第 2 章 词法分析和语法分析

引言故事：

词法分析和语法分析是对语言语义进行理解的基础。自古以来，中国以文字、音韵和训诂为主导的语言文字，一直是古代人文学术的重要内容。汉字从甲骨文开始的演化过程，逐渐形成了较为系统的 “表意”特点，每个字都是音、形与义的综合表示。

虽然我国的汉语言未能如西方语言那样，较早地形成以语言中词法及语法组织结构为研究对象的系统性的语法学，但是，中国古代也有着严肃语法观念。中国语言学史上的语法研究早已在经典文献中得到启示。《尚书》和《周礼》等文献中，已经提及了名词和实际事物之间的关系。《论语》和《孟子》中探讨了词义的多义性和上下文的影响，而《说文解字》深入研究了汉字的构成和意义。这些文献中所包含的语法分析，也许就是中国古代语法学的雏形。在古汉语中，语法主要依赖于语序和虚词进行构建。时间先后是汉语语序中的一个重要规则，而虚词则用于表达实词之外的语法意义。在唐宋时期，中国的语法研究不仅来源于梵文佛典，还受到了阿拉伯语和波斯语语法学的影响。汉语学者们通过对这些语法理论的研究和比较，逐渐形成了自己的语法学体系。清朝时期，学者们提出语法学概念，并对句法结构有了清晰的认识，明确了根据句法结构进行训诂考据的方法。例如，基于句法规则来校勘古籍。这样的语法分析方法，使得中国的语法研究从零散走向系统。直到 1957 年，林守翰教授发表的《汉语语法》标志着汉语语法学的现代化，开创了汉语语法学的新时代。

语音、词汇和语法构成了对古汉语辨析的三要素。其中，语法是组词成句的结构规律与约束，分析语法能够探求词义。而词性则不同，在不同语句中同一个词所表达的意思可能完全不同。根据上下文的语义环境和句法特点，来准确地辨析词性，将有助于准确理解语句。例如：设酒杀鸡食（出自《桃花源记》），这里的“食”为名词，意为“饭” ； 杀鸡为黍而食 之（出自《论语·微子》），这里的“食”则为使役动词，意为“使（子路）食”。另外，词语在句子中一般都具有句法功能，而不同的句法功能，则对应着不同的词义。例如：李斯上书说，乃止逐客令（出自《秦始皇本纪》），这里的“客”充当宾语，意为“外来的人”。齐将田忌善而客待之；齐将田忌善

而客待之（出自《孙子吴起列传》），这里的“客”充当状语，意为“像上宾般款待”。句法结构的分析着重于句中各部分之间的关系。即，如何通过虚词和词序这些语法手段来构成各种不同的句式。此外，在古汉语中往往存在这样的情况，几个语句所用的某些词语相同，但因句法不同而导致词义不同；而有些特殊结构的句法，更使某些词语显示出特殊的含义。例如：或王命急宣，有时朝发白帝，暮到江陵（出自《三峡》），这里的“或”，意为“如或”；为医或在齐，或在赵，在赵者名扁鹊（出自《史记·扁鹊列传》），这里的“或”，则意为“或者”。

闻一多先生曾在《诗经通义》的“匏有苦叶”条中提到：“背儒无语法观念，其致误往往若是”。这正说明了语法分析对确定词义的重要性。语言是一个整体，语法也不是简单的结构规律，对词性、词义和句法的辨析是对古汉语文献理解的重要组成部分。

同样地，在编译原理的学习当中，词法分析和句法分析（即语法分析）是程序编译过程中不可或缺的重要步骤，是后续语义理解和代码生成的基础。软件开发人员使用高级编程语言编写程序，使人类能够相对容易地理解程序，但计算机无法直接理解。因此，我们引入编译器将程序源代码转换为机器可以理解的形式是必要的一步，而如何分析程序源代码的词法和语法则是编译过程中的第一个步骤。

本章要点：

词法分析是编译器工作流程中的第一个关键步骤。在该步骤中，编译器将源代码从字符流转换为词法单元序列，并输入至语法分析器中。而语法分析的主要目标是检验源代码是否符合语法规范。本章内容主要涵盖了在编译过程中进行词法分析和语法分析的过程以及关键算法。其中，重点讨论了词法分析中正则表达式的基本概念和原理，并分析了非确定的有限状态机和确定的有限状态机之间的转化和技术特点。在语法分析部分，本章介绍了上下文无关文法、以及自顶向下和自底向上两大类语法分析算法，以形成完整的编译器词法和语法分析步骤。

此外，本章中也讨论了对应的技术实践内容，在所提供的技术指导下，编写一个程序对使用 C--语言书写的源代码进行词法分析和语法分析（C--语言的文法参见附录 A），并打印出相应的分析结果。该实践任务要求使用词法分析工具 GNU Flex 和语法分析工具 GNU Bison，并使用 C

语言来完成。在这两个强大工具的帮助下，结合本章前面介绍的理论方法，编写一个能进行合理词法分析和语法分析的程序将是一件轻松愉快的事情。

需要注意的是，由于在后面的实践内容中还会用到本章中已经写好的代码，因此保持良好的代码风格、系统地设计代码结构和各模块之间的接口对于整个编译实践来讲是相当重要的。

思维导图(理论方法部分):


第二章
词法分析
词法分析的主要任务
输入：源程序的输入字符序列
输出：词法单元序列
实现：词法分析器
正则表达式
一种描述词法单元模式的重要表示方法
正则表达式的表示方法
正则表达式的运算规则
有限状态自动机
将正则表达式的模式转化为流图：状态转移图
不确定的有限状态自动机（NFA） — 转移图的边没有限制
确定的有限状态自动机（DFA） — 每个状态只有一条可以离开的边
从NFA到DFA的转化 — 子集构造法
简化DFA的状态 — 状态最小化算法
语法分析
语法分析的主要任务
输入：词法单元序列
输出：语法分析树（显式或隐式）
实现：语法分析器
高效的语法表示方法 — 上下文无关文法
终结符号集合
非终结符号集合
产生式集合
开始符号
自顶向下的语法分析
由语法分析树的根节点开始分析，对串自左至右完成最左推导
递归向下的语法分析
会遇到冲突，需要回溯
基于预测，多向前看1个字符
First集
Follow集
构造预测分析表
消除左递归 — 避免预测分析表出现多个项
LL(1)文法 — 不需要进行回溯的文法
自底向上的语法分析
由语法分析树的叶子节点开始分析，对串自左至右完成最右推导
LR语法分析
基于移入-归约的语法分析引擎
LR分析表的构建
LR分析方法的改进
SLR(1)分析
LR(1)分析
LALR(1)分析

2.1 词法分析和语法分析的理论方法

编译器的主要功能是将程序从一种语言翻译到另一种语言；为此，编译器需要将源程序进行拆分，解析其中的结构和意义，再将这些成分用另一种合理的方式组合起来。源程序的成分解析主要由编译器的前端完成，通常包含三个关键步骤：词法分析、语法分析和语义分析，其中词法分析步骤主要是将源程序分解成独立的词法单元，语法分析步骤则以词法分析器输出的词法单元序列作为输入，进一步分析程序的语法结构，而语义分析步骤则主要是解析程序中各个符号的具体含义。在本章中，我们将介绍词法分析与语法分析步骤的理论方法。

2.1.1 词法分析概要

本节我们主要讨论如何构建一个词法分析器。词法分析是编译的第一阶段，词法分析器的主要任务是读入源程序的字符信息，根据词法规则将它们组成**词素**，生成并输出一个**词法单元（Token）**序列，每个词法单元对应于一个词素。词法分析器通过解析字符信息获得的词法单元序列被输出到语法分析器进行语法分析。词法分析器还要和**符号表**进行交互。当词法分析器发现了一个标识符的词素时，它需要将这个词素添加到符号表中。

图 2.1 展示了这种交互过程。通常，交互是由语法分析器调用词法分析器来实现的。图中的语法分析器调用 *getNextToken* 使得词法分析器从它的输入中不断读取字符，直到它识别出下一个词素为止。词法分析器根据这个词素生成下一个词法单元并返回给语法分析器。

词法分析器在编译器中主要负责源程序的读取与词素的识别，除此之外，它还会完成其它任务。例如，一个重要任务是过滤掉源程序中的注释和空白（空格、换行符、制表符以及在输入中用于分隔词法单元的其它字符）；而另一个任务是将编译器生成的错误消息与源程序的位置联系起来，反馈给开发者。例如，词法分析器可以负责记录遇到的换行符的个数，以便给每条出错消息赋予一个行号。在某些编译器中，词法分析器会建立源程序的一个副本，并将出错消息插入到适当的位置。同时，如果源程序中使用了一个宏预处理器，则宏的扩展也可以由词法分析器完成。

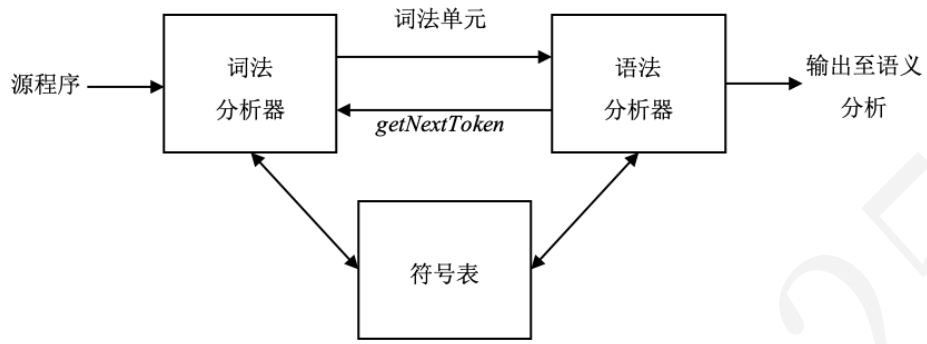

图 2.1 词法分析器与语法分析器之间的交互

一般而言，词法分析器的工作流程可以分成两个主要的阶段：

（1）**扫描阶段：**主要负责处理在词法单元字符生成阶段不需要的字符，比如删除注释和将多个连续的空白字符压缩成一个字符；

（2）**词法分析：**主要负责处理扫描阶段的输出并生成词法单元，它是词法分析器中的核心部分。

在讨论词法分析时，我们使用以下三个相关术语：

词法单元：由一个词法单元名和一个可选的属性值组成。词法单元名是一个表示某种词法单元的符号，比如一个特定的关键字，或者代表一个标识符的输入字符序列。

模式：描述了一个词法单元的词素可能具有的形式。当词法单元是一个关键字时，它的模式就是组成这个关键字的字符序列。对于标识符和其它词法单元，模式则是一个更加复杂的结构，它可以和很多符号串匹配。

词素：源程序中的一个字符序列，它和某个词法单元的模式匹配，并被词法分析器识别为该词法单元的一个实例。

表 2.1 给出了一些常见的词法单元和它们非正式的描述模式，并给出了一些示例词素。

表 2.1 词法单元的例子[1]

词法单元	非正式描述	词素示例

[1] 该例子来源于: 《编译原理》，Alfred V. Aho等著，赵建华等译，机械工业出版社，第69页，2009年。

if	字符 i，f	if
else	字符 e，l，s，e	else
comparison	< 或 > 或 <= 或 >= 或 == 或 !=	<=，!=
id	字母开头的字母 / 数字串	pi，score，D2
number	任意数字常量	3.14159，0.602
literal	在“ ”之间，除“”以外的任何字符	“core dumped”

在很多程序设计语言中，下面的类别覆盖了大部分词法单元：

（1）表示关键字的词法单元，其模式就是关键字本身的字符序列；

（2）表示运算符的词法单元，其模式可以是单个运算符，也可以是一类运算符；

（3）表示标识符的词法单元，其模式通常由若干字符或数字序列构成；

（4）表示常量的词法单元，比如数字和字面值字符串，其模式就是常量本身；

1. 词法单元的属性

如果有多个词素可以与一个模式匹配，那么词法分析器必须向编译器的后续阶段提供有关被匹配词素的附加信息。例如，0 和 1 都能与词法单元 number 的模式匹配，但是对于代码生成器而言重要的是知道在源程序中找到了哪个词素。因此，在很多情况下，词法分析器不仅仅向语法分析器返回一个词法单元名字，还会返回一个描述该词法单元的词素的属性值。词法单元的名字将影响语法分析过程中的决定，而这个属性则会影响语法分析之后对这个词法单元的翻译。

我们假设一个词法单元至多有一个相关的属性值，当然这个属性值可能是一个组合了多种信息的结构化数据。一般来说，与一个标识符有关的信息（例如它的词素、类型、第一次 出现的位置等）都保存在符号表中。因此，一个标识符的属性值可以是一个指向符号表中该标识符对应条目的指针。

2. 词法错误的处理

如果没有其它组件的帮助，词法分析器很难发现源代码中的错误。比如，当词法分析器在 C 程序片断：`fi(a==f(x))`中第一次遇到 `fi` 时，它无法指出 `fi` 究竟是关键字 `if` 的误写还是一个未声明的函数标识符。由于 `fi` 是函数标识符的一个合法词素，因此词法分析器必须向语法分析器返回这个词法单元，而让编译器的另一个阶段（在这个例子里是语法分析器）去处理这个错误。然而，假设出现所有词法单元的模式都无法与剩余输入的某个前缀相匹配的情况，此时词法

分析器就不能继续处理输入。当出现这种情况时，最简单的错误恢复策略是“**恐慌模式（Panic）**”恢复。我们从剩余的输入中不断删除字符，直到词法分析器能够在剩余输入的开头发现一个正确的词法单元为止。这种恢复技术可能会给语法分析器带来混乱，但是在交互计算环境中，这种技术已经足够了。

2.1.2 正则表达式

正则表达式（Regular Expression）是一种用来描述词素模式的重要表示方法。虽然表达式无法表达出所有可能的模式，但是它们已经能够高效地描述在处理词法单元时要用的模式类型。在介绍正则表达式之前，我们首先介绍一下字母表、串和语言等相关的概念。

字母表（Alphabet）是一个有限的符号集合。常见的符号包括字母、数位和标点符号等。常见的字母表有只包含两个符号的二进制字母表{0，1}、用于多种软件系统的 ASCII 字母表，以及包含了大约 10 万个世界各地字符的 Unicode 字母表等。字母表上的一个**串（String）**是该字母表中符号的一个有穷序列。在语言理论中，术语“句子”和“字”也常常当作“串”的同义词。串 s 的长度，记作|s|，是指 s 中符号出现的次数。**空串（Empty String）**是长度为 0 的串，通常用 ε 表示。**语言（Language）**是某个给定字母表上一个任意的可数的串的集合。这个宽泛的定义使得空集∅和仅包含空串的集合{ε}也都是语言。

下面列出一些与串相关的常用术语：

- 串 s 的**前缀（Prefix）**是从 s 的尾部删除 0 个或多个符号后得到的串。例如 ε、app 和 apple 都是 apple 的前缀。
- 串 s 的**后缀（Suffix）**是从 s 的开始处删除 0 个或多个符号后得到的串。例如 ple、apple 和 ε 都是 apple 的后缀。
- 串 s 的**子串（Substring）**是删除 s 的某个前缀和后缀之后得到的串。例如 ppl、ple 和 ε 都是 apple 的子串。
- 串 s 的**真前缀、真后缀、真子串**分别是 s 的不等于 ε 和本身的前缀、后缀和子串。
- 串 s 的**子序列（Subsequence）**是从 s 中删除 0 个或多个符号后得到的串，删除的符号可以不相邻。例如，ale 是 apple 的一个子序列。

如果 a 和 b 是串，那么 a 和 b 的**连接（Concatenation）**是把 b 附加到 a 后面形成的串。空串是连接运算的单位元，对于任何串 s 都有 $s\varepsilon=\varepsilon s=s$。如果把串的连接看作串的“乘积”，那么就可以定义串的“指数”运算：定义 $s^0=\varepsilon$，并且对于 $i > 0$，$s^i=s^{i-1}s$。由 $\varepsilon s=s$ 可知，$s^1 = s$，$s^2 = ss$，$s^3 = sss$，依次类推。

在词法分析中，最重要的语言上的运算是并、连接和闭包运算，如表 2.2 所示。语言间的连接就是以各种可能的方式，从第一个语言中任取一个串，从第二个语言中任取一个串，然后将它们连接得到所有串的集合。一个语言 L 的 **Kleene 闭包（Closure）**记为 L^*，即 L 自身连接 0 次或者连接多次后得到的串集。其中，L 连接 0 次的集合 L^0 被定义为{ε}。同理，L^i 可以被归纳地定义为 $L^{i-1}L$。L 的**正闭包**和 Kleene 闭包基本相同，记为 L^+，但是不包含 L^0。

