第二章 编译原理基础知识

编译是将计算机高级语言如 C++、Java、C#编写的源程序翻译成可以在计算机上执行的机器语言的翻译过程。编译过程中分：词法分析、语法分析、语义分析、源代码优化、代码生成和目标代码优化几个过程。ANTLR 解决的是词法分析和语法分析的问题，下面介绍一下编译原理中有关词法分析和语法分析的基本知识。

词法分析是对源程序一个一个字符地读取，从字符中识别出标识符、关键字、常量等相对独立的**记号**（token，也叫符号或单词），形成记号序列**记号流**的过程。如 c、l、a、s、s 五个字符构成了关键字 class,2、3 构成了一个整型数 23。词法分析过程中会滤掉源程序中的空格、换行符和注释等不属于源程序的字符，还可以将记号归类，哪些记号属于标识符，哪些记号属于关键字、整数、浮点数等。记号流是语法分析的基础。

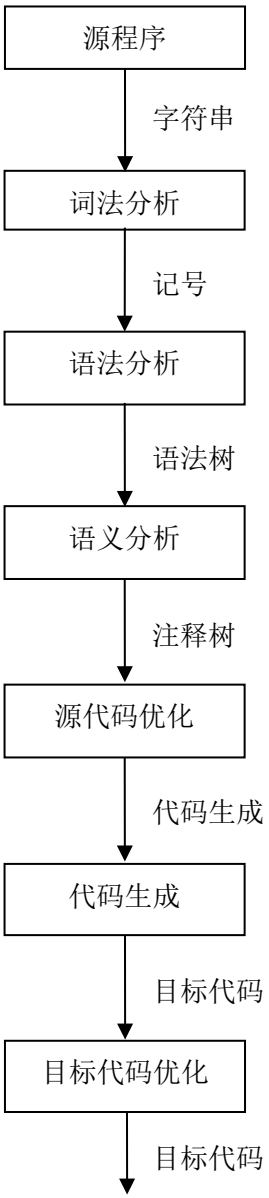


图 2.1

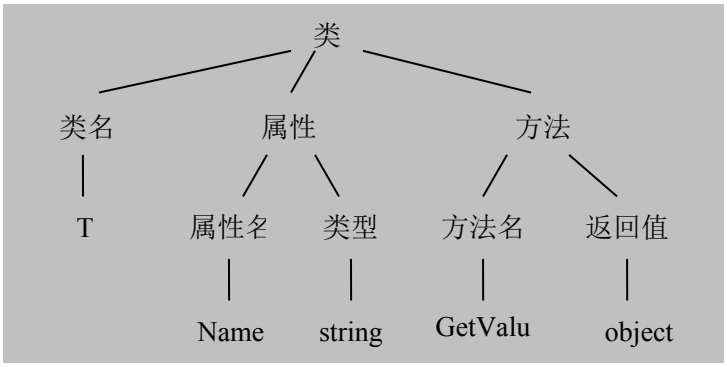
语法分析是根据词法分析输出的记号流，分析源程序的语法结构，并添加代表语法结构的抽象单词（如：表达式、类、方法等），按照语法结构生成语法树的过程。前面讲的词法分析后形成的记号序列是描述程序的直接标识符序列，是线性的。它没有反映出源程序的结构而语法分析后生成的语法树是可以表示源程序结构的数据结构，语法树的叶子节点就是记号。下面举一个简单的例子说明词法分析和语法分析之关的系统，有如下的源程序：

```
class T {
  string Name;// name of T
  object GetValue() {
  }
}
```

进行词法分析后形成记号流：

class T { string Name ; object GetValue ( ) { } }。

进行语法分析后形成语法树：



我们这里不介绍编译原理的其它部分，因为 ANTLR 只涉及到了词法分析和语法分析这两个部分。读者可以去参考原理的书籍。了解 ANTLR 在编译过程中所处的位置后。我们来详细学习一下有关词法分析和语法分析的基础概念。

2.1 什么是文法

一种程序设计语言的语法是规定源程序的写法是否合法的规则，它存在于词法分析和语法分析两个阶段。如：词法分析中 123 表示合法整数，1_23 是不合法整数。在语法分析中 if(boolVar) {}是合法的语句，if(boolVar) { 是不合法的语句。那么我们怎样来定义语法规则呢？定义语法规则的工具是**文法**（grammar），文法是由若干定义语法规则的推导式组成的。下面例子中用文法定义了人类语言的语法规则：

