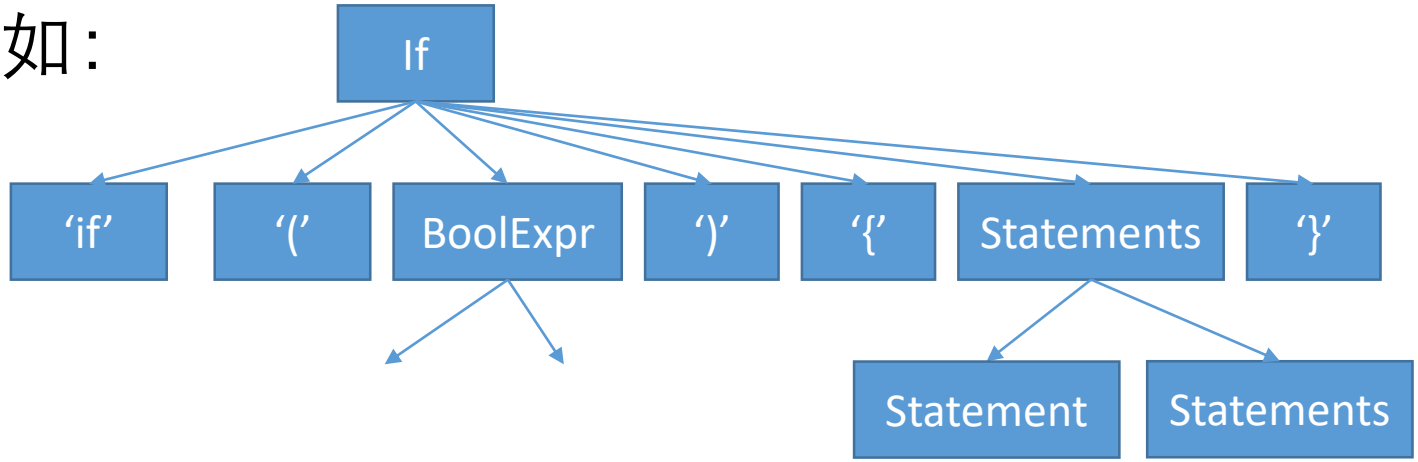
程序的表示

- 抽象语法树
- 三地址码
- 控制流图

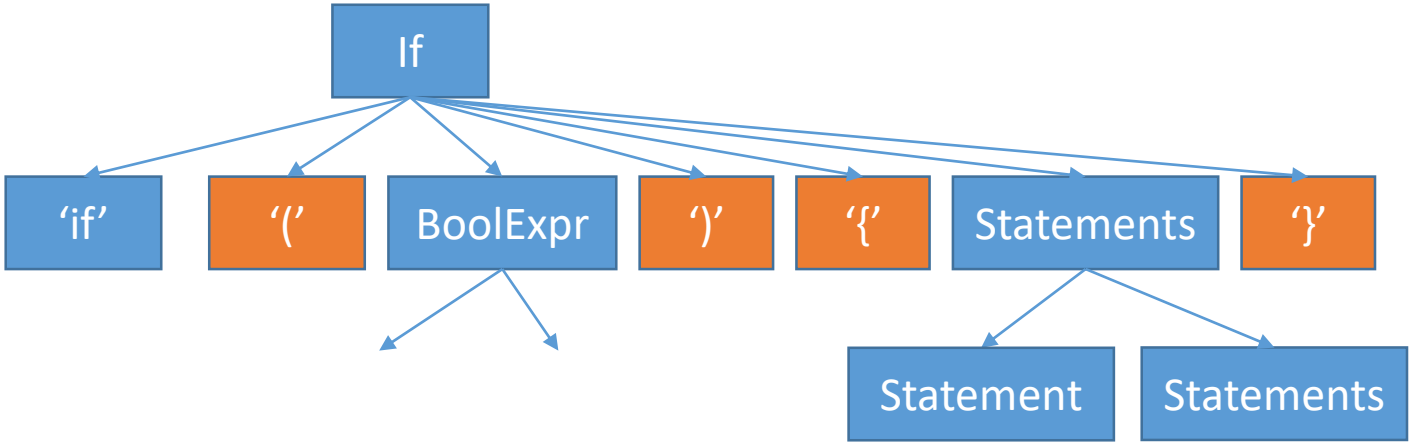
- 均有相应的编译器部件/程序分析框架可以生成

具体语法树

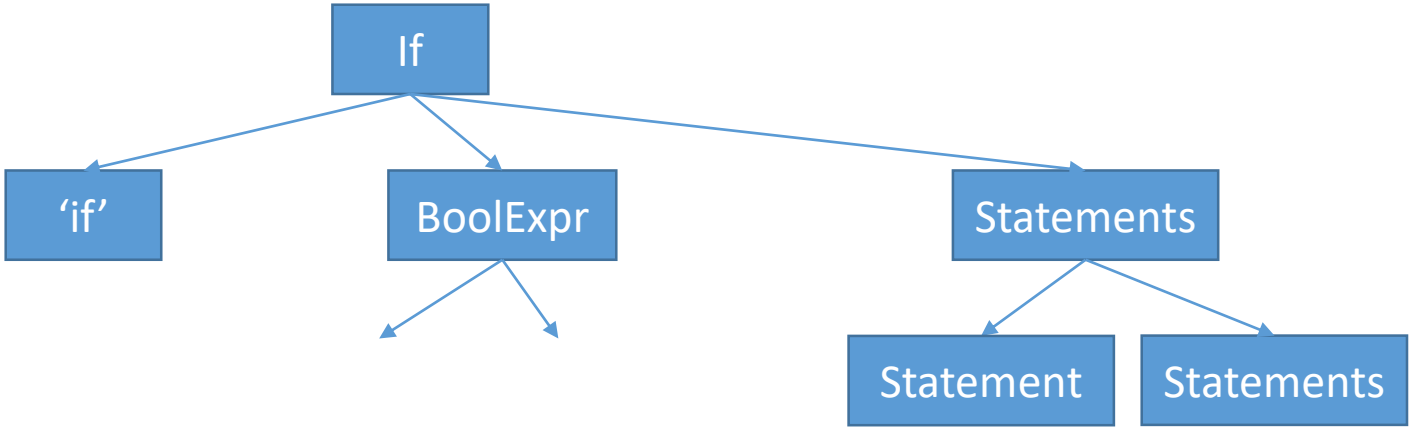
- 具体语法树是一棵树
  - 叶子节点标记为终结符
  - 非叶子节点标记为非终结符
  - 根是开始符号
  - 每个非叶子节点N和其子节点对应一条上下文无关文法推导规则，从N对应的非终结符产生其子节点对应的符号序列
- 如：



抽象语法树



- 具体语法树中有很多我们后期不再关心的符号
- 将这些符号删掉，形成抽象语法树



三地址码

- 中间代码通常采用三地址码的形式表示

```
x = (-b + sqrt(b^2 - 4*a*c)) / (2*a)
```

```
t1 := b * b
t2 := 4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x := t9
```

- 每个三地址码指令只包含一个操作符
- 因为涉及的变量通常不超过3个，所以叫三地址码

三地址码

- 控制语句通常用goto表示

```
...

for (i = 0; i < 10; ++i) {
    b[i] = i*i;
}

...
```

```
      t1 := 0
L1:   if t1 >= 10 goto L2
      t2 := t1 * t1
      t3 := t1 * 4
      t4 := b + t3
      *t4 := t2
      t1 := t1 + 1
      goto L1
L2:
```

控制流图

- 控制流——语句执行的顺序
- 控制流图——表示语句执行顺序的图

- 控制流图的节点
  - 基本块：依次顺序执行的语句序列
- 控制流图的边
  - 基本块之间的可能转移关系

控制流图示例

```
a = 1;
cnt = 0;
while (a < 100) {
   if (a % 7 == 0) cnt++;
   a++;
}
cout << cnt;
```