因为正则表达式可以描述所有通过对某个字母表上的符号应用这些运算符而得到的语言，我们可以用正则表达式这一种表示方法描述语言，如表 2. 2 所示。正则表达式可以由较小的正则表达式按照递归的方式构建。每个正则表达式 re 表示一个语言 L(re)，这个语言本身也是根据 re 的子表达式所表示的语言递归定义的。下面的规则定义了某个字母表 Σ 上的正则表达式以及这些表达式所表示的语言：

表 2. 2 正则表达式运算符号及其意义

符号	意义
a	一个表示字符本身的原始字符
ε	空字符串
X | Y	可选，在 X 和 Y 之间选择
X Y	连接，X 之后跟随 Y
X *	重复（0 次或 0 次以上）
X +	重复（1 次或 1 次以上）
X ?	选择，X 的 0 次或 1 次出现
[a – z A – Z]	字符集
.	句点表示除换行符之外的任意单个字符
“a . * + ”	引号，引号中的字符串表示文字字符串本身

正则表达式构建的归纳基础包含如下两个基本规则：

（1）ε 是一个正则表达式，且 L(ε) = {ε}，同时该语言只包含空串；

（2）如果 x 是Σ上的一个符号，那么 x 是一个正则表达式，并且 L(x)={x}。该语言包含一个长度为 1 的符号串 x；

正则表达式的归纳可以看作是由较小的正则表达式按照一定的规则进行递归构建较大正则表达式的过程。假定 x 和 y 都是较小的正则表达式，其语言分别为 L(x)和 L(y)，那么，归纳规则可以表示如下：：

（1）(x)|(y) 是一个正则表达式，表示语言 L(x)∪L(y)；

（2）(x) (y) 是一个正则表达式，表示语言 L(x)L(y)；

（3）(x)*是一个正则表达式，表示语言(L(x))*；

（4）(x)是一个正则表达式，表示语言 L(x)，表达式两边加上括号不影响其所表示的语言；

根据上面的定义，正则表达式可能包含一些冗余的括号。如果我们按照如下的约定，就可以删减一些不必要的括号：

（1）一元运算符*具有最高的优先级，并且是左结合的；

（2）连接具有次高的优先级，并且是左结合的；

（3）|的优先级最低，也是左结合的；

应用这个约定，我们可以将(x)|(y)*(z)改写为 x|y*z。这两个表达式都表示相同的串集合，其中的元素要么是单个 x，要么是由 0 个或多个 y 后面再跟一个 z 组成的串。表 2. 3 给出了一些对于任意正则表达式都成立的代数定律。

表 2. 3 正则表达式的代数规律

定律	**描述**
r|s = s|r	| 是可以交换的
r|(s|t) = (r|s)|t	| 是可结合的
r(st) = (rs)t	连接是可结合的
r(s|t) = rs|rt;　(s|t)r = sr|tr	连接对 | 是可分配的
εr = rε= r	ε 是连接的单位元
r* = (r|ε)*	闭包中一定包含 ε
r** = r*	* 具有幂等性

可以用一个正则表达式定义的语言叫做**正则语言**（**Regular Language**）。如果两个正则表达式 x 和 y 表示同样的语言，则称 x 和 y **等价**（**Equivalent**）。比如 a|b=b|a。正则表达式遵守一些代数定律，每个定律都断言两个具有不同形式的表达式等价。为了表示方便，我们可以给某些正则表达式命名，并在之后的表达式中像使用符号一样使用这些名字。如果Σ是基本符号的集合，那么一个**正则定义（Regular Definition）**是具有如下形式的定义序列：

$def_1 \rightarrow re_1$

$def_2 \rightarrow re_2$

…

$def_n \rightarrow re_n$

其中，每个 def_i 都是一个不在字母表Σ中的新符号，并且各不相同；每个 re_i 是字母表 $\Sigma \cup \{def_1, def_2, \cdots, def_{i-1}\}$上的正则表达式。

我们限制每个 re_i 中只含有Σ中的符号和在它之前定义的各个 def_j，因此避免了递归定义的问题，并且我们可以为每个 re_i 构造出只包含Σ中符号的正则表达式。我们可以先将 re_2（不使用 def_1 之外的任何 def）中的 def_1 替换为 re_1，然后将 re_3 中的 def_1 和 def_2 替换为 re_1 和（替换之后的）re_2，依次类推。最后，我们将 re_n 中的 def_i（i=1，2，…，n-1）替换为 re_i 的替换后的版本。这些版本中都只包含Σ中的符号。

自从 Kleene 在 20 世纪 50 年代提出了带有基本运算符并、连接和 Kleene 闭包的正则表达式之后，已经出现了很多针对正则表达式的扩展，它们被用来增强正则表达式描述串模式的能力。在这里，我们介绍一些早期的可以用在词法分析器规约中的扩展表示法，如表 2.4 所示：

表 2.4 单词的正则表达式示例[1]

正则表达式	符号意义
if	匹配“if”文本
[a–z][a–z0–9]*	标识符
[0–9]+	自然数
([0–9]+“.”[0–9]*)|([0–9]*“.”[0–9]+)	实数

[1] 该例子来源于：《现代编译原理》，Andrew W.Appel等著，赵克佳等译，人民邮电出版社，第13页，2006年。

(“ - - ” [a – z] * “ \n ”) | (“ ” | “ \n ” | “ \t ”) +	空白/注释
.	报错

（1）运算符+表示前一个元素出现一次或多次，例如，ab+c 匹配“abc”，“abbc”，“abbbc”，但不匹配 ac；

（2）运算符?表示前一个元素出现零次或一次，例如，ab?c 匹配“ac”和“abc”；

（3）运算符*表示前面的元素出现零次或多次。例如，ab*c 匹配“ac”，“abc”，“abbc”，“abbbc”等；

（4）一个正则表达式 $c_1|c_2|...|c_n$ 可以缩写为$[c_1c_2...c_n]$。如果 c_1、c_2、...、c_n 形成一个逻辑上连续的序列，比如大写字母、小写字母或数字时，我们可以将它们表示成 c_1-c_n。我们只需要写出第一个和最后一个符号，中间用连字符隔开。因此，我们可以用[xyz]作为 x|y|z 的缩写，而用[a-z]作为 a|b|...|z 的缩写。

2.1.3 有限状态自动机

正则表达式可以用来表示一个模式，我们可以将这个模式转换成具有特定风格的流图，称为“状态转移图”。**状态转移图（State Transition Diagram）**有一组称为**“状态”（State）**的节点或圆圈。词法分析器在扫描输入串的过程中寻找与某个模式匹配的词素，而转换图中的每一个状态代表一个可能在这个过程中出现的情况。状态图中的**边（Edge）**从图的一个状态指向另一个状态，每条边上的标号包含了一个或多个符号。对于某个状态 s，如果下一个输入符号是 x，那么我们应该寻找一条从 s 离开且标号为 x 的边。如果找到这样的一条边，那么就可以进入状态转换图中该边所指的状态，如图 2. 2 所示。一些关于状态转换图的重要约定如下。

（1）有一个状态被指定为**开始状态**，也称为**初始状态**，该状态由一条没有入边的标号为“start”的边指明。在读入任何输入符号之前，状态转换图总是位于它的开始状态。

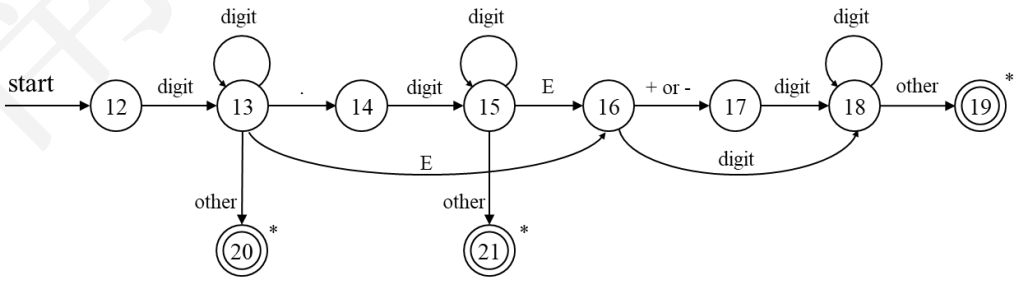

图 2.2 无符号数字的状态转移图

（2）某些状态称为**接受状态**或者**最终状态**，这些状态表明已经找到了一个词素。我们用双层的圈表示一个接受状态，并且如果该状态要执行一个动作（通常是向语法分析器返回一个词法单元和相关属性值），那么我们将把这个动作附加到该接受状态上。

（3）如果需要回退一个位置（即相应的词素并不包含那个在最后一步使我们到达接受状态的符号），那么我们将在该接受状态的附近加上一个*。

状态转移图可以识别正则表达式表示的模式，在实际中，我们在词法分析过程中使用一种与状态转移图类似的图，被称为**有限状态自动机（Finite State Automata）**。但是有限状态自动机和状态转移图之间也有不同：

（1）有限状态自动机是**识别器（Recognizer）**，它们只能对每个可能的输入串简单地回答“是”或“否”。

（2）有限状态自动机分为两类：

(a) **非确定性有限状态自动机（Nondeterministic Finite Automata，NFA）**对其中的边没有任何限制，可以存在以同一个符号作为标号的离开同一个状态的多条边，并且空串 ε 也可以作为离开状态的边，如图 2.3 所示。

(b) **确定性有限状态自动机（Deterministic Finite Automata，DFA）**对其中的边有较为严格的限制，对于其中的每个状态及自动机输入字母表中的每个符号而言，有且只有一条以该符号为标号的边离开该状态；且 DFA 中不存在空串 ε 作为标号的边。

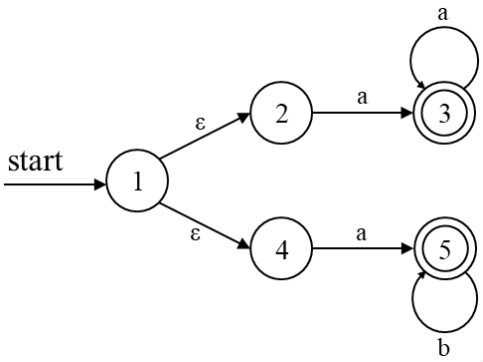

图 2.3 接受 aa* | bb*的 NFA

理论上讲，DFA 是 NFA 的一个特例，我们可以看到，与 NFA 相比，DFA 具有如下特点：：

（1）没有输入 ε 之上的转换动作。

（2）对于每一个状态 *s* 和每个输入符号 *x*，有且仅有一条标号为 *x* 的边离开状态 *s*。

DFA 与 NFA 能够识别的语言集合是相同的。事实上，这些语言的集合正好是能够用正则表达式描述的语言的集合。这个集合称为**正则集合（Regular Set）**，而其对应的语言，即正则语言。

一般地，一个 NFA 由以下几部分组成：

（1）一个有限状态的集合 *S*。

（2）一个输入符号集合 Σ，即**输入字母表（Input Alphabet）**。我们假设代表空串的 ε 不是 *Σ*中的元素。

（3）一个**转换函数（Transition Function）**，它为每个状态和 Σ ∪ {ε}中的每个符号都给出了相应的**后续状态（Next State）**的集合。

（4）*S* 中的一个状态 s_0 被指定为**开始状态**，或者说**初始状态**。

（5）*S* 的一个子集 *F* 被指定为**接受状态**（或者说**最终状态**）集合。

不管是 NFA 还是 DFA，我们都可以将其表示为一张**转换图（Transition Graph）**。图中的节点是状态，带有标号的边表示自动机的转换函数。从状态 *s* 到状态 *t* 存在一条标号为 *a* 的边当且仅当状态 *t* 是状态 *s* 在输入 *a* 上的后继状态之一。这个图和我们前面介绍的状态转移图十分类似，其不同点在于：

（1）同一个符号可以标记从同一状态触发到达多个目标状态的多条边；

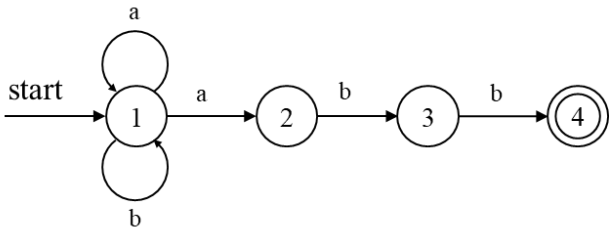

图 2.4 接受(a|b)*abb 的 NFA

表 2.5 对应于图 2.4 的 NFA 的转换表

状态	a	b	ε
1	{1,2}	{1}	∅
2	∅	{3}	∅
3	∅	{4}	∅
4	∅	∅	∅

（2）一条边的标号不仅可以是输入字母表中的符号，也可以是空符号串 ε。

我们也可以将一个 NFA 表示为一张转换表（Transition Table），表的各行对应于各个状态，各列对应于输入符号和 ε。对应于一个给定状态和给定输入的表项是将 NFA 的转换函数应用于这些参数后得到的值。如果转换函数没有给出对应于某个状态—输入对的信息，我们就把∅放入对应的表项中。

表 2.5 列出了图 2.4 中的 NFA 的转换表。转换表的优点是我们很容易确定与一个给定状态和一个输入符号相对应的转换。但是它的缺点是：如果输入字母表很大，且部分状态在大多数输入字符上没有转换时，转换表需要占用大量空间。一个 NFA 接受（Accept）输入字符串 x，当且仅当对应的转换图中存在一条从开始状态到某个接受状态的路径，使得该路径中各条边上的标号组成字符串 x。路径中的 ε 标号将被忽略，因为空串不会影响到根据路径构建得到的符号串。由一个 NFA 定义（或接受）的语言是从开始状态到某个接受状态的所有 路径上的标号串的集合，如图 2.5 所示。

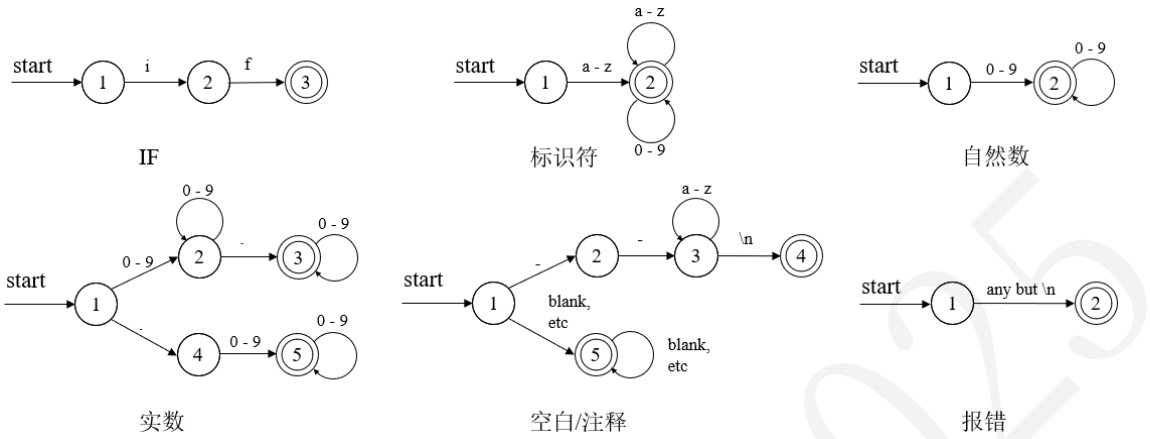

图 2.5 一些词法单元的有限状态自动机示例[1]

如果我们使用转换表来表示一个 DFA，那么表中每一个表项就是一个状态。因此，我们可以不使用花括号，直接写出这个状态，因为花括号只是用来说明表项内容是一个集合。NFA 抽象地表示了用来识别某个语言中的串的算法，而相应的 DFA 则是一个简单具体的识别串的算法。在构造词法分析器的时候，我们真正实现或者模拟的是 DFA。每个正则表达式和每个 NFA 都可以被转换成为一个接受相同语言的 DFA。

我们注意到，图 2.5 中展示的六个有限状态机均是独立的，而我们在通常是需要把它们合并为一个有限自动机从而实现词法分析器。我们将中关于 IF、标识符、自然数以及报错的表达式都转换成 NFA，然后合并这些 NFA，将每一个表达式的开始汇合成一个新的初始结点，将表达式的结束用不同单词类型标记成终态结点，得到的结果如图 2.6 所示。也就是说，我们设置一个初始结点，即图 2.6 中的结点 1，该结点为所有表达式的公共初始结点，当初始结点接收到不同的输入时会进入不同的表达式分支，直到不再接受新的输入时表达式结束，到达 NFA 的终态结点，即图中标位 IF、标识符、自然数以及报错对应的结点 3、8、13 和 15。