语言 =>（句子）+
句子 => 主语 谓语
谓语 => 动词 宾语
主语 => 名词
宾语 => 名词
名词 =>‘张三’| ‘代码’
动词 =>‘编写’

如，‘张三编写代码’这句话在文法中的推导过程是：
语言 => 主语 谓语
　　=> 张三 动词 宾语
　　=> 张三 编写 名词
　　=> 张三 编写 代码

另外编译原理中一般用大写字母表示一个文法的名称，再加上文法的启始规则组成文法的表示符号。如上面的文法如果名称为 G，可以表示为 G[语言]文法。

2.2 符号表、符号串、推导式和句子

不管是人类语言还是计算机语言都是用符号组成的，英文由字母、数字和标点符号等组成，中文由汉字、数字和标点符等组成，计算机语言由关键字、字母、数字和一些专用符号组成。

这些组成语言的基本**符号**加上推导出基本**符号**的抽象符号集合在一起称为**符号表**，用 V 来表示，符号表是不允许为空的。如 G[语言]文法的符号表是：{语言，句子，主语，谓语，宾语，名词， 动词，‘张三’，‘代码’，‘编写’}，符号表中可以继续推导的中间符号称为非终结符，用 Vn 表示，不能再继续推导的符号称为终结符，用 Vt 表示。G[语言]文法的非终结符集合为：{语言，句子，主语，谓语，宾语，名词， 动词}，终结符集合为{ ‘张三’，‘代码’，‘编写’}。

符号表中符号的任意有穷组合序列称为**符号串**。‘张三张三’、‘张三代码编写’、‘张三语言句子宾语宾语’都是 G[语言]文法符号串。很明显一种文法的符号串不一定是这种文法的合法句子。符号串是有长度的，它的长度是符号的个数，如‘张三张三’的长度是 2，张三语言句子宾语宾语’的长度是 5。

文法是定义语法规则的工具，语法规则简称**规则**（rule）又称推导式或产生式。假设 a 和 b 都是一个文法的符号串，我们用 a => b 表示一个规则，其中 a 不能为空。也就是说“句子 =>”是合法的规则“ => 主语”是不合法的，一个文法要由至少要有一个规则。规则 a => b 使用 b 来替换 a 的过程叫做**推导**，反用 b 来替换 a 的过程叫**归约**。

如 G[S]是一个文法，S 为启始规则，从 S 推导若干次后形成的符号串叫做 G[S]文法的**句型**。如果推导出的符号串全都由终结符组成此符号串叫做 G[S]的**句子**。前面示例中“张三 动词 宾语”是 G[语言]文法的句型，而“张三 编写 代码”是 G[语言]文法的句子。编译原理中也使用四元组来表示文法 G[Vn,Vt,P,S]，其中 G 为文法句称，Vn 为非终结符的集合，Vt 为终结符的集合，P 是文法规则的集合，S 为启始规则。

2.3 文法的类型

一个文法 G[S]，S 为启始规则，如果它的所有规则符合形如：a => b 其中 a 和 b 都是 G[S]文法的符号串，但 a 中至少要有一个非终结符，这时 G[S]文法是**短语文法**。G[语言]为例“宾语张三 => 名词张三”是短语文法的规则，“张三编写=> 名词张三”则不是短语文法，因为“张三”和“编写”都是终结符规则左则没有非终结符。我们可以看出短语文法是对规

则做了一些限制后形成的，下面的文法是对短语文法做进一步限制形成的。

如果 G[S]的所有规则都满足形如：a => b 其中 a 的长度要小于等于 b，这时 G[S]文法是**上下文有关文法**（context-free grammars）。上下文有关文法的更形象的定义是：文法的所有规则满足 aBc => abc 的形式，其中 B 是非终结符，a、b、c 是符号串。也就是说 B => b 只在前面有 a 后面有 c 的情况下才能推导，所以是上下文有关的。例如：“张三动词程序 => 张三编写程序”是上下文有关文法的规则。

如果 G[S] 的所有规则都满足形如：a => b 其中 a 是一个非终结符，b 是符号串，这时 G[S]文法是**上下文无关文法**（context-sensitive grammars）。就是说 a 推导出 b 与其前后是什么符号串无关。上面 G[语言]的文法就是上下文无关文法。