[1] 该例子来源于: 《现代编译原理》，Andrew W.Appel等著，赵克佳等译，人民邮电出版社，第15页，2006年。

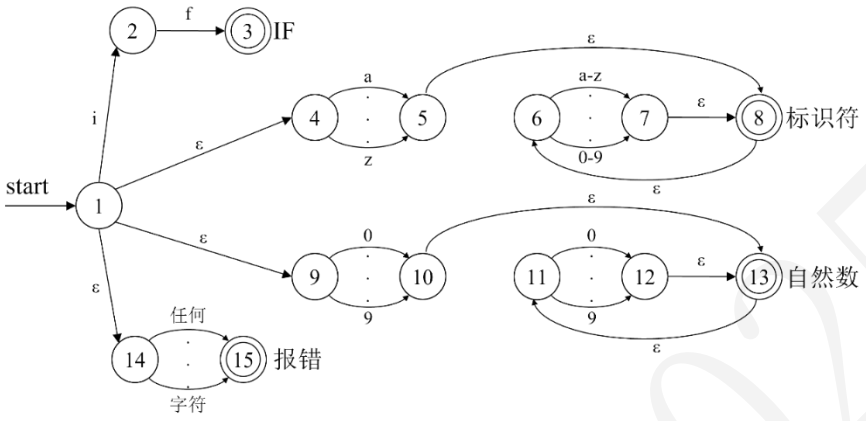

图 2.6 基于图 2.5 中 IF，标识符，自然数以及报错四个词法单元的有限状态自动机合并而成的 NFA

2.1.4 从 NFA 到 DFA 的转换

当前，词法分析器的实现一般模拟 DFA 或 NFA 的执行。由于 NFA 对于一个输入符号可以选择不同的状态转换，使得需要其他的信息以明确转换到的状态。同时，它还可以支持输入 ε 上的转换，因此对 NFA 模拟更加繁琐，不如对 DFA 的模拟更加简洁。因此，通常情况下，对 DFA 的模拟较为常见。然而，从逻辑上，我们构造一个 NFA 会更加直观便捷，所以，我们一般是先根据正则表达式从构建 NFA，然后将 NFA 转换为一个识别相同语言的 DFA。

子集构造法（Subset Construction）可用于将 NFA 转换为 DFA，其基本思想较为简明：让构造得到的 DFA 的每个状态对应于 NFA 的一个状态集合，DFA 再读入 $c_1c_2\cdots c_n$ 之后到达的状态分别对应于相应 NFA 从开始状态出发，沿着以 $c_1c_2\cdots c_n$ 为标号的路径能够到达的状态的集合。

一般情况下，识别相同语言的 DFA 的状态数有可能是 NFA 状态数的指数倍，在这种情况下，我们试图实现 DFA 时会十分困难。然而，基于自动机的词法分析方法的处理能力源于如下的事实：对于一个真实的语言，它的 NFA 和 DFA 的状态数量大致相同，状态数量呈指数关系的情形尚未在实践中出现过。因此，我们可以采用**子集构造**算法实现从 NFA 到 DFA 的转换。

由 NFA 构造 DFA 的**子集构造**算法包含如下内容：

输入：一个 NFA *N*。

输出：一个接受同样语言的 DFA *D*。

方法：我们的算法为 D 构造一个转换表 D_{tran}。D 的每一个状态是一个 NFA 状态集合，我们将构造 D_{tran}，使得 D 可以并行地模拟 N 在遇到一个给定输入串时可能执行的所有动作。我们面对的第一个问题是正确处理 N 的 ε 转换。在表 2.6 可以看到一些函数的定义。这些函数描述了一些需要在这个算法中执行的 N 的状态集上的基本操作。其中，s 表示 N 的单个状态，而 T 代表 N 的一个状态子集。

表 2.6 NFA 状态集上的操作

操作	描述
ε-closure(s)	能够从 NFA 的状态 s 开始只通过 ε 转换到达的 NFA 状态集合
ε-closure(T)	能够从 T 中某个 NFA 状态 s 开始只通过 ε 转换到达的 NFA 状态集合，即 $\cup_{s\in T}$ ε-closure(s)
move(T, a)	能够从 T 中某个状态 s 出发通过标号为 a 的转换到达的 NFA 状态的集合

我们必须找出当 N 读入了某个输入串之后可能位于的所有状态的集合。首先，我们定义 s_0 是 N 的开始状态，在读入第一个输入符号之前，N 可以位于集合 ε-closure(s_0)中的任何状态上。下面我们进行归纳，假定 N 在读入输入串 x 之后可以位于集合 T 中的任何状态上。如果下一个输入符号是 a，那么 N 可以移动到集合 move(T, a)中的任何状态。然而，N 可以在读入 a 后再执行 ε 转换，因此 N 在读入 xa 之后可位于 ε-closure(move(T, a))中的任何状态上。根据这个思想，我们可以得到图 2.7 显示的算法，该方法构造了 D 的状态集合 D_{states} 和 D 的转换函数 D_{tran}。

D 的开始状态是 ε-closure(s_0)，D 的接受状态是所有至少包含了 N 的一个接受状态的状态集合。我们只需要解决对 NFA 的任何状态集合 T 计算 ε-closure(T)，就可以获得完整的子集。这个计算过程如图 2.7 所示，其基本思想是将 NFA 的每个状态看作 DFA 的一个状态，NFA 的每个状态集合看作 DFA 的一个状态集合，并根据 NFA 的转移函数构造 DFA 的转移函数。图 2.8 对应于图 2.6 中的 NFA 经过我们的转换方法后得到后的 DFA。

```
开始时，ε-closure(s0)是 Dstates 中的唯一状态，且它未加标记;
while (在 Dstates 中有一个未标记状态 T) {
    给 T 加上标记;
    for (每个输入符号 a) {
    U = ε-closure(move(T, a));
    if (U 不在 Dstates 中)
        将 U 加入到 Dstates 中，且不加标记;
        Dtran[T, a] = U;
    }
}
```

图 2.7 子集构造算法

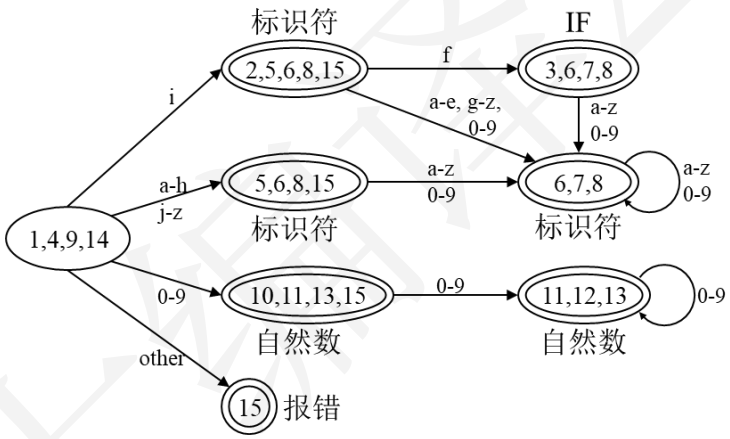

图 2.8 基于图 2.6 所示 NFA 转换成为的 DFA

2.1.5 状态最小化算法

对于同一个语言，可以存在多个识别此语言的 DFA。这些 DFA 状态的名字和个数可能都不相同。如果使用 DFA 来实现词法分析器，我们总是希望使用的 DFA 的状态数量尽可能少，因为描述词法分析器的转换表需要为每个状态分配条目。状态的名字是次要的，如果我们只需要改变状态名字就可以将一个自动机转换为另一个自动机，我们称这两个自动机是**同构**的。实际上，任何正则语言都有一个唯一的（不计同构）状态数目最少的 DFA。而且，从任意一个接受相同语言的 DFA 出发，通过分组合并等价的状态，我们总是可以构建得到这个状态数最少的 DFA。接下

来，我们将给出一个将任意 DFA 转化为等价的状态最少的 DFA 的算法。该算法首先创建输入 DFA 的状态结合的划分。为了理解这个算法，我们需要了解输入串时如何区分各个状态的。如果分别从状态 s 和 t 出发，沿着标号为 x 的路径到达的两个状态中只有一个是接受状态，我们就说串 x 区分了状态 s 和 t。如果存在某个能够区分状态 s 和 t 的串，那么它们就是**可区分的（Distinguishable）**。

DFA 状态最小化算法的工作原理是将一个 DFA 的状态集合划分成多个组，每个组中的各个状态之间不可区分，然后将每个组中的状态合并成状态最少的 DFA。算法在执行过程中维护了状态集合的一个划分，划分中的每个组内的各个状态不能区分，但是来自不同组的任意两个状态是可以区分的。当任意一个组都不能再被分解为更小的组时，这个划分就不能再进一步简化，此时我们得到了状态最少的 DFA。

具体而言，我们首先将状态划分为两个组，即接受状态组和非接受状态组。算法的基本步骤是从当前划分中选一个状态组，比如 $A=\{s_1, s_2, \ldots, s_k\}$，并选定某个输入符号 a，检查 a 是否可以用于区分 A 中的某些状态。我们检查 $s_1, s_2, \ldots, s_k$ 在 a 上的转换，如果这些转换到达的状态落入当前划分的两个或者多个组中，我们就将 A 分割成为多个组。我们认为两个状态 s_i 和 s_j 在同一组中当且仅当它们在 a 上的转换都能到达同一个组的状态。我们重复这个分割过程，直到无法根据某个输入符号对任意组进行分割为止。我们将这个算法思想描述在下面的算法中：

输入：一个 DFA D，其状态集合为 S，输入字母表为 Σ，开始状态为 s_0，接受状态集为 F。

输出：一个 DFA D'，它和 D 接受相同的语言，且状态数最少。

方法：

（1）构造包含两个组 F 和 $S-F$ 的初始划分 Π，这两个组分别是 D 的接受状态组和非接受状态组；

（2）应用图 2.9 中的过程，构造新的划分 Π_{new}；

> 最初，令 $\Pi_{new}=\Pi$;
> for (Π中每一个组 G) {
> 将 G 划分为更小的组，两个状态 s 和 t 在同一个小组中当且仅当对于所有的输入符号 a，状态 s 和 t 在 a 上的转换都到达 Π中的同一组;
> /* 在最坏的情况下，每个状态各自组成一个组 */
> 在 Π_{new} 中将 G 替换为对 G 进行划分得到的小组;
> }

图 2. 9 DFA 状态最小化过程中新划分 Π_{new} 的构造算法

（3）如果 $\Pi_{new}=\Pi$，令 $\Pi_{final}=\Pi$ 并执行步骤 4；否则用 Π_{new} 替换 Π 并重复步骤 2；

（4）在划分 Π_{final} 的每个组中选取一个状态作为该组的代表。这些代表构成了状态最少的 DFA D'的状态。D'的其它部分按照如下步骤构建：

(a) D'的开始状态是包含了 D 的开始状态的组的代表；

(b) D'的接受状态是那些包含了 D 的接受状态的组的代表。其中，每个组要么只包含接受状态，要么只包含非接受状态，因为我们一开始就将这两类状态分开了，而图 2.9 中的过程总是通过分解已经构造得到的组来得到新的组；

令 s 是 Π_{final} 中某个组 G 的代表，并令 D 中在输入 a 上离开 s 的转换到达状态 t。令 r 为 t 所在组 H 的代表。那么在 D'中存在一个从 s 到 r 在输入 a 上的转换。在 D 中，组 G 中的每一个状态必然在输入 a 上进入组 H 中的某个状态，否则组 G 应该已经被图 2.9 的过程分割成更小的组了。

2.1.6 语法分析概要

在常见的编译器模型中，语法分析器从词法分析器中获得一个由词法单元组成的序列，并验证这个序列符合源语言约定的文法。如图 2.10 所示，我们期望语法分析器能够以易于理解的方式报告语法错误，并且能够从常见的错误中恢复并处理程序的其余部分。从概念上讲，如果认为一个程序不含有语法错误或可诊断的语义错误，我们则认为该程序是**良构的程序**。对于良构的程序，语法分析器能够构造出一棵语法分析树，并把它传递到编译器的其它部分进一步处理。但实际上，

我们并不需要显式地构造这棵语法分析树，因为在实践中，对源程序的检查和翻译动作可以与语法分析过程交错完成。因此，语法分析器和编译器前端的其它部分可以用同一个模块来实现。

编译器中常用的语法分析方法可以分为自顶向下和自底向上的。顾名思义， 自顶向下的 方法从语法分析树的顶部（根节点）开始向底部（叶子节点）构造语法分析树，而自底向上 的方法则从叶子节点开始，逐渐向根节点方向构造。在这两种分析方法中，语法分析器的输 入总是按照从左到右的方式被扫描，每次扫描一个符号。

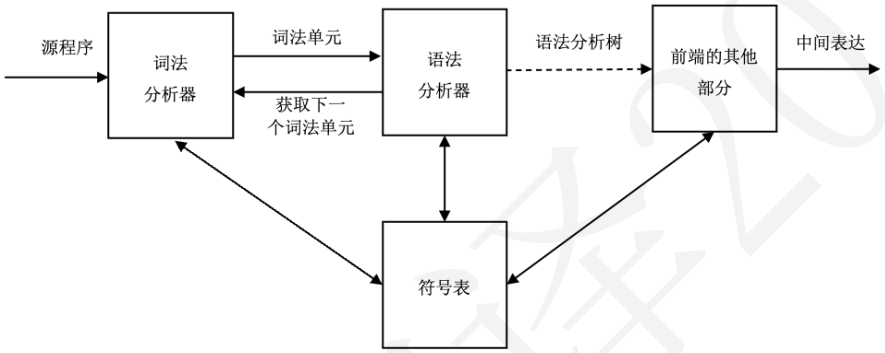

图 2. 10 编译器工作流程中语法分析器的位置

在语法分析中，自顶向下和自底向上是两种常见的方法。不过它们只能处理某些文法子类，如 LL 和 LR 文法。不过这两个子类的表达能力已经足够描述现代程序设计语言的大部分语法构造了。一般手工实现的语法分析器会选择自顶向下的分析方法，如预测语法分析方法，用以处理 LL 文法。而处理较为复杂的 LR 文法类的语法分析器则通常使用自动化工具进行构造。在本章中我们假设语法分析器的输出是语法分析树的某种表现形式，该语法分析树对应于来自词法分析器的词法单元流。

示例文法介绍

在本节中我们将给出一些文法示例[1]，这些示例主要关注表达式。因为运算符的结合性和优先级的原因，表达式的处理更具有挑战性。

[1] 这些例子来源于: 《编译原理》，Alfred V. Aho等著，赵建华等译，机械工业出版社，第122页，2009年。

以下文法指明了运算符的结合性和优先级。*E* 表示一组以+号分隔的项所组成的表达式，*T* 表示由一组以*号分隔的因子所组成的项，而 *F* 表示因子，它可能是括号括起来的表达式或标识符：

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

该文法属于 LR 文法类，适用于自底向上的语法分析技术。这个文法经过修改可以处理更多的运算符和更多的优先级层次。然而，它不能用于自顶向下的语法分析，因为它是左递归的。我们将在本章后续的部分介绍左递归的消除技术。在此为了讲解方便，我们直接给出该表达式文法的无左递归版本，该版本将被用于自顶向下的语法分析：

$E \rightarrow TE'$
$E' \rightarrow +TE \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \varepsilon$
$F \rightarrow (E) \mid id$

文法错误的处理

现代编译器实现中采用的 LL 和 LR 语法分析方法的一个优点是能够在第一时间精准地发现语法错误，也就是说，当来自词法分析器的词法单元流不能根据该源语言的文法进一步分析时，语法分析器就会发现错误。更精确地讲，一旦它们发现输入的某个前缀不能够通过添加一些符号而形成这个语言的串，就可以立刻检测到语法错误。语法分析器中的错误处理程序的目标说起来很简单，但实现起来却很有挑战性，具体而言，我们需要：清晰精确地报告出现的错误；能很快地从各个错误中恢复，以继续检测后面的错误；尽可能少地增加处理正确程序时的开销。

幸运的是，常见的语法错误都很简单，使用相对直接的错误处理机制就足以达到目标。实践中，一个常用而有效的策略就是打印出有问题的那一行，然后用一个指针指向检测到错误的地方。

另外一个需要考虑的问题是，当检测到一个错误时，语法分析器应该如何恢复呢？虽然还没有一个普遍接受的策略，但有一些方法的适用范围很广。其中最简单的方法是让语法分析器在检测到第一个错误时给出错误提示信息，然后退出。如果语法分析器能够把自己恢复到某个状态，且有理由预期从那里开始继续处理输入将提供有意义的诊断信息，那么它通常会发现更多的错误。