如果 G[S] 的所有规则都满足形如：A=> aB 或 A=>a 其中 A 和 B 是非终结符，a 是终结符，这时 G[S]文法是**正规文法**（regular grammars）。就是说规则的右则要以终结符开头。如：“谓语 => 编写 宾语”，“动词 => 编写” 都是正规文法的规则简称**正规式**，“谓语 => 动词 宾语” 就不是正规式。

这四种文法是对规则的限制逐步加强形成的。正规文法是上下文无关文法的特例，上下文无关文法是上下文有关文法的特例，上下文有关文法是短语文法的的特例。文法产生的语言就是该文法的语言，如：上下文无关文法产生的语言就是**上下文无关语言**，正规文法产生的语言就是**正规语言**。文法是语言模型。计算机语言中普遍采用上下文无关文法来定义语法规则。下面我们介绍上下文无关文法的语法树。

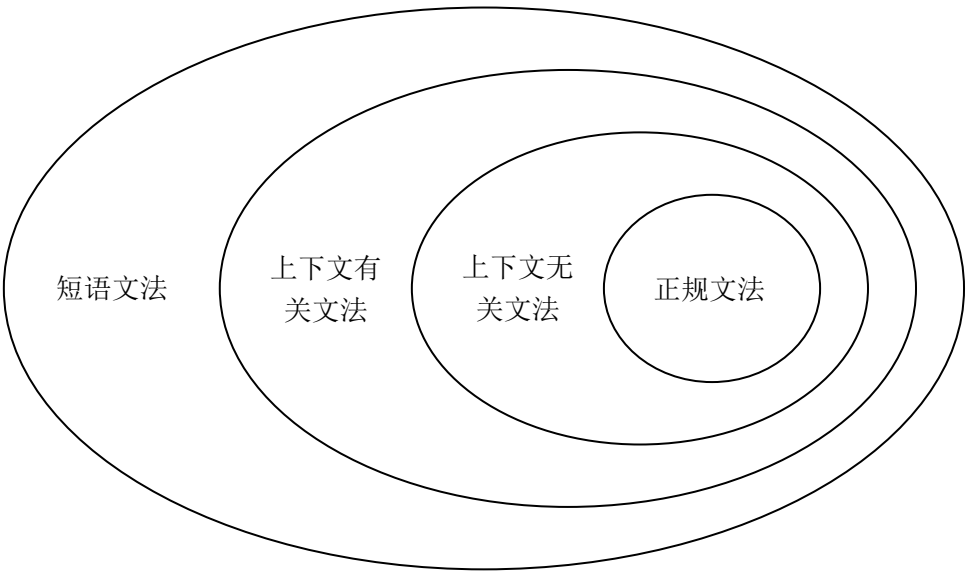


图 2.2

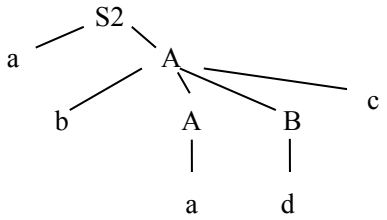
2.4 语法树

编译技术中用语法树来更直观的表示一个句型的推导过程。前面我们已经提到过语法树，

相信读者已经对语法树有了一定的认识，这里我们给出上下文无关文法语法树的定义：给定上下文无关文法 G[S]，它的语法树的每一个节点都有一个 G[S]文法的符号与之对应。S 为语法树的根节点。如果一个节点有子节点。则这个节点对应的符号一定是非终结符。如果一个节点对应的符号为 A，它的子节点对应的符号分别为 A1，A2，A3…..Ak，那么 G[S]文法中一定有一个规则为：A=>A1 A2 A3 …..Ak。满足这些规定的树**语法树**也叫**推导树**。下面给出一下文法 K[S2]和 K[S2]文法的一个语型，我们用语法树来显示这个语型的推导过程。

K[S2]文法： S2 => aA
A=> bABc
A=>a
B=>d

K[S2]文法对于语型 abadc 的推导树为：

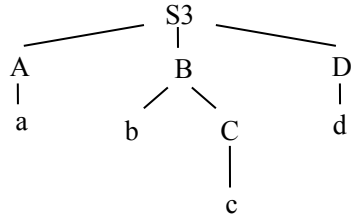


推导的过程中优先选择不同的规则进行推导会使推导过程有所不同。下面举一个例子。

K[S3]文法：S3 => ABD
A=>a
B=>bC
C=>c
D=>d

下面是对于句型 abcd 的三种不同推导过程。

① S3=> ABD => aBD => abCD => abcD => abcd
② S3=> ABD =>AbCD=>AbcD=>abcD=>abcd
③ S3=> ABD =>ABd=>AbCd=>Abcd=>abcd



我们可以注意到①过程中所有推导都是选择的最左边的非终结符进行替换。③过程中所有推导都是选择的最右边的非终结符进行替换。其中①被称为最左推导，③被称为最右推导。

这三种推导都对应一棵语法树，这说明语法树反应了此句型的所有推导过程。

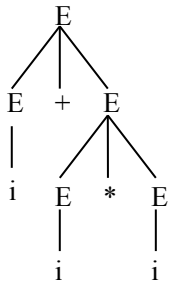
但是对于有些句型来说，它对应的语法树不一定唯一的。也不是说一棵语法树不一定能反应一个句型的所有推导过程，如下面给定文法。

S4[E]文法：
E => E + E
E => E * E
E => (E)
E => i

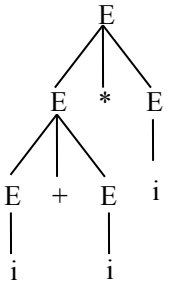
对于 i + i * i 句型，我们可以写出下面两种最左推导的过程：

① E => E + E => E + E * E => i + E * E => i + i * E
② E => E * E => E + E * E => i + E * E => i + i * E

①过程中第一步使用了 E => E + E 规则，②过程中第一步使用了 E => E * E 规则，不管选择哪个规则都是最左推导。下面有两棵语法树与之对应。对于一个文法的句型如果有多于一棵的语法树与之对应，则这个文法是有**二义性**的文法。也可以用另一种方法判断，如果一个文法的最左或最右推导的过程是不唯一的也可以说这个文法是有二义性的文法。



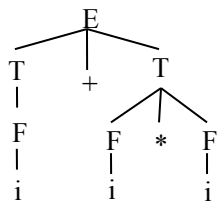
推导①的语法



推导②的语法

二义性文法是在开发语法分析器时需要解决的问题，我们将 S4[E]加入操作符优先关系改成下面形式可以去掉文法的二义性。

S5[E]文法：
E => T + T
E => T
T => F * F
T => F
F => (E)
F => i



使用 S5[E]文法对于 i + i * i 句型的推导过程和语法树是唯一的：
E => T + T => T + F * F => F + F * F => i + F * F => i + i * F => i + i * i

由于文法简单所以二义性比较容易解决，但是当文法很复杂的时候，检查文法中是否存

在二义性就困难了。但 ANTLR 的开发者不用担心，ANTLR 会象我们编译普通源程序那样提示文法中的问题,其中包括文法的二义性问题,这使我们可以很容易的找到存在二义性的规则。

2.5 分析方法

前面讲到了句型的推导过程和生成语法树的过程,有了语法树就已经很清晰的看到了句型的结构，我们可以很容易的从语法树中获得我们相要的信息，这个过程就是语法分析。如图 2.3 显示了对于 SELECT F1, F2 FROM Table1 WHERE F1=”a”的语句进了语法分析后生成的语法树，利用非终结符节点 SeletctList 很容易对应 Select 语句的 F1,F2 部分。

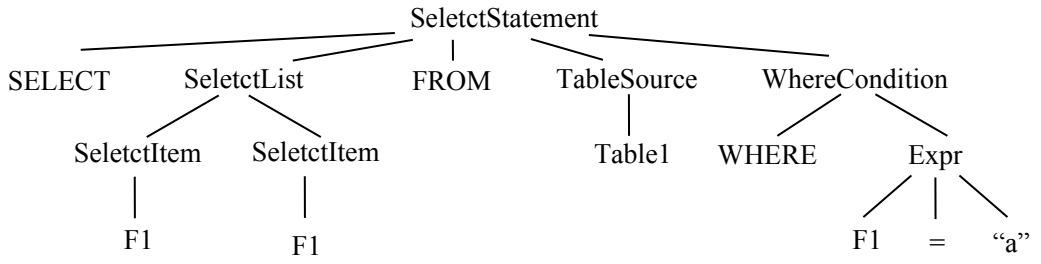
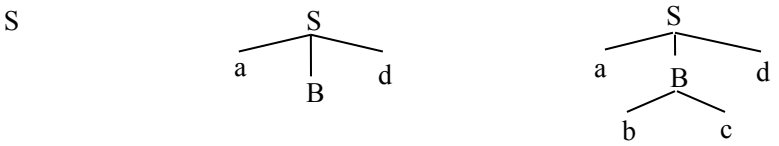