但是如果错误太多，那么最好让编译器在超过某个错误数量上界之后停止分析。这样做要比让编译器产生大量恼人的“可疑”错误信息更有用。

2.1.7 上下文无关文法

在形式语言的理论研究中，文法是一种用于描述程序设计语言语法的表示方法，可以表示大多数程序设计语言的层次化语法结构。文法定义了语言中字符串的产生式规则，这些规则描述了如何用语言的字母表生成符合语言语法的字符串。当前，学界通常将形式语言的文法分为四类，包括无限制文法、上下文有关文法、上下文无关文法和正则文法。现代几乎所有程序设计语言都是通过上下文无关文法来定义的，主要原因在于上下文无关文法拥有足够强的表达力来表示大多数程序设计语言的语法，又提供了足够简明的机制使得我们可以构造高效的检验算法来判定字符串的有效性。

一个**上下文无关文法（Context-Free Grammar，CFG）**通常由四个元素组成：

- **终结符号**集合，也被称为“**词法单元**”集合，包含了该文法定义的语言基本符号。当前常见的文法中一般使用小写字母、运算符号、标点符号和数字等表示终结符号。
- **非终结符号**集合，也被称为“**语法变量**”集合，包含了该文法定义的非终结符号。每一个非终结符号表示一个终结符号串的集合，用于定义由文法生成的程序设计语言。当前常见的文法中一般使用大写字母和小写的希腊字母等表示非终结符号。
- **产生式**集合，包含了该文法定义的产生式，产生式主要用于表示某个构造的某种书写形式。一个产生式包含一个产生式头（或者称为**左部**）的非终结符号，推导符（通常通过箭头表示），和一个产生式体（或者称为**右部**）的由终结符号和非终结符号组成的序列。
- **开始符号**，通常是一个非终结符号，出现在第一个产生式的头部。开始符号表示的串集合就是文法生成的语言。

表 2.7 一个程序的文法样例[1]

文法	**描述**
$S \rightarrow E$	S 是文法开始符号。

[1] 该例子来源于: 《现代编译原理》，Andrew W.Appel等著，赵克佳等译，人民邮电出版社，第29页，2006年。

$S \rightarrow id := E$	id 是文法终结符号，E 是文法非终结符号。
$S \rightarrow print\ (\ L\ )$	print ( )是文法终结符号，L 是文法非终结符号。
$E \rightarrow id$	id 表示直线式程序的标识符。
$E \rightarrow num$	num 是文法终结符号，表示数字。
$E \rightarrow E+E$	+是文法终结符号，产生式体表示程序中的加法语句。
$E \rightarrow (S,E)$	,是文法终结符号。
$L \rightarrow E$	产生式体表示 L 可以用 E 替换。
$L \rightarrow L\ ,E$	产生式体表示用逗号分割的 L 和 E 表示的终结符号。

一个直线式程序文法示例如表 2.7 所示，此例中的终结符号集合为：{id, print, num , +, (,), :=, ;}，非终结符集合为：{*S*, *E*, *L*}，开始符号是 *S*。文法包含了终结符号和非终结符号构成的九条产生式。

对于产生式 $S \rightarrow E$，其中 *S* 是文法 *G* 的开始符号，*E* 是 *G* 的一个**句型（Sentential Form）**。句型可能同时包含终结符号和非终结符号，也可能是空串。文法 G 的一个**句子（Sentence）**是不包含非终结符号的句型。文法能够表示的所有句子的集合就是一个文法生成的语言。可以由上下文无关文法生成的语言被称为**上下文无关语言（Context-Free Language）**。如果两个文法生成的语言相同，我们就可以认为这两个文法等价。

在语法分析过程中，如果将产生式看作是重写规则，即不断地将某个非终结符号替换为该非终结符号的某个产生式的体，则可以从推导的角度描述构造语法分析树的方法。语法分析树是推导过程的图形化表示，它忽略了推导过程中对非终结符号应用产生式的顺序。语法分析树中每个内部节点表示一个产生式的应用，内部节点的编号是此产生式中的非终结符号。这个节点的子节点标号从左到右组成了在推导过程中替换这个非终结符的产生式体，如图 2.11 所示。

整个推导过程，就是从开始符号出发，每一个重写步骤把一个非终结符号替换为它的某个产生式的体。比如对于表 2.7 中的产生式 $E \rightarrow id$ 和 $E \rightarrow E+E$，重写步骤可以将 *E* 替换为产生式体中的 *id*，因此就可以生成 *id* + *id*。推导的思想对应于自顶向下构造语法分析树的过程，但是其概念中所包含的精确性也可以用于讨论自底向上的语法分析过程。自底向上语法分析与“最右”推导相关，此推导每一步重写的都是最右边的非终结符号。为了给出推导的一般性定义，考虑一个文法符号序列中间的非终结符号 *A*，比如 αAβ，其中 α 和 β 表示任意的文法符号串。对于产生式 A→ω，可以写作 αAβ⇒αωβ，其中⇒表示“一步推导出”。当一个推导序列 $\beta_1 \Rightarrow \beta_2 \Rightarrow \cdots \Rightarrow \beta_n$ 将 β_1 替换为 β_n，

则认为是从 β_1 推导出 β_n。更一般地，对于经过“零步或者多步推导出”，可以使用符号$^*\Rightarrow$表示这种推导关系。因此，可以得出：

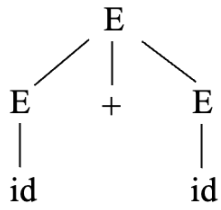

图 2. 11 id+id 的语法分析树

对于任意串 α，α⇒α，并且 α⇒β 且 β⇒ω，那么 α⇒ω。类似地，符号$^+\Rightarrow$还可以表示“一步或者多步推导出”关系。对于给定的产生式集合，推断出其生成的特定的语言是十分有用的。当研究一个复杂的构造时，可以写出该结构的一个简洁、抽象的文法，并研究其生成的语言。

如果一个文法能够推导出具有两棵不同语法树的句子，则该文法是**二义性的（Ambiguous）**。例如，表 2. 7 中所示文法就是具有二义性的，对于句子 *id := id + id + id*，其可以推导出两棵语法树，如图 2.12 所示。大部分语法分析器都要求文法是无二义性的，否则不能唯一确定一个句子的语法分析树。

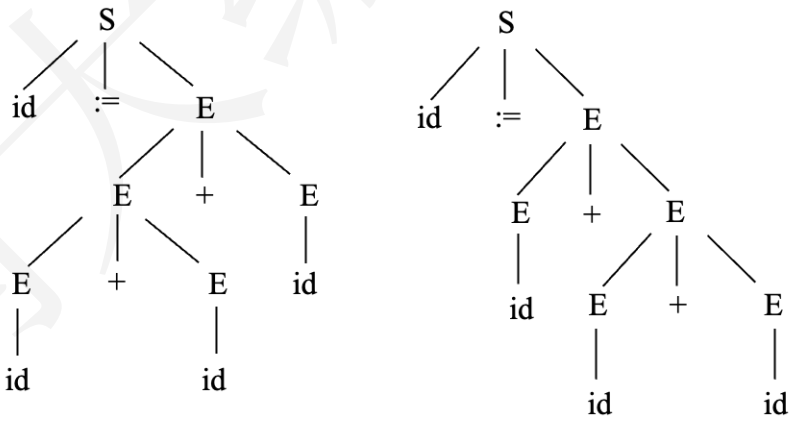

图 2. 12 id := id + id + id 的两棵语法树

2.1.8 自顶向下的语法分析算法

由语法分析树的根节点开始进行语法分析的方法称为**自顶向下的语法分析**（Top-down Parsing）算法。相较于后文所述的自底向上的语法分析，自顶向下的语法分析较为直观简单，便于理解。自顶向下的语法分析可以看成是寻找输入的语法串的最左推导（总是推导符号串的最左非终结符）的过程，以下是一个自顶向下的语法分析的示例：

对于文法：

$S \rightarrow aB$

$S \rightarrow g$

$B \rightarrow cBd$

$B \rightarrow t$

那么，当我们输入串 acctdd，自顶向下的推导过程如图 2.13 所示：

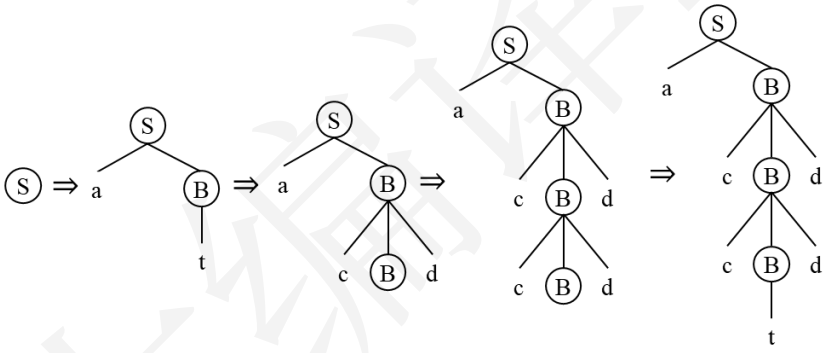

图 2. 13 串 acctdd 自顶向下的推导过程

在一个自顶向下的语法分析的过程中，关键的问题是对于一个非终结符，我们应当采用哪一种产生式，使得剩余的字符串能够与产生式的终结符相匹配。自顶向下的语法分析可以使用一种称为**递归下降**的语法分析算法进行推导。

递归下降的语法分析

在一些文法中，采取了**递归下降（Recursive Descent）**的语法分析算法，进行文法产生式的推导。递归下降算法将每一个文法产生式都转化为递归函数的一个子句。但在实际转化时会经常遇到冲突，不知道该使用哪个产生式。例如对图 2.13 中的节点 B 进行转化时，会面临对产生式 $B \rightarrow cBd$ 和 $B \rightarrow t$ 的抉择。为了避免抉择失败而反复回溯所产生的时间和空间成本，我们通常采用

一种基于**预测的递归下降**的语法分析，该方法通常采用 First 和 Follow 两种函数来实现。两种函数在产生式上运算生成的 First 集合和 Follow 集合，使得我们可以根据下一个输入的符号来辅助选择应用哪一个产生式，充分利用了每个表达式第一个终结符号所提供的信息。

（1）First 函数：First(x)被定义为当前的符号为 x 时，所有可以推导得到的串的第一个终结符的集合。如果 x 也可以推导得到ε，那么将ε也加入到 First(x)中。

（2）Follow 函数：Follow(x)被定义为当前的符号为 x 时，所有可直接跟随于 x 后的终结符的集合。

我们可以简单地举一个计算 First 和 Follow 函数的例子，对于先前所述的文法构造 First 集合与 Follow 集合（表 2. 8）：

表 2. 8 First 与 Follow 集合的构造

	S	*B*
Nullable（可空）	no	no
First		
Follow		

在每次迭代式地读取产生式后，我们可以获得以下的结果（表 2. 9）：

表 2. 9 迭代式地读取结果

	S	*B*
Nullable（可空）	no	no
First	a, g	c, t, $
Follow		d, $

基于 First 和 Follow 函数，我们可以生成预测分析表，并且根据这张预测分析表构造一个预测分析器，用于辅助递归向下的语法分析器进行语法分析。例如，我们可以基于上述的这张表构造预测分析表如下（表 2. 10）：

表 2. 10 预测分析表

	a	c	d	g	t
S	S→a*B*			*S*→g	
B		B→c*B*d			*B*→t

这张表的首行和首列分别由非终结符集合 X 和终结符集合 Y 构成。在构造时，对于每个 $T\in \mathrm{First}(\gamma)$，在表的第 X 行第 T 列，填入产生式 $X\to\gamma$。此外，如果 γ 是可空的，则对于每个 $T\in \mathrm{Follow}(X)$，在表的第 X 行第 T 列，也填入该产生式。

使用这张预测分析表，我们可以完成对输入 acctdd 的如下推导（见表 2-11）：

表 2.11 对输入 acctdd 进行预测分析时执行的推导步骤

已匹配	栈	输入	动作
	S$	acctdd$	
	a*B*$	acctdd$	输出 $S\to$a*B*
a	*B*$	cctdd$	匹配 a
a	c*B*d$	cctdd$	输出 $B\to$c*B*d
ac	*B*d$	ctdd$	匹配 c
ac	c*B*dd$	ctdd$	输出 $B\to$c*B*d
acc	*B*dd$	tdd$	匹配 c
acc	tdd$	tdd$	输出 $B\to$t
acct	dd$	dd$	匹配 t
acctd	d$	d$	匹配 d
acctdd	$	$	匹配 d

消除左递归

在部分预测分析表中，可能存在双重定义的情况，即预测分析表的某个单元格可能出现不止一个产生式。在这种情况下，当前的预测无法确定需要在多个产生式中选取哪一个。以如下的文法为例：

$E \to E+T$

$E \to T$

这样的一个文法能够使得任何属于 First(T)的符号同时也属于 First(E+T)，这种情况被称为**存在左递归**。存在左递归会使得我们的语法分析在推导时无法确定应当选择的产生式。

如果我们有左递归文法：

$A \to A$a | b

使用若干次 *A*a 替换 *A*，则可推导出句型：Aaaaaa...，而要推导出一个有限长度的句子，我们一定会使用产生式 *A* → b，因此生成的句子必然以 b 开头。

因此我们可以试图用产生式 *A* → b*A*'来替代原先的句子。

文法可被改写为：

A → b*A*'

A' → a*A*'| ε

从而将左递归消除，这种简单的改写规则已足以处理大多数文法。

LL(1)语法分析技术

LL(1)语法分析技术是一种不需要回溯的递归下降语法分析技术；其中，第一个 L 表示从左到右地扫描输入，第二个 L 表示产生最左推导，1 则表示每次推导中只向前看一个符号来决定分析动作。LL(1) 语法分析技术是 LL(*k*)文法分析技术的一个特例。理论上，*k* 可以设定为任意正整数，即使得我们在进行语法分析的时候，可以向前看 *k* 个符号来决定对产生式的选择；但是在实践中，绝大多数文法，设定 *k* 为 1 就可以确定分析过程中选择哪个产生式用以推导。

能够使用 LL(1)语法分析技术进行确定的自顶向下分析的文法需要满足以下条件：

对于所有的非终结符 *A* 和 *B*，若存在 *S*→*A* | *B*，则

（1）First(*A*)∩First(*B*) = ϕ;

（2）如果 *A* 能够推导出ε，那么 First(*B*)∩Follow(*A*) = ϕ。

满足以上条件，只向前看一个符号，也不会同时存在多个满足条件的产生式。

在满足条件后，基于上文所述计算 First 集合与 Follow 集合，构造预测分析表，然后基于预测分析表能够构造递归向下的语法分析算法，进行高效的语法分析与推导。

2.1.9 自底向上的语法分析算法

一个自底向上的语法分析过程对应于为一个输入串构造语法分析树的过程，它从叶子节点（底部）开始逐渐向上到达根节点（顶部）。将语法分析描述为语法分析树的构造过程会比较方便，虽然编译器前端实际上不会显式地构造出语法分析树，而是直接进行翻译。图 2.14 的分析树构造过程演示了按照下列文法对词法单元序列 id*id 进行自底向上语法分析的过程：

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow (E) \mid \mathrm{id}$

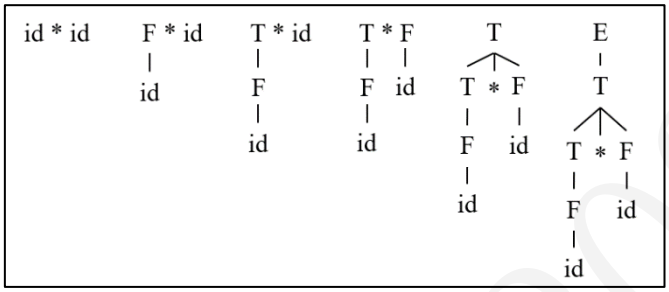

图 2. 14 id*id 自底向上的分析过程

本节将介绍一个基于移入-规约语法分析的自底向上语法分析方法，即LR分析方法，并构造出高效的LR分析器。

LR 语法分析方法

LR(*k*)中的L、R、*k*分别表示从左至右分析、最右推导、向前查看*k*个单词（Left-to-right Parse、Rightmost-Derivation、k-token Lookhead）。从直觉上看，使用最右推导似乎有些奇怪，它是如何与从左至右的分析过程保持一致呢？ 表2. 12举例[1]说明了使用表2. 7中的文法（增加了一个新的产生式*S*'→*S*$）对下面这个程序进行的LR分析：

```
a := 7 ;
b := c + (d := 5 + 6, d)
```

表 2. 12 对句子进行移入-规约分析的过程，栈列的数字下标是 DFA 的状态符号（见表 2. 13）

栈	输入	动作
|	a:=7;b:=c+(d:=5+6,d)$	移入
| id_4	:=7;b:=c+(d:=5+6,d)$	移入
| id_4 :=$_6$	7;b:=c+(d:=5+6,d)$	移入
| id_4 :=$_6$ num_{10}	;b:=c+(d:=5+6,d)$	归约 $E \rightarrow \mathrm{num}$
| id_4 :=$_6$ E_{11}	;b:=c+(d:=5+6,d)$	归约 $S \rightarrow \mathrm{id} := E$
| S_2	;b:=c+(d:=5+6,d)$	移入

[1] 该例子来源于: 《现代编译原理》，Andrew W.Appel等著，赵克佳等译，人民邮电出版社，第39页，2006年。

$\| S_2 ;_3$	b:=c+(d:=5+6,d)$	移入
$\| S_2 ;_3 id_4$	:=c+(d:=5+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6$	c+(d:=5+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 id_{20}$	+(d:=5+6,d)$	归约 $E \rightarrow \mathrm{id}$
$\| S_2 ;_3 id_4 :=_6 E_{11}$	+(d:=5+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16}$	(d:=5+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8$	d:=5+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4$	:=5+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4 :=_6$	5+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4 :=_6 num_{10}$	+6,d)$	归约 $E \rightarrow \mathrm{num}$
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4 :=_6 E_{11}$	+6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4 :=_6 E_{11} +_{16}$	6,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4 :=_6 E_{11} +_{16} num_{10}$	,d)$	归约 $E \rightarrow \mathrm{num}$
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4 :=_6 E_{11} +_{16} E_7$	,d)$	归约 $S \rightarrow E+E$
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 id_4 :=_6 E_{11}$	,d)$	归约 $\mathrm{S} \rightarrow \mathrm{id} := E$
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 S_{12}$	,d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18}$	d)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18} id_{20}$	)$	归约 $E \rightarrow \mathrm{id}$
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18} E_{21}$	)$	移入
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} (_8 S_{12} ,_{18} E_{21})_{22}$	$	归约 $S \rightarrow (S, E)$
$\| S_2 ;_3 id_4 :=_6 E_{11} +_{16} E_{17}$	$	归约 $S \rightarrow E+E$
$\| S_2 ;_3 id_4 :=_6 E_{11}$	$	归约 $S \rightarrow \mathrm{id} := E$
$\| S_2 ;_3 S_5$	$	归约 $S \rightarrow S;S$
$\| S_2$	$	接受

该分析器有一个栈和一个输入，输入中的前k个单词为提前查看的符号。根据栈的内容和超前查看的符号，分析器将执行移入-归约两种动作：

（1）移入：将第一个输入单词压入至栈顶；

（2）归约：选择一个文法规则$X \rightarrow ABC$，依次从栈顶弹出C、B和A，然后将X压入栈中。

开始时栈为空，分析器位于输入的开始。移进文件终结符S的动作称为**接受**，它导致分析过程成功结束。表2.12列出了在每一个动作之后的栈和输入，也指明了所执行的是什么动作。将栈和输入合并起来形成的一行总是构成一个最右推导。事实上，表2.12自下而上地给出了对输入字符

串的最右推导过程。可以发现，LR语法分析过程也是通过表格驱动的，在这一点与我们前文提到的LL语法分析很相似。

LR 语法分析器

那么LR分析器该如何进行移入和归约呢？答案是通过确定的有穷状态自动机（DFA），这种DFA不是作用于输入（因为有限自动机太弱而不适合上下文无关文法），而是作用于栈。DFA的边是用可以出现在栈中的符号（终结符和非终结符）来标记的。表2.13是前面文法（表2.7）的转换表。

表 2.13 表 2.7 所示文法的 LR 分析表

	id	**num**	**print**	**;**	**,**	**+**	**:=**	**(**	**)**	**$**	***S***	***E***	***L***
1	s4		s7								g2		
2				s3						a			
3	s4		s7								g5		
4							s6						
5				r1	r1					r1			
6	s20	s10						s8				g11	
7								s9					
8	s4		s7								g12		
9												g15	g14
10				r5	r5	r5			r5	r5			
11				r2	r2	s16				r2			
12				s3	s18								
13				r3	r3					r3			
14					s19				s13				
15					r8				r8				
16	s20	s10						s8				g17	
17				r6	r6	s16			r6	r6			
18	s20	s10						s8				g21	
19	s20	s10						s8				g23	
20				r4	r4	r4			r4	r4			
21									s22				
22				r7	r7	r7			r7	r7			
23					r9	s16			r9				

这个转换表中的元素标有下面四种类型的动作：

（1）s*n* 表示移入状态*n*；

（2）g*n* 表示转换到状态*n*；

（3）r*k* 表示用规则*k*归约；

（4）a 表示接收。

如果有错误，就用表中的空项目来表示。

为了使用该表进行分析，要将移入和归约动作看成DFA的边，并查看栈的内容。例如，若栈为id := *E*，则DFA将从状态1依次转换到4、6和11。若下一个输入单词是一个分号，状态11的“;”所在列则指出将根据规则2进行归约，因为文法的第二个规则是*S* → id := *E*。于是栈顶的3个符号被弹出，同时*S*被压入栈顶。

对于每个符号，分析器不是重新扫描栈，而是记住每个栈元素所到达的状态，算法如图2.15所示。

查看栈顶状态和输入符号，从而得到对应的动作：

如果动作是

移进 (n) ：前进至下一个单词，将 n 压入栈；

归约 (k) ：从栈顶依次弹出单词，弹出单词的次数与规则 k 的右部符号个数相同。

令 X 是规则 k 的左部符号;

在栈顶现在所处的状态下，查看 X 得到动作“转换到 n”;

将 n 压入栈顶。

接收：停止分析，报告成功。

错误：停止分析，报告失败。

图 2. 15 LR 分析表的使用算法

更强大的 LR 分析方法：

表 2. 12 对应的分析过程是一种最简单的 LR 分析方法，即 LR(0)分析方法，在使用分析器时无需提前查看输入的前 *N* 个单词。可以使用 LR(0)分析的文法，表达性通常会很弱。对于此类文法，只需查看栈就可以进行分析，它的移入或归约不需要超前查看。但对于其它更加复杂的文法，程序输入在解析到不同的状态时会面临选择冲突。例如在处理某个符号时，分析器既可以选择归约到一个状态，也可以选择移入另一个状态。此时需要一些处理能力更强大的分析算法，这些算法大多在 LR(0)的基础上进行改进，根据处理冲突能力的不同又有 **SLR**、**LR(1)**和 **LALR** 等分析方法，我们在此对这类方法进行简要概述。

（1）**SLR 分析（Simple LR）**：SLR 分析器的构造方法基本与 LR(0)相同，但它规定只在 Follow 集指定的地方放置归约动作，即寻找可行前缀：要把 a 归约成为 *A*，后面的输入必须是 Follow(*A*)中的终结符号，否则只能移入；

（2）**LR(1)分析**：LR(1)比 LR(0)多向前看一个符号，因此 LR(1)中的项可以利用包含的信息来消除一些归约动作，相当于分裂一些 LR(0)项中的状态，更精确地指明应当何时归约，但面临更多的状态处理；

（3）**LALR 分析（Look-Ahead LR）**：LALR 基于 LR(0)项集，但是每个项都带有向前看符号。分析能力强于 SLR 方法，且分析表的规模和 SLR 分析表规模相同。LALR 分析方法已经可以处理大多数现实场景的程序设计语言了。

2.2 词法分析和语法分析的实践技术

词法分析和语法分析这两部分内容是当前编译器研究中被自动化得最好的部分。也就是 说，即使没有任何理论基础，在掌握了相应工具的用法之后，也应该可以在短时间内做出功能完备的词法分析和语法分析程序。当然这并不意味着，词法分析和语法分析部分的理论基础不重要。恰恰相反，这一部分被认为是计算机理论在工程实践中最成功的应用之一。希望读者将本节中对工具的介绍与前面介绍的理论方法相结合，深刻理解编译器在实现词法分析和语法分析过程中的要点。

本节是本书讨论的编译器构造技术的第一部分实践内容，完成实践内容一不需要太多的理论基础，只要看完并掌握了前面介绍的理论内容即可。本节将分别介绍词法分析工具GNU Flex和语法分析工具GNU Bison的使用。需要提醒的是，虽然本节介绍的内容是对于工具的使用，但其中重点在于研究和理解前面介绍的基于正则表达式和有限状态自动机的词法分析技术的实现细节，以及前面讨论的自顶向下和自底向上语法分析算法的实现细节。

2.2.1 词法分析实现思想概述

词法分析程序的主要任务是将输入文件中的字符流组织成为词法单元流，在某些字符不符合程序设计语言词法规范时它也要有能力报告相应的错误。词法分析程序是编译器所有模块中唯一读入并处理输入文件中每一个字符的模块，它使得后面的语法分析阶段能在更高抽象层次上运作，而不必纠结于字符串处理这样的细节问题。

高级程序设计语言大多采用英文作为输入方式，而英文有个非常好的性质，就是它比较容易断词：相邻的英文字母一定属于同一个词，而字母与字母之间插入任何非字母的字符（如空格、运算符等）就可以将一个词断成两个词。判断一个词是否符合语言本身的词法规范也相对简单，一个直接的办法是：我们可以事先建一张搜索表，将所有符合词法规范的字符串都存放在表中，每次我们从输入文件中断出一个词之后，通过查找这张表就可以判断该词究竟合法还是不合法。

正因为词法分析任务的难度不高，在实用的编译器中它常常是手工写成的，而并非使用工具生成。例如，我们下面要介绍的这个工具GNU Flex原先就是为了帮助GCC进行词法分析而被开发出来的，但在4.0版本之后，GCC的词法分析器已经一律改为手写了。不过，本书中实践内容一要求使用工具来做，而词法分析程序生成工具所基于的理论基础，是计算理论中最入门的内容：正则表达式和有限状态自动机。

一个正则表达式由特定字符串构成，或者由其他正则表达式通过以下三种运算得到：

- **并运算：**两个正则表达式 r 和 s 的并记作 r | s，意为 r 或 s 都可以被接受。
- **连接运算：**两个正则表达式 r 和 s 的连接记作 rs，意为 r 之后紧跟 s 才可以被接受。
- **Kleene 闭包：**一个正则表达式 r 的 Kleene 闭包记作 r*，它表示：ε | r | rr | rrr | …。

有关正则表达式的内容在本书的理论部分讨论过。正则表达式之所以被广泛应用，一方面是因为它在表达能力足够强（基本上可以表示所有的词法规则）的同时还易于书写和理解；另一方面是因为判断一个字符串是否被一个特定的正则表达式接受可以做到非常高效（在线性时间内即可完成）。比如，我们可以将一个正则表达式转换为一个 NFA，然后将这个 NFA 转换为一个 DFA，再将转换好的 DFA 进行化简，之后我们就可以通过模拟这个 DFA 的运行来对输入串进行识别了。具体的 NFA 和 DFA 的含义，以及如何进行正则表达式、NFA 及 DFA 之间的转换等，请参考本书的理论部分。这里我们仅需要知道，前面所述的所有转换和识别工作，都可以由工具自动完成。而我们所需要做的，仅仅是为工具提供作为词法规范的正则表达式。

2.2.2 GNU Flex 介绍

Flex的前身是Lex。Lex是1975年由Mike Lesk和当时还在AT&T做暑期实习的Eric Schmidt，共同完成的一款基于UNIX环境的词法分析程序生成工具。虽然Lex很出名并被广泛使用，但它的低效和诸多问题也使其颇受诟病。后来伯克利实验室的Vern Paxson使用C语言重写Lex，并将这个新的程序命名为Flex（意为Fast Lexical Analyzer Generator）。无论在效率还是在稳定性上，Flex都远远好于它的前辈Lex。我们在Linux下使用的是Flex在GNU License下的版本，称作GNU Flex。

GNU Flex在Linux下的安装非常简单。其官方网站提供了安装包，不过在基于Debian的Linux系统下，更简单的安装方法是直接在命令行敲入如下命令：

```
sudo apt-get install flex
```

虽然版本不一样，但GNU Flex的使用方法与本书介绍的Lex基本相同。首先，我们需要自行完成包括词法规则等在内的Flex代码。至于如何编写这部分代码我们在后面会提到，现在先假设这部分写好的代码名为lexical.l。随后，我们使用Flex对该代码进行编译：

```
flex lexical.l
```

编译好的结果会保存在当前目录下的lex.yy.c文件中。该文件是一份C语言的源代码。这份源代码里目前对我们有用的函数只有 yylex()，该函数的作用就是读取输入文件中的一个词法单元。我们可以再为它编写一个main函数：

```
1  int main(int argc, char** argv) {
2    if (argc > 1) {
3      if (!(yyin = fopen(argv[1], "r"))) {
4        perror(argv[1]);
```

```
5         return 1;
6       }
7     }
8     while (yylex() != 0);
9     return 0;
10  }
```

这个main函数通过命令行读入若干个参数，取第一个参数为其输入文件名并尝试打开该输入文件。如果文件打开失败则退出，如果成功则调用yylex()进行词法分析。其中，变量yyin是Flex内部使用的一个变量，表示输入文件的文件指针，如果我们不去设置它，那么Flex会将它自动设置为stdin（即标准输入，通常连接到键盘）。注意，如果我们将main函数独立设为一个文件，则需要声明yyin为外部变量：extern FILE* yyin。

将这个main 函数单独放到文件 main.c中（也可以直接放入lexical.l中的用户自定义代码部分，这样就可以不必声明yyin；甚至可以不写main函数，因为Flex会自动配一个，但不推荐这么做），然后编译这两个C源文件。我们将输出程序命名为scanner：

```
gcc main.c lex.yy.c -lfl -o scanner
```

注意编译命令中的“-lfl”参数不可缺少，否则GCC会因为缺少库函数而报错。之后我们就可以使用这个scanner程序进行词法分析了。例如，想要对一个测试文件test.cmm进行词法分析，只需要在命令行输入：

```
./scanner test.cmm
```

这样就可以得到我们想要的结果了。

2.2.3 Flex：编写源代码

以上介绍的是使用Flex创建词法分析程序的基本步骤。在整个创建过程中，最重要的文件无疑是所编写的Flex源代码，它完全决定了词法分析程序的一切行为。接下来我们介绍如何编写Flex源代码。

Flex源代码文件包括三个部分，由“%%”隔开，如下所示：

```
1  {definitions}
2  %%
3  {rules}
4  %%
5  {user subroutines}
```

第一部分为**定义部分**，实际上就是给某些后面可能经常用到的正则表达式取一个别名，从而简化词法规则的书写。定义部分的格式一般为：

```
name  definition
```

其中name是名字，definition是任意的正则表达式（正则表达式该如何书写后面会介绍）。例如，下面的这段代码定义了两个名字：digit和letter，前者代表0到9中的任意一个数字字符，后者则代表任意一个小写字母、大写字母或下划线：

```
1  …
2  digit  [0-9]
3  letter  [_a-zA-Z]
4  %%
5  …
6  %%
7  …
```

Flex源代码文件的第二部分为规则部分，它由正则表达式和相应的响应函数组成，其格式为：

```
pattern  {action}
```

其中pattern为正则表达式，其书写规则与前面的定义部分的正则表达式相同。而action则为将要进行的具体操作，这些操作可以用一段C代码表示。Flex将按照这部分给出的内容依次尝试每一个规则，尽可能匹配最长的输入串。如果有些内容不匹配任何规则，Flex默认只将其复制到标准输出，想要修改这个默认行为只需要在所有规则的最后加上一条“ . ”（即匹配任何输入）规则，然后在其对应的action部分书写想要的行为即可。

例如，下面这段Flex代码在遇到输入文件中包含一串数字时，会将该数字串转化为整数值并打印到屏幕上：

```
1  …
2  digit  [0-9]
3  %%
4  {digit}+  { printf("Integer value %d\n", atoi(yytext)); }
5  …
6  %%
7  …
```

其中变量yytext的类型为char*，它是Flex为我们提供的一个变量，里面保存了当前词法单元所对应的词素。函数atoi()的作用是把一个字符串表示的整数转化为int类型。

Flex源代码文件的第三部分为**用户自定义代码部分**。这部分代码会被原封不动地复制到lex.yy.c中，以方便用户自定义所需要执行的函数（之前我们提到过的main函数也可以写在这里）。值得一提的是，如果用户想要对这部分所用到的变量、函数或者头文件进行声明，可以在前面的定义部分（即Flex源代码文件的第一部分）之前使用“%{”和“%}”符号将要声明的内容添加进去。被“%{”和“%}” 所包围的内容也会一并复制到lex.yy.c的最前面。