图 2.3

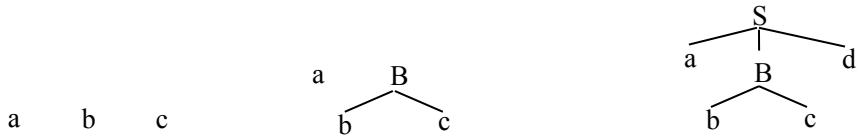
我们前面的推导是靠自己主观判断，选择适当的规则进行推导的。那么如何用程序来实现这个过程呢？语法分析方法分两大类: **自顶向下的分析方法**和**自下而上的分析方法**。ANTLR 使用的是自顶向下的分析方法。自顶向下的分析方法的思路是从起始规则开始选择适当的规则反复推导，直到推导出待分析的语型。如果推导失败则反回选择其它规则进行推导（这个过程叫做回朔（backtrack)），如果所有规则都失败说明这个句型是非法的。下面举一个分析的示例。

D1[S]文法：
S => aBd
B => b
B = > bc

对于 abcd 句型进行自顶向下分析，第一步唯一的选择规则 S => aBd，第二步对非终结符 B 的推导，先选择 B => b 推导出 S => abd 这和句型 abcd 不同所以推导失败。现在返回到对 B 的推导，选择另一个规则 B => bc 行出 S => abcd 这次推导成功。



自下而上的分析方法与自顶向下分析方法相反，过程是逐个的扫描句型的符号使用适当的规则进行反复归约，直到归约成启始规则 S。如果这个过程失败，则返回选择其它规则进行归约。我们使用自下而上的分析方法对 D1[S]示例进行分析。首先是扫描到了第一个符号 a，a 无法归约没有象 X => a 这样的规则。然后继续扫描符号 b，b 可以用 B = > b 来归约得出 aB。然后扫描到符号 c，这时 aBc 不能继续进行归约造成过程失败。所以要返回前一步使用 B => bc 来归约得出 aBd，aBd 可以用 S => aBd 归约到 S。



不管是自下而上的分析方法还是自顶向下分析方法如果选择的规则不正确，就要返回重新尝试用其它规则进行推导或归约。这在实际操作中会浪费很多时间分析程序的执行效率会降低。为了解决这个问题，在编译技术中使用一种向前探测符号的方法（lookahead）保证可以正确选择规则。如 D1[S]示例的自顶向下分析的第二步如果选择 B => b 则得出 ab 句型后面的符号为 c，如果选择 B => b 规则推导将得出 abd，所以不能选择 B => b 规则。如果选择 B => bc 可以得出 abc 和后面的符号 d 相符，所以应该选择 B => bc 规则。

在自下而上的分析方法中读取前两个符号 ab 时 b 可以用规则 B=>b 归约，这时向前探测一符号为 c 可以得出 aBc，但 aBc 没有规则可以归约。所以再读取一个符号 c 符，选择 B=>bc 规则归约。向前探测一符号为 d，aBd 可以规约成 S 分析成功。

2.6 有害规则

在文法可能会出现一些无用的、造成文法二义性的规则。如左右两侧相同的规则 A =>A，这种规则在文法中没有意义，如果还有一条规则 S => A，当我们用 A 归约时 A=>A 会干扰使分析器不知道应该用哪一个规则归约同，如果不断使 A => A 归约会造成死循环。如果一个非终结符不出现在任何规则的右部，那么这个非终结符是**不可达**的，也就是说没有句型在推导或归约过过程中会用到这个非终结符。如一个文法中有规则 A => a 但是没有形如 X => A 的规则那么 A=>a 在文法中是多余的。还有一种叫做**不可终止**的非终结符，如一个文法中对于 A 非终结符来说只有 A => Aa 这个规则，可以看出 A 无法推导出一个句子它也是多余的。这些规则应该在文法中删除。