下面通过一个简单的例子来说明Flex源代码该如何书写[1]。我们知道UNIX /Linux下有一个常用的文字统计工具wc，它可以统计一个或者多个文件中的（英文）字符数、单词数和行数。利用Flex我们可以快速地写出一个类似的文字统计程序：

```
%{
  /* 此处省略#include 部分 */
  int chars = 0;
  int words = 0;
  int lines = 0;
%}
letter  [a-zA-Z]
%%
{letter}+  { words++; chars+= yyleng; }
\n  { chars++; lines++; }
.  { chars++; }
%%
int main(int argc, char** argv) {
  if (argc > 1) {
    if (!(yyin = fopen(argv[1], "r"))) {
      perror(argv[1]);
      return 1;
    }
  }
  yylex();
  printf("%8d%8d%8d\n", lines, words, chars);
  return 0;
}
```

其中yyleng是Flex为我们提供的变量，可以将其理解为strlen(yytext)。我们用变量chars记录输入文件中的字符数、words记录单词数、lines记录行数。上面这段程序很好理解：每遇到一个换行符就把行数加一，每识别出一个单词就把单词数加一，每读入一个字符就把字符数加一。最后在main函数中把chars、words和lines的值全部打印出来。需要注意的是，由于在规则部分中我们没有让 yylex()返回任何值，因此在main函数中调用yylex()时可以不套外层的while循环。

真正的wc工具可以一次传入多个参数从而统计多个文件。为了能够让这个Flex程序对多个文件进行统计，我们可以对 main 函数进行如下修改：

```
int main(int argc, char** argv) {
  int i, totchars = 0, totwords = 0, totlines = 0;
  if (argc < 2) {  /* just read stdin */
    yylex();
    printf("%8d%8d%8d\n", lines, words, chars);
    return 0;
  }
  for (i = 1; i < argc; i++) {
    FILE *f = fopen(argv[i], "r");
    if (!f) {
```

[1] 该例子来源于：《flex & bision》，John Levine著，陆军译，东南大学出版社，第29页，2011年。

```
11        perror(argv[i]);
12        return 1;
13      }
14      yyrestart(f);
15      yylex();
16      fclose(f);
17      printf("%8d%8d%8d %s\n", lines, words, chars, argv[i]);
18      totchars += chars; chars = 0;
19      totwords += words; words = 0;
20      totlines += lines; lines = 0;
21    }
22    if (argc > 1)
23      printf("%8d%8d%8d total\n", totlines, totwords, totchars);
24    return 0;
25  }
```

其中yyrestart(f)函数是Flex提供的库函数，它可以让Flex将其输入文件的文件指针yyin设置为f（当然也可以像前面一样手动设置令yyin = f）并重新初始化该文件指针，令其指向输入文件的开头。

2.2.4 Flex：书写正则表达式

Flex源代码中无论是定义部分还是规则部分，正则表达式都在其中扮演了重要的作用。那么，如何在Flex源代码中书写正则表达式呢？下面介绍一些规则。

（1）符号“.”匹配除换行符“\n”之外的任何一个字符。

（2）符号“[”和“]”共同匹配一个字符类，即方括号之内只要有一个字符被匹配上了，那么被方括号括起来的整个表达式就都被匹配上了。例如，[0123456789]表示0~9中任意一个数字字符，[abcABC]表示a、b、c三个字母的小写或者大写。方括号中还可以使用连字符“-”表示一个范围，例如[0123456789]也可以直接写作[0-9]，而所有小写字母字符也可直接写成[a-z] 。如果方括号中的第一个字符是“^”，则表示对这个字符类取补，即方括号之内如果没有任何一个字符被匹配上，那么被方括号括起来的整个表达式就认为被匹配上了。例如，[^_0-9a-zA-Z]表示所有的非字母、数字以及下划线的字符。

（3）符号“^”用在方括号之外则会匹配一行的开头，符号“$”用于匹配一行的结尾，符号“<<EOF>>”用于匹配文件的结尾。

（4）符号“{”和“}”含义比较特殊。如果花括号之内包含了一个或者两个数字，则代表花括号之前的那个表达式需要出现的次数。例如，A{5}会匹配AAAAA，A{1,3}则会匹配A、AA或者AAA。如果花括号之内是一个在Flex源代码的定义部分定义过的名字，则表示那个名字对应的

正则表达式。例如，在定义部分如果定义了letter为[a-zA-Z]，则{letter}{1,3}表示连续的 1 ～ 3 个英文字母。

（5）符号“*”为**Kleene闭包**操作，匹配零个或者多个表达式。例如{letter}*表示零个或者多个英文字母。

（6）符号“+”为**正闭包**操作，匹配一个或者多个表达式。例如{letter}+表示一个或者多个英文字母。

（7）符号“?”匹配零个或者一个表达式。例如表达式-?[0-9]+表示前面带一个可选的负号的数字串。无论是*、+还是?，它们都只对其最邻近的那个字符生效。例如abc+表示ab后面跟一个或多个c，而不表示一个或者多个abc。如果要匹配后者，则需要使用小括号“(”和“)”将这几个字符括起来：(abc)+。

（8）符号“|”为**选择**操作，匹配其之前或之后的任一表达式。例如，faith|hope|charity表示这三个串中的任何一个。

（9）符号“\”用于表示各种转义字符，与C语言字符串里“\”的用法类似。例如，“\n”表示换行，“\t”表示制表符，“*”表示星号，“\\”代表字符“\”等。

（10）符号“""（英文引号）将逐字匹配被引起来的内容（即无视各种特殊符号及转义字符）。例如，表达式"..."就表示三个点而不表示三个除换行符以外的任意字符。

（11）符号“/”会查看输入字符的上下文，例如，x/y识别x仅当在输入文件中x之后紧跟着y，0/1可以匹配输入串01中的0但不匹配输入串02中的0。

（12）任何不属于上面介绍过的有特殊含义的字符在正则表达式中都仅匹配该字符本身。

下面我们通过几个例子来练习一下Flex源代码里正则表达式的书写：

（1）带一个可选的正号或者负号的数字串，可以这样写：[+–]?[0-9]+。

（2）带一个可选的正号或者负号以及一个可选的小数点的数字串，表示起来要困难一些，可以考虑下面几种写法：

1) [+–]?[0-9.]+会匹配太多额外的模式，像1.2.3.4；

2) [+–]?[0-9]+\.?[0-9]+会漏掉某些模式，像12.或者.12；

3) [+–]?[0-9]*\.?[0-9]+会漏掉12.;

4) [+–]?[0-9]+\.?[0-9]*会漏掉.12;

5) [+–]?[0-9]*\.?[0-9]*会多匹配空串或者只有一个小数点的串；

6) 正确的写法是：[+–]?([0-9]*\.?[0-9]+|[0-9]+\.)。

（3）假设我们现在做一个汇编器，目标机器的CPU中有32个寄存器，编号为0…31。汇编源代码可以使用r后面加一个或两个数字的方式来表示某一个寄存器，例如r15表示第15号寄存器，r0或r00表示第0号寄存器，r7或者r07表示第7号寄存器等。现在我们希望识别汇编源代码中所有可能的寄存器表示，可以考虑下面几种写法：

1) r[0-9]+可以匹配r0和r15，但它也会匹配r99999，目前世界上还不存在 CPU 能拥 有 100 万个寄存器。

2) r[0-9]{1,2}同样会匹配一些额外的表示，例如r32和r48等。

3) r([0-2][0-9]?|[4-9]|(3(0|1)?))是正确的写法，但其可读性比较差。

4) 正确性毋庸置疑并且可读性最好的写法应该是：r0|r00|r1|r01|r2|r02|r3|r03|r4|r04|r5|r05|r6|r06|r7|r07|r8|r08|r9|r09|r10|r11|r12|r13|r14|r15|r16|r17|r18|r19|r20|r21|r22|r23|r24|r25|r26|r27|r28|r29|r30|r31，但这样写可扩展性又非常差，如果目标机器上有128甚至256个寄存器呢？

2.2.5 Flex：高级特性

基于前面介绍的 Flex 内容已足够完成实践内容一的词法分析部分。下面介绍一些 Flex 的高级特性，使用户更方便和灵活地使用 Flex。这部分内容是可选的，跳过也不会对实践内容一的完成产生影响。

1. **yylineno 选项：**

在写编译器程序的过程中，经常需要记录行号，以便在报错时提示输入文件的哪一行出现了问题。为了能记录这个行号，我们可以自己定义某个变量，例如 lines，来记录词法分析程序当前读到了输入文件的哪一行。每当识别出“\n”，我们就让 lines = lines + 1。

实际上，Flex 内部已经为我们提供了类似的变量，叫作 yylineno。我们不必去维护 yylineno 的值，它会在每行结束自动加一。不过，默认状态下它并不开放给用户使用。如果我们想要读取 yylineno 的值，则需要在 Flex 源代码的定义部分加入语句：

```
%option yylineno
```

需要说明的是，虽然 yylineno 会自动增加，但当我们在词法分析过程中调用 yyrestart()函数读取另一个输入文件时它却不会重新被初始化，因此我们需要自行添加初始化语句 yylineno = 1。

2. 输入缓冲区：

课本上介绍的词法分析程序其工作原理都是在模拟一个 DFA 的运行。这个 DFA 每次读入一个字符，然后根据状态之间的转换关系决定下一步应该转换到哪个状态。事实上，实用的词法分析程序很少会从输入文件中逐个读入字符，因为这样需要进行大量的磁盘操作，效率较低。更加高效的办法是一次读入一大段输入字符，并将其保存在专门的输入缓冲区中。

在 Flex 中，所有的输入缓冲区都有一个共同的类型，叫作 YY_BUFFER_STATE。可以通过 yy_create_buffer()函数为一个特定的输入文件开辟一块输入缓冲区，例如：

```
1  YY_BUFFER_STATE bp;
2  FILE* f;
3  f = fopen(…, "r");
4  bp = yy_create_buffer(f, YY_BUF_SIZE);
5  yy_switch_to_buffer(bp);
6  …
7  yy_flush_buffer(bp);
8  …
9  yy_delete_buffer(bp);
```

其中 YY_BUF_SIZE 是 Flex 内部的一个常数，通过调用 yy_switch_to_buffer()函数可以让词法分析程序到指定的输入缓冲区中读字符，调用 yy_flush_buffer()函数可以清空缓冲区中的内容，而调用 yy_delete_buffer()则可以删除一个缓冲区。

如果词法分析程序要支持文件与文件之间的相互引用（例如 C 语言中的#include），可能需要在词法分析的过程中频繁地使用 yyrestart()切换当前的输入文件。在切换到其他输入文件再切换回来之后，为了能继续之前的词法分析任务，需要无损地保留原先输入缓冲区的内容，这就需要使用一个栈来暂存当前输入文件的缓冲区。虽然 Flex 也提供了相关的函数来 帮助我们做这件事情，但这些函数的功能比较弱，建议最好自己手写。

3. Flex 库函数 input：

Flex 库函数 input()可以从当前的输入文件中读入一个字符，这有助于我们不借助正则表达式来实现某些功能。例如，下面这段代码在输入文件中发现双斜线“//”后，将从当前字符开始一直到行尾的所有字符全部丢弃掉：

```
%%
"//"  {
    char c = input();
    while (c != '\n') c = input();
  }
```

4. Flex 库函数 unput：

Flex 库函数 unput(char c)可以将指定的字符放回输入缓冲区中，这对于宏定义等功能的实现是很方便的。例如，假设之前定义过一个宏#define BUFFER_LEN 1024，当在输入文件中遇到字符串 BUFFER_LEN 时，下面这段代码将该宏所对应的内容放回输入缓冲区：

```
char* p = macro_contents("BUFFER_LEN");  // p = "1024"
char* q = p + strlen(p);
while (q > p) unput(*--q);  // push back right-to-left
```

5. Flex 库函数 yyless 和 yymore：

Flex 库函数 yyless(intn)可以将刚从输入缓冲区中读取的 yyleng–n 个字符放回到输入缓冲区中，而函数 yymore()可以告诉 Flex 保留当前词素，并在下一个词法单元被识别出来之后将下一个词素连接到当前词素的后面。配合使用 yyless()和 yymore()可以方便地处理那些边界难以界定的模式。例如，我们在为字符串常量书写正则表达式时，往往会写成由一对双引号引起来的所有内容\"[^\"]*\"，但有时候被双引号引起来的内容里面也可能出现跟在转义符号之后的双引号，例如”This is an\”example\ ”。那么如何使用 Flex 处理这种情况呢？方法之一就是借助于 yyless 和 yymore：

```
%%
\"[^\"]*\"  {
   if (yytext[yyleng - 2] == '\\') {
     yyless(yyleng - 1);
     yymore();
       } else {
     /* process the string literal */
   }
  }
```

6. Flex 宏 REJECT：

Flex 宏 REJECT 可以帮助我们识别那些互相重叠的模式。当我们执行 REJECT 之后，Flex 会进行一系列的操作，这些操作的结果相当于将 yytext 放回输入之内，然后去试图匹配当前规则之后的那些规则。例如，考虑下面这段 Flex 源代码：

```
1  %%
2  pink  { npink++; REJECT; }
3  ink  { nink++; REJECT; }
4  pin  { npin++; REJECT; }
```

这段代码会统计输入文件中所有的 pink、ink 和 pin 出现的个数，即使这三个单词之间互有重叠。

Flex 还有更多的特性，感兴趣的读者可以参考其用户手册。

2.2.6 词法分析实践的额外提示

为了完成实践内容一，首先需要阅读 C−−语言文法（见附录 A），包括其文法定义和补充说明。除了 INT、FLOAT 和 ID 这三个词法单元需要自行为其书写正则表达式之外，剩下的词法单元都没有难度。

阅读完 C−−语言文法，对 C−−的词法有大概的了解之后，就可以开始编写 Flex 源代码了。在输入所有的词法之后，为了能检验词法分析程序是否工作正常，可以暂时向屏幕打印当前的词法单元的名称，例如：

```
1  %%
2  "+"  { printf("PLUS\n"); }
3  "-"  { printf("SUB\n"); }
4  "&&"  { printf("AND\n"); }
5  "||"  { printf("OR\n"); }
6  …
```

为了能够报告错误类型 A，可以在所有规则的最后增加类似于这样的一条规则：

```
1  %%
2  …
3  . {
4     printf("Error type A at Line %d: Mysterious characters \'%s\'\n",
5       yylineno, yytext);
6  }
```

完成 Flex 源代码的编写之后，使用前面介绍过的方法将其编译出来，就可以自己书写一些小规模的输入文件来测试所编写的词法分析程序了。一定要确保词法分析程序的正确性！如果词法

分析这里出了问题没有检查出来，到了后面语法分析发现了前面的问题再回头调试，那将增加许多不必要的麻烦。为了在编写 Flex 源代码时少走弯路，给出以下几条建议：

- 留意空格和回车的使用。如果不注意，有时很容易让本应是空白符的空格或者回车变成正则表达式的一部分，有时又很容易让本应是正则表达式一部分的空格或回车变成 Flex 源代码里的空白符。
- 永远不要在正则表达式和其所对应的动作之间插入空行。
- 如果对正则表达式中的运算符优先级有疑问，那就不要吝啬使用括号来确保正则表达式的优先级确实是我们想要的。
- 使用花括号括起每一段动作，即使该动作只包含一行代码。
- 在定义部分我们可以为许多正则表达式取别名，这一点要好好利用。别名可以让后面的正则表达式更容易阅读、扩展和调试。

在正则表达式中引用之前定义过的某个别名（例如 digit）时，时刻谨记该别名一定要用花括号“{”和“}”括起来。

2.2.7 语法分析实现思想概述

词法分析的下一阶段是语法分析。语法分析程序的主要任务是读入词法单元流、判断输入程序是否匹配程序设计语言的语法规范，并在匹配规范的情况下构建起输入程序的静态结构。语法分析使得编译器的后续阶段看到的输入程序不再是一串字符流或者单词流，而是一个结构整齐、处理方便的数据对象。

语法分析与词法分析有很多相似之处：它们的基础都是形式语言理论，它们都是计算机理论在工程实践中最成功的应用，它们都能被高效完成，它们的构建都可以被工具自动化完成。不过，由于语法分析本身要比词法分析复杂得多，手写一个语法分析程序的代价太大，所以目前绝大多数实用的编译器在语法分析这里都是使用工具帮助完成的。

正则表达式难以进行任意大的计数，所以很多在程序设计语言中常见的结构（例如匹配的括号）无法使用正则文法进行表示。为了能够有效地对常见的语法结构进行表示，人们使用了比正则文法表达能力更强的**上下文无关文法**。然而，虽然上下文无关文法在表达能力上要强于正则语

言，但在判断某个输入串是否属于特定 CFG 的问题上，时间效率最好的算法也要 $O(n^3)$[1]，这样的效率让人难以接受。因此，现代程序设计语言的语法大多属于一般 CFG 的一个足够大的子集，比较常见的子集有 LL(*k*)文法以及 LR(*k*)文法。判断一个输入是否属于这两种文法都只需要线性时间。

上下文无关文法 *G* 在形式上是一个四元组：终结符号（也就是词法单元）集合 *T*、非终结符号集合 *NT*、初始符号 *S* 以及产生式集合 *P*。产生式集合 *P* 是一个文法的核心，它通过产生式定义了一系列的推导规则，从初始符号出发，基于这些产生式，经过不断地将非终结符号替换为其他非终结符号以及终结符号，即可得到一串符合语法归约的词法单元。这个替换和推导的过程可以使用树形结构表示，称作**语法树**。事实上，语法分析的过程就是把词法单元流变成语法树的过程。尽管在之前曾经出现过各式各样的算法，但目前最常见的构建语法树的技术只有两种：**自顶向下**方法和**自底向上**方法。我们下面将要介绍的工具 Bison 所生成的语法分析程序就采用了自底向上的 LALR(1)分析技术（通过一定的设置还可以让 Bison 使用另一种被称为 GLR 的分析技术，不过对该技术的介绍已经超出了本书的讨论范围）。而其他的某些语法分析工具，例如基于 Java 语言的 JTB[2]生成的语法分析程序，则是采用了自顶向下的 LL(1)分析技术。当然，具体的工具采用哪一种技术这种细节，对于工具的使用者来讲都是完全屏蔽的。与词法分析程序的生成工具一样，工具的使用者所要做的仅仅是将输入程序的程序设计语言的语法告诉语法分析程序生成工具，虽然工具本身不能直接构造出语法树，但我们可以通过在语法产生式中插入语义动作这种更加灵活的形式，来实现一些甚至比构造语法树更加复杂的功能。

2.2.8 GNU Bison 介绍

Bison的前身为基于 UNIX 的Yacc。令人惊讶的是，Yacc的发布时间甚至比Lex还要早。Yacc所采用的LR分析技术的理论基础早在50年代就已经由Knuth逐步建立了起来，而Yacc本身则是贝尔实验室的S.C. Johnson基于这些理论在1975 ～ 1978年写成的。到了1985年，当时在UC Berkeley

[1] 《自动机理论、语言和计算导论》，John E. Hopcroft等著，刘田、姜晖和王捍贫译，机械工业出版社，中信出版社，第209页，2004年。

[2] http://compilers.cs.ucla.edu/jtb/.

的一位研究生 Bob Corbett在BSD下重写了Yacc，后来GNU Project接管了这个项目，为其增加了许多新的特性，于是就有了我们今天所用的GNU Bison。

GNU Bison在Linux下的安装非常简单，可以去它的官方网站上下载安装包自行安装，基于Debian的Linux系统下更简单的方法同样是直接在命令行敲入如下命令：

```
sudo apt-get install bison
```

虽说版本不一样，但GNU Bison的基本使用方法和课本上所介绍的Yacc没有什么不同。首先，我们需要自行完成包括语法规则等在内的Bison源代码。如何编写这份代码后面会提到，现在先假设这份写好的代码名为 syntax.y。随后，我们使用Bison对这份代码进行编译：

```
bison syntax.y
```

编译好的结果会保存在当前目录下的syntax.yy.c文件中。打开这个文件就会发现，该文件本质上就是一份C语言的源代码。事实上，这份源代码里目前对我们有用的函数只有 yyparse()，该函数的作用就是对输入文件进行语法分析，如果分析成功没有错误则返回0，否则返回非0。不过，只有这个yyparse()函数还不足以让我们的程序运行起来。前面说过，语法分析程序的输入是一个个的词法单元，那么Bison通过什么方式来获得这些词法单元呢？事实上，Bison在这里需要用户为它提供另外一个专门返回词法单元的函数，这个函数正是 yylex()。

函数yylex()相当于嵌在Bison里的词法分析程序。这个函数可以由用户自行实现，但因为我们之前已经使用Flex生成了一个yylex()函数，能不能让Bison使用Flex生成的yylex()函数呢？答案是肯定的。

仍以Bison源代码文件syntax.y为例。首先，为了能够使用Flex中的各种函数，需要在Bison源代码中引用lex.yy.c：

```
#include "lex.yy.c"
```

随后在使用Bison编译这份源代码时，我们需要加上“-d”参数：

```
bison -d syntax.y
```

这个参数的含义是，将编译的结果分拆成syntax.tab.c和syntax.tab.h两个文件，其中.h文件里包含着一些词法单元的类型定义之类的内容。得到这个.h文件之后，下一步是修改我们的Flex源代码lexical.l，增加对syntax.tab.h的引用，并且让Flex源代码中规则部分的每一条action都返回相应的词法单元，如下所示：

```
1  %{
```

```
 2   #include "syntax.tab.h"
 3   …
 4 %}
 5 …
 6 %%
 7 "+"  { return PLUS; }
 8 "-"  { return SUB; }
 9 "&&"  { return AND; }
10 "||"  { return OR; }
11 …
```

其中，返回值PLUS和SUB等都是在Bison源代码中定义过的词法单元（如何定义它们后文会提到）。由于我们刚刚修改了lexical.l，需要重新将它编译出来：

```
flex lexical.l
```

接下来重写 main函数。由于Bison会在需要时自动调用yylex()，我们在main函数中也就不需要调用它了。不过，Bison是不会自己调用yyparse()和yyrestart()的，因此仍需要在 main 函数中显式地调用这两个函数：

```
 1 int main(int argc, char** argv)
 2 {
 3   if (argc <= 1) return 1;
 4   FILE* f = fopen(argv[1], "r");
 5   if (!f)
 6   {
 7     perror(argv[1]);
 8     return 1;
 9   }
10   yyrestart(f);
11   yyparse();
12   return 0;
13 }
```

现在我们有了三个C语言源代码文件：main.c、lex.yy.c以及syntax.tab.c，其中lex.yy.c已经被syntax.tab.c引用了，因此我们最后要做的就是把main.c和syntax.tab.c放到一起进行编译：

```
gcc main.c syntax.tab.c -lfl -ly -o parser
```

其中“-lfl”不要省略，否则GCC会因缺少库函数而报错，但一般情况下可以省略“-ly”。现在我们可以使用这个parser程序进行语法分析了。例如，想要对一个输入文件test.cmm进行语法分析，只需要在命令行输入：

```
./parser test.cmm
```

就可以得到想要的结果了。

2.2.9 Bison：编写源代码

前面介绍了使用 Flex和Bison联合创建语法分析程序的基本步骤。在整个创建过程中，最重要的文件无疑是所编写的Flex源代码和Bison源代码文件，它们完全决定了所生成的语法分析程序的行为。

同Flex源代码类似，Bison源代码也分为三个部分，其作用与Flex源代码大致相同。第一部分是**定义部分**，所有词法单元的定义都可以放到这里；第二部分是**规则部分**，其中包括具体的语法和相应的语义动作；第三部分是**用户函数部分**，这部分的源代码会被原封不动地复制到syntax.tab.c中，以方便用户自定义所需要的函数（main函数也可以写在这里，不过不推荐这么做）。值得一提的是，如果用户想要对这部分所用到的变量、函数或者头文件进行声明，可以在定义部分（也就是Bison源代码的第一部分）之前使用“%{”和“%}”符号将要声明的内容添加进去。被“%{”和“%}”所包围的内容也会被一并复制到 syntax.tab.c的最前面。

下面我们通过一个例子来对Bison源代码的结构进行解释。一个在控制台运行可以进行整数四则运算的小程序，其语法如下所示（这里假设词法单元INT代表Flex识别出来的一个整数，ADD代表加号+，SUB代表减号–，MUL代表乘号*，DIV代表除号/）：

```
Calc → ε
| Exp
Exp → Factor
| Exp ADD Factor
| Exp SUB Factor
Factor → Term
| Factor MUL Term
| Factor DIV Term
Term → INT
```

这个小程序的Bison源代码为：

```
 1  %{
 2    #include <stdio.h>
 3  %}
 4
 5  /* declared tokens */
 6  %token INT
 7  %token ADD SUB MUL DIV
 8
 9  %%
10  Calc :  /* empty */
11    | Exp  { printf("= %d\n", $1); }
12    ;
13  Exp : Factor
14    | Exp ADD Factor  { $$ = $1 + $3; }
15    | Exp SUB Factor  { $$ = $1 - $3; }
16    ;
17  Factor : Term
18    | Factor MUL Term  { $$ = $1 * $3; }
```

```
19    | Factor DIV Term  { $$ = $1 / $3; }
20    ;
21  Term : INT
22    ;
23  %%
24  #include "lex.yy.c"
25  int main() {
26    yyparse();
27  }
28  yyerror(char* msg) {
29    fprintf(stderr, "error: %s\n", msg);
30  }
```

这段Bison源代码以“%{”和“%}”开头，被“%{”和“%}”包含的内容主要是对stdio.h的引用。接下来是一些以%token开头的词法单元（终结符号）定义，如果需要采用Flex生成的yylex()的话，那么在这里定义的词法单元都可以作为Flex源代码里的返回值。与终结符号相对的，所有未被定义为%token的符号都会被看作非终结符号，这些非终结符号要求必须在任意产生式的左边至少出现一次。

第二部分是书写产生式的地方。第一个产生式左边的非终结符号默认为初始符号（也可以通过在定义部分添加%start X来将另外的某个非终结符号 X指定为初始符号）。产生式里的箭头在这里用冒号“:”表示，一组产生式与另一组之间以分号“;”隔开。产生式里无论是终结符号还是非终结符号都各自对应一个属性值，产生式左边的非终结符号对应的属性值用$$表示，右边的几个符号的属性值按从左到右的顺序依次对应为$1、$2、$3等。每条产生式的最后可以添加一组以花括号“{”和“}”括起来的语义动作，这组语义动作会在整条产生式归约完成之后执行，如果不明确指定语义动作，那么Bison将采用默认的语义动作{ $$ = $1 }。语义动作也可以放在产生式的中间，例如$A \to B\ \{\ldots\}\ C$，这样的写法等价于$A \to BMC$，$M \to \varepsilon\ \{\ldots\}$，其中M为额外引入的一个非终结符号。需要注意的是，在产生式中间添加语义动作在某些情况下有可能会在原有语法中引入冲突，因此使用的时候要特别谨慎。

在这里可能存在疑问：每一个非终结符号的属性值都可以通过它所产生的那些终结符号 或者非终结符号的属性值计算出来，但是终结符号本身的属性值该如何得到呢？答案是：在yylex()函数中得到。因为我们的yylex()函数是由Flex源代码生成的，因此要想让终结符号 带有属性值，就必须回头修改Flex源代码。假设在我们的Flex源代码中，INT词法单元对应着一个数字串，那么我们可以将Flex源代码修改为：

```
1  …
2  digit  [0-9]
3  %%
4  {digit}*  {
5     yylval = atoi(yytext);
6     return INT;
7    }
8  …
9  %%
10 …
```

其中yylval是Flex的内部变量，表示当前词法单元所对应的属性值。我们只需将该变量的值赋成atoi(yytext)，就可以将词法单元INT的属性值设置为它所对应的整数值了。

回到之前的Bison源代码中。在用户自定义函数部分我们写了两个函数：一个是只调用了yyparse() 的 main 函数，另一个是没有返回类型并带有一个字符串参数的 yyerror() 函 数。yyerror() 函数会在语法分析程序每发现一个语法错误时被调用，其默认参数为“syntax error”。默认情况下yyerror()只会将传入的字符串参数打印到标准错误输出上，而我们也可以自己重新定义这个函数，从而使它打印一些其他内容，例如上例中我们就在该参数前面多打印了“error: ”的字样。

现在，编译并执行这个程序，先在控制台输入 10-2+3，然后按下回车键，最后按下组合键Ctrl+D 结束，会看到屏幕上打印出了计算结果 11。

2.2.10 Bison：属性值的类型

在上面的例子中，每个终结符号或非终结符号的属性值都是 int类型。但在我们构建语法树的过程中，我们希望不同的符号对应的属性值能有不同的类型，而且最好能对应任意的类型而不仅仅是int类型。下面我们介绍如何在Bison中解决这个问题。

第一种方法是对宏YYSTYPE进行重定义。Bison里会默认所有属性值的类型以及变量yylval的类型都是YYSTYPE，默认情况下YYSTYPE被定义为int。如果在Bison源代码的“%{”和“%}”之间加入#define YYSTYPE float，那么所有的属性值就都被定义 float类型。如何使不同的符号对应不同的类型呢？可以将YYSTYPE定义成一个联合体类型，这样就可以根据符号的不同来访问联合体中不同的域，从而实现多种类型的效果。

这种方法虽然可行，但在实际操作中还是稍显麻烦，因为每次对属性值的访问都要自行指定哪个符号对应哪个域。实际上，在Bison中已经内置了其他的机制来方便用户对属性值类型进行处理， 一般而言我们还是更推荐使用这种方法而不是上面介绍的那种。

我们仍然还是以前面的四则运算小程序为例，来说明Bison中的属性值类型机制是如何工作的。原先这个四则运算程序只能计算整数值，现在我们加入浮点数运算的功能。修改后的代码如下所示：

```
Calc → ε
| Exp
Exp → Factor
| Exp ADD Factor
| Exp SUB Factor
Factor → Term
| Factor MUL Term
| Factor DIV Term
Term → INT
| FLOAT
```

在这份语法中，我们希望词法单元INT能有整型属性值，而FLOAT能有浮点型属性值，为了简单起见，我们让其他的非终结符号都具有双精度型的属性值。这份语法以及类型方案对应的Bison源代码如下：

```
 1  %{
 2    #include <stdio.h>
 3  %}
 4
 5  /* declared types */
 6  %union {
 7    int type_int;
 8    float type_float;
 9    double type_double;
10  }
11
12   /* declared tokens */
13  %token <type_int> INT
14  %token <type_float> FLOAT
15  %token ADD SUB MUL DIV
16
17  /*  declared non-terminals */
18  %type <type_double> Exp Factor Term
19
20  %%
21  Calc :  /* empty */
22    | Exp  { printf("= %lf\n, $1); }
23    ;
24  Exp : Factor
25    | Exp ADD Factor  { $$ = $1 + $3; }
26    | Exp SUB Factor  { $$ = $1 - $3; }
27    ;
28  Factor : Term
```

```
29    | Factor MUL Term  { $$ = $1 * $3; }
30    | Factor DIV Term  { $$ = $1 / $3; }
31    ;
32  Term : INT  { $$ = $1; }
33    | FLOAT  { $$ = $1; }
34    ;
35
36  %%
37  …
```

首先，我们在定义部分的开头使用%union{…}将所有可能的类型都包含进去。接下来，在%token部分我们使用一对尖括号<>把需要确定属性值类型的每个词法单元所对应的类型括起来。对于那些需要指定其属性值类型的非终结符号而言，我们使用%type加上尖括号的办法确定它们的类型。当所有需要确定类型的符号的类型都被定下来之后，规则部分里的$$、$1等就自动地带有了相应的类型，不再需要我们显式地为其指定类型了。

2.2.11 Bison：词法单元的位置

实践要求中需要输出每一个语法单元出现的位置。我们当然可以自己在Flex中定义每个行号和列号，并在每个动作中维护这个行号和这个列号，同时将它们作为属性值的一部分返回给语法单元。这种做法需要我们额外编写一些维护性的代码，这非常不方便。Bison有没有内置的位置信息供我们使用呢？答案是肯定的。

前面介绍过在Bison中每个语法单元都对应了一个属性值，在语义动作中这些属性值可以使用$$、$1和$2等进行引用。实际上除了属性值之外，每个语法单元还对应了一个位置信息，在语义动作中这些位置信息同样可以使用@$、@1、@2等进行引用。位置信息的数据类型是 YYLTYPE，其默认的定义是：

```
1  typedef struct YYLTYPE {
2    int first_line;
3    int first_column;
4    int last_line;
5    int last_column;
6  }
```

其中的first_line和first_column分别是该语法单元对应的第一个词素出现的行号和列号，而last_line和last_column分别是该语法单元对应的最后一个词素出现的行号和列号。有了这些内容，输出所需的位置信息就比较方便了。但应注意，如果直接引用@1、@2等将每个语法单元的first_line打印出来，会发现打印出来的行号全都是1。

为什么会出现这种问题呢？主要原因在于，Bison并不会主动替我们维护这些位置信息，我们需要在Flex源代码文件中自行维护。不过只要稍加利用Flex中的某些机制，维护这些信息并不需要编写大量代码。我们可以在Flex源文件的开头部分定义变量yycolumn，并添加宏定义YY_USER_ACTION：

```
%option yylineno
…
%{
  /* 此处省略#include 部分 */
  int yycolumn = 1;
  #define YY_USER_ACTION \
    yylloc.first_line = yylloc.last_line = yylineno; \
    yylloc.first_column = yycolumn; \
    yylloc.last_column = yycolumn + yyleng - 1; \
    yycolumn += yyleng;
%}
```