2.7 左递归、右递归

形如 A => Ab 的规则，A 的定义是递归的可以推导出 Abbbb…b，左侧的非终结符 A 可以不断地推导出 Ab，这种处于规则左侧的递归叫**左递归**。递归也可能出现在多个非终结符之间 A=>Bd，B=>Bc 这里的 A=>Bd 也是左递归。例如我们要定义一个整型数其规则为：INT => INT Digital，Digital => 0|1|2|3|4|5|6|7|8|9，规则 INT 用左递归实现了多位整型数的定义。相反形如 A => bA 的规则，A 的定义也是递归的但和左递归相反非终结符 A 在规则的右侧这样递归叫做**右递归**。我们可以把整型数定义的规则用右递归的方法定义为 INT => Digital INT，Digital => 0|1|2|3|4|5|6|7|8|9。使用这两种递归的方法时，要看语法分析程序的分析方式，如果语法分析程序是从左向右分析的，那么使用右递归比较适合，反之使用左递归比较适合。

2.8 文法定义基础

ANTLR 的文法定义使用了类似 EBNF（Extended Backus-Naur Form）的定义方式，是一种强大简洁的文法定义方式。本章前面的文法定义的写法比较繁琐，定义复杂的文法时非常不便，文法的可读性也会较差。ANTLR 的文法定义方式形象直观，可以用很短的行数描述以前要很多行才能表示的文法内容。

规则的表示：文法是由规则组成的，本章前面的规则都是用 A=>a 形式来表示的。ANTLR 用 A : a;来表示规则，“:”代替了“=>”。ANTLR 的规则要以分号“;”结束。我们可以方便地称规则 A : a 为规则 A。在规则中有几种运算关系，选择、连接、重复、可选。

连接“ ”：规则 A : a b c; a、b、c 之间用空格分隔。此规则接收句型 abc，符号 a、b、c 是按顺序连接起来的关系。

选择“| ”：规则 A : a | b | c; “|”表示“或”的关系，符号 A 可以推导出 a 或 b 或 c，也就是在 a、b、c 中选择。这要比写成 A : a; A : b; A : c;方便得多。连接和选择可以联合起来使用，如 A : a b c | c d e;。有进也会使句型的数量增多如：A : B D; B : a | b; D : c | d;这时符号 A 推导出的句型有 ac、ad、bc、bd 四种。

重复“*，+”：定义符号的多重性，规则 A : a*; “*”表示 a 可以出现 0 次或多次。A : a*; 相当于 A : A a | ;。这样可以避免递归的定义，可文法定义中递归往往引起文法的二义性。如果 a 至少要出现一次可以表示为 A : a+; “+”表示 a 可以出现 1 次或多次。相当于 A : A a | a;。重复可以和连接、选择一起使用如：A : a * b | c + d;。

可选“？”：规则 A : a?; “?”表示 a 可以出现 0 次或 1 次，即 a 可有可无。相当于 A : a | ;。可选可以和连接、选择、重复一起使用如：A : a* b? | c+ d?;。

子规则“（）”：规则 A ： (a b) | b; a 与 b 在括号中，这样“(a b)”形成了一个子规则，也就是说可以把规则写成 A ： B | b; B ： a b;两个规则表示，我们把 B 规则用括号括起来放到 A 规则中这样就是 A 规则的子规则了。利用子规则也可以把多个符一起进行描述，A ： (a b c)* 规则中 a、b、c 三个符号可以一起重复 0 次或多次。子规则有利于我们把很复杂的多个规则写到一起，有时这样写会使文法既简练又直观。子规则和前面的各种特性用到一起可以把复杂的文法写的很浓缩。如：A ： (a b c)* | (c d)+ e?;。

值得注意的是如果我们的规则中有“()”的字符该如何表示？因为子规则也是用“()”表示的。在 ANTLR 中表示字符要用“’”单引号括起来，用‘(’ ‘)’来表示括号字符。前面讲到的表示文法规则的符号“| * + () ?”叫做文法的元符号。

注释“// //”：**和一般编程语言一样，ANTLR 在文法定义中也可以添加注释。用//来添加单行注释，如规则 E ： ‘(’ E ‘)’ | INT // E 表示算术表达式。用/*…*/添加多行注释，与 C++相同。

2.9 本章小结

本章学习了编译原理的基础知识。包括：什么叫词法分析和语法分析，ANTLR 在编译技术中所处的位置。什么叫文法，规则。什么叫短语文法，上下有关文法，上下文法无关文法，正规文法。语法树，句型的最左推导最右推导和文法的二义性，自顶向下的分析方法和自下而上的分析方法。ANTLR 的文法定义方法。