在上述代码中，yylineno是内置变量，用于记录当前处理的行号；yylloc也是内置变量，表示当前词法单元所对应的位置信息；YY_USER_ACTION是宏，表示在执行每一个动作之前需要先被执行的一段代码，默认为空，而这里我们将其改成了对位置信息的维护代码。除此之外，最后还要在Flex源代码文件中做更改，即发现换行符之后对变量yycolumn进行复位：

```
…
%%
…
\n  { yycolumn = 1; }
```

另外，我们要在Bison源代码文件的第一部分加上%locations：

```
%locations
…
%%
…
```

在上述代码中，%locations指令用于启用Bison的位置跟踪功能，使Bison能够记录每个语法符号在源代码中的位置信息。

这样就可以实现在Bison中正确打印位置信息。

2.2.12 Bison：二义性与冲突处理

二义性是文法设计时的常见问题（如本书 2.1.5 节的文法示例所示）。Bison 有一个非常好用但也很恼人的特性：对于一个有二义性的文法，它有自己的一套隐式的冲突解决方案（一旦出现归约/归约冲突，Bison 总会选择靠前的产生式；一旦出现移入/归约冲突，Bison 总会选择移入）从而生成相应的语法分析程序，而这些冲突解决方案在某些场合可能并不是我们所期望的。

因此，我们建议在使用 Bison 编译源代码时要留意它所给的提示信息，如果提示文法有冲突，那么请一定对源代码进行修改，尽量处理完所有的冲突。

前面那个四则运算的小程序，如果它的语法变成这样：

```
Calc → ε
| Exp
Exp → Factor
| Exp ADD Exp
| Exp SUB Exp
…
```

虽然看起来好像没什么变化（Exp → Exp ADD Factor | Exp SUB Factor变成了Exp → Exp ADD Exp | Exp SUB Exp），但实际上前面之所以没有这样写，是因为这样做会引入二义性。例如，如果输入为1 – 2 + 3，语法分析程序将无法确定先算1 – 2还是2 + 3。语法分析程序在读到1 – 2的时候可以归约（即先算1 – 2）也可以移入（即先算2 + 3），但由于Bison默认移入优先于归约，语法分析程序会继续读入+ 3然后计算2 + 3。

为了解决这里出现的二义性问题，要么重写语法（Exp → Exp ADD Factor | Exp SUB Factor相当于规定加减法为左结合），要么显式地指定运算符的优先级与结合性。一般而言，重写语法是一件比较麻烦的事情，而且会引入不少像Exp → Term这样除了增加可读性之外 没任何实质用途的产生式。所以更好的解决办法还是考虑优先级与结合性。

在Bison源代码中，我们可以通过“%left”、“%right”和“%nonassoc” 对终结符号的结合性进行规定，其中“%left”表示左结合，“%right”表示右结合，而“%nonassoc”表示不可结合。例如，下面这段结合性的声明代码主要针对四则运算、括号以及赋值号：

```
1  %right ASSIGN
2  %left ADD SUB
3  %left MUL DIV
4  %left LP RP
```

其中ASSIGN表示赋值号，LP表示左括号，RP表示右括号。此外，Bison也规定任何 排在后面的运算符其优先级都要高于排在前面的运算符。因此，这段代码实际上还规定括号优先级高于乘除、乘除高于加减、加减高于赋值号。在实践内容一所使用的C--语言里，表达式Exp的语法便是冲突的，因此，需要模仿前面介绍的方法，根据C--语言的文法补充说明中的内容为运算符规定优先级和结合性，从而解决掉这些冲突。

另外一个在程序设计语言中很常见的冲突就是嵌套if-else所出现的冲突（也被称为悬空else问题）。考虑C--语言的这段语法：

```
Stmt → IF LP Exp RP Stmt
| IF LP Exp RP Stmt ELSE Stmt
```

假设我们的输入是：if (x > 0) if (x == 0) y = 0; else y = 1;，那么语句最后的这个else是属于前一个if还是后一个if呢？标准C语言规定在这种情况下else总是匹配距离它最近的那个if，这与Bison的默认处理方式（移入/归约冲突时总是移入）是一致的。因此即使我们不在Bison源代码里对这个问题进行任何处理，最后生成的语法分析程序的行为也是正确的。但如果不处理，Bison总是会提示我们该语法中存在一个移入/归约冲突。有没有办法把这个冲突去掉呢？

显式地解决悬空else问题可以借助于运算符的优先级。Bison源代码中每一条产生式后面都可以紧跟一个%prec标记，指明该产生式的优先级等同于一个终结符号。下面这段代码通过定义一个比ELSE优先级更低的LOWER_THAN_ELSE运算符，降低了归约相对于移入ELSE的优先级：

```
1  …
2  %nonassoc LOWER_THAN_ELSE
3  %nonassoc ELSE
4  …
5  %%
6  …
7  Stmt : IF LP Exp RP Stmt  %prec LOWER_THAN_ELSE
8    |  IF LP Exp RP Stmt ELSE Stmt
```

这里ELSE和LOWER_THAN_ELSE的结合性其实并不重要，重要的是当语法分析程序读到IF LP Exp RP时，如果它面临归约和移入ELSE这两种选择，它会根据优先级自动选择移入ELSE。通过指定优先级的办法，我们可以避免Bison在这里报告冲突。

前面我们通过优先级和结合性解决了表达式和if-else语句里可能出现的二义性问题。事实上，有了优先级和结合性的帮助，我们几乎可以消除语法中所有的二义性，但我们不建议使用它们解决除了表达式和if-else之外的任何其他冲突。原因很简单：只要是Bison报告的冲突，都有可能成为语法中潜在的一个缺陷，这个缺陷很可能源于一些我们没有意识到的语法问题。我们敢使用优先级和结合性的方法来解决表达式和二义性的问题，是因为我们对冲突的来源非常了解，除此之外，只要是Bison认为有二义性的语法，大部分情况下我们也能看出语法存在二义性。此时要做的不是掩盖这些语法上的问题，而是仔细对语法进行修订，发现并解决语法本身的问题。

2.2.13 Bison：源代码的调试

以下这部分内容是可选的，跳过不会对实践内容一的完成产生影响。

在使用Bison进行编译时，如果增加-v参数，那么Bison会在生成.yy.c文件的同时帮我们多生成一个.output文件。例如，执行

```
bison -d -v syntax.y
```

命令后，会在当前目录下发现一个新文件syntax.output，这个文件中包含Bison所生成的语法分析程序对应的LALR状态机的一些详尽描述。如果在使用Bison编译的过程中发现自己的语法里存在冲突，但无法确定冲突出现在何处，就可以阅读这个.output文件，里面对于每一个状态所对应的产生式、该状态何时进行移入何时进行归约、语法有多少冲突，以及这些冲突在哪里等都有十分完整的描述。

例如，如果我们不处理前面提到的悬空else问题，.output文件的第一句就会是：

```
1  state 112 conflicts: 1 shift/reduce
```

继续向下翻，找到状态112，.output文件对该状态的描述为：

```
1  State 112
2
3  36 Stmt : IF LP Exp RP Stmt .
4  37   | IF LP Exp RP Stmt . ELSE Stmt
5
6  ELSE shift, and go to state 114
7
8  ELSE [reduce using rule 36 (Stmt)]
9  $default reduce using rule 36 (Stmt)
```

这里我们发现，状态112在读到ELSE时既可以移入又可以归约，而Bison选择了前者，将后者用方括号括了起来。知道是这里出现了问题，我们就可以以此为线索修改Bison源代码或者重新修订语法了。

对于一个有一定规模的语法规范（如C--语言）而言，Bison所产生的LALR语法分析程序可以有一百甚至几百个状态。即使将它们都输出到了.output文件里，在这些状态里逐个寻找潜在的问题也是挺费劲的。另外，有些问题，（例如语法分析程序在运行时刻出现“Segmentation fault”等），很难从对状态机的静态描述中发现，必须要在动态、交互的环境下才容易看出问题所在。为了达到这种效果，在使用Bison进行编译的时候，可以通过附加-t参数打开其诊断模式（或者在代码中加上#define YYDEBUG 1）：

```
bison -d -t syntax.y
```

在main函数调用yyparse()之前我们加一句：yydebug = 1;，然后重新编译整个程序。之后运行这个程序就会发现，语法分析程序现在正像一个自动机一样，一个一个状态地在进行转换，并将当前状态的信息打印到标准输出上，以方便用户检查自己的代码哪里出现了问题。以前面的那个四则运算小程序为例，在打开诊断模式之后运行程序，屏幕上会出现如下字样：

```
1  Starting parse
2  Entering state 0
3  Reading a token:
```

如果我们输入4，会明显地看到语法分析程序出现了状态转换，并将当前栈里的内容打印出来：

```
 1  Next token is token INT ()
 2  Shifting token INT ()
 3  Entering state 1
 4  Reducing stack by rule 9 (line 29):
 5    $1 = token INT ()
 6  →$$ = nterm Term ()
 7  Stack now 0
 8  Entering state 6
 9  Reducing stack by rule 6 (line 25):
10    $1 = nterm Term ()
11  →$$ = nterm Factor ()
12  Stack now 0
13  Entering state 5
14  Reading a token:
```

继续输入其他内容，我们可以看到更进一步的状态转换。

注意，诊断模式会使语法分析程序的性能下降不少，建议在不使用时不要随便打开。

2.2.14 Bison：错误恢复

当输入文件中出现语法错误的时候，Bison总是会让它生成的语法分析程序尽早地报告错误。当语法分析程序从yylex()得到了一个词法单元，如果当前状态并没有针对这个词法单元的动作，那Bison就会认为输入文件里出现了语法错误，此时它默认进入如下错误恢复模式：

（1）调用 yyerror("syntax error")，该函数默认会在屏幕上打印出 syntax error 的字样。

（2）从栈顶弹出所有还没有处理完的规则，直到语法分析程序回到了一个可以移入特殊符号 error 的状态。

（3）移入 error，然后对输入的词法单元进行丢弃，直到找到一个能够跟在 error 之后的符号为止（该步骤也被称为再同步）。

（4）如果在 error 之后能成功移入三个符号，则继续正常的语法分析；否则，返回前面的步骤二。

这些步骤看起来似乎很复杂，但实际上需要我们做的事情只有一件，即在语法里指定error符号应该放到哪里。不过，需谨慎考虑放置error符号的位置：一方面，我们希望error后面跟的内容越多越好，这样再同步就会更容易成功，这提示我们应该把error尽量放在高层的产生式中；另一方面，我们又希望能够丢弃尽可能少的词法单元，这提示我们应该把error尽量放在底层的产生式中。在实际应用中，人们一般把error放在例如行尾、括号结尾等地方，本质上相当于让行结束符“;”以及括号“{”、“}”、“(”、“)”等作为错误恢复的同步符号：

```
Stmt → error SEMI
CompSt → error RC
Exp → error RP
```

以上几个产生式仅仅是示例，并不意味着把它们照搬到Bison源代码中就可以让语法分 析程序满足实践内容一的要求。需要进一步思考如何书写包含error的产生式才能够检查出输入文件中存在的各种语法错误。

2.2.15 语法分析实践的额外提示

想要做好一个语法分析程序，第一步要仔细阅读并理解C−−语法规范。C−−的语法要比它的词法更复杂，如果缺乏对语法的理解，在调试和测试语法分析程序时将感到无所适从。另外，如果没弄懂C−−语法中每条产生式背后的具体含义，则无法在后面的实践内容二中去分析这些产生式的语义。

接下来，我们建议先写一个包含所有语法产生式但不包含任何语义动作的Bison源代码，然后将它和修改以后的Flex源代码、main函数等一块编译出来先看看效果。对于一个没有语法错误的输入文件而言，这个程序应该什么也不输出；对于一个包含语法错误的输入文件而言，这个程序应该输出syntax error。如果我们的程序能够成功地判别有无语法错误，再去考虑优先级与结合性该如何设置以及如何进行错误恢复等问题；如果程序输出的结果不对，或者说程序根本无法编译，

那我们需要重新阅读前文并仔细检查哪里出了问题。好在此时代码并不算多，借助于Bison的.output文件以及诊断模式等，要查出错误并不是太难的事情。

再下一步需要考虑语法树的表示和构造。语法树是一棵多叉树，因此为了能够建立它需要实现多叉树的数据结构。我们需要专门写函数完成多叉树的创建和插入操作，然后在Bison源代码文件中修改属性值的类型为多叉树类型，并添加语义动作，将产生式右边的每 一个符号所对应的树节点作为产生式左边的非终结符号所对应的树节点的子节点逐个进行插入。这棵多叉树的数据结构怎么定义，以及插入操作怎么完成等问题完全取决于我们自己的设计，在设计过程中有一点需要注意：在后续实践内容中我们还会在这棵语法树上进行一些其他的操作，所以现在的数据结构设计会对后面的语义分析产生一定的影响。

构造完这棵树之后，下一步就是按照实践内容一要求中提到的缩进格式将它打印出来，同时要求打印的还有行号以及一些相关信息。为了能打印这些信息，需要写专门的代码对生成的语法树进行遍历。由于要求打印行号，因此在之前生成语法树的时候就需要将每个节点第一次出现时的行号都记录下来（使用位置信息@n、使用变量yylineno，或者自己维护这个行号均可以）。这段负责打印的代码仅是为了实践内容一而写，后面的实践不会再用，所以我们建议将这些代码组织到一起或者写成函数接口的形式，以方便日后对代码进行调整。

2.3 词法分析和语法分析的实践内容

本节主要介绍具体的实践内容。实践要求来自 2.1 节中的理论基础，例如完成基本的词法分析和语法分析功能，这包括词法单元流的生成（2.1.1 节至 2.1.5 节）、语法分析树的生成（2.1.6 节至 2.1.9 节），以及基本的错误处理等。

在完成相应实践内容时可以参考 2.2 节中的实践技术指导部分，基于 Flex（2.2.2 节至 2.2.5 节）完成词法分析部分，基于 Bison（2.2.8 节至 2.2.14 节）完成语法分析部分，并根据具体的实践要求进行额外的设计与改进。

2.3.1 实践要求

实践中所需要编写的程序要能够查出 C−−源代码中可能包含的下述几类错误。

（1）**词法错误（错误类型 A）：**出现 C−−词法中未定义的字符以及任何不符合 C−−词法单元定义的字符；

（2）**语法错误（错误类型 B）：**出现不符合 C−−语法中定义语法规则的语句。

除此之外，程序可以选择完成以下部分或全部的要求：

（1）**要求 2.1：**识别八进制数和十六进制数。若输入文件中包含符合词法定义的八进制数（如 0123）和十六进制数（如 0x3F），程序需要得出相应的词法单元；若输入文件中包含不符合词法定义的八进制数（如 09）和十六进制数（如 0x1G），程序需要给出输入文件有词法错误（即错误类型 A）的提示信息。八进制数和十六进制数的定义参见附录 A。

（2）**要求 2.2：**识别指数形式的浮点数。若输入文件中包含符合词法定义的指数形式的浮点数（如 1.05e-4），程序需要得出相应的词法单元；若输入文件中包含不符合词法定义的指数形式的浮点数（如 1.05e），程序需要给出输入文件有词法错误（即错误类型 A）的提示信息。指数形式的浮点数的定义参见附录 A。

（3）**要求 2.3：**识别“//”和“/*…*/”形式的注释。若输入文件中包含符合定义的“//”和“/*…*/”形式的注释，程序需要能够滤除这样的注释；若输入文件中包含不符合定义的注释（如“/*…*/”注释中缺少“/*”），程序需要给出由不符合定义的注释所引发的错误的提示信息。注释的定义参见附录 A。

程序在输出错误提示信息时，需要输出具体的错误类型、出错的位置（源程序行号）以及相关的说明文字。

2.3.3 输入格式

程序的输入是一个包含 C−−源代码的文本文件，程序需要能够接收一个输入文件名作为参数。例如，假设程序名为 cc、输入文件名为 test1、程序和输入文件都位于当前目录下，那么在 Linux 命令行下运行./cc test1 即可获得以 test1 作为输入文件的输出结果。

2.3.3 输出格式

实践内容一要求通过标准输出打印程序的运行结果。对于那些包含词法或者语法错误的输入文件，只要输出相关的词法或语法有误的信息即可。在这种情况下，注意不要输出任何与语法树有关的内容。要求输出的信息包括错误类型、出错的行号以及说明文字，其格式为：

```
Error type [错误类型] at Line [行号]: [说明文字].
```

说明文字的内容没有具体要求，但是错误类型和出错的行号一定要正确，因为这是判断输出的错误提示信息是否正确的唯一标准。请严格遵守实践要求中给定的错误分类（即词法错误为错误类型 A，语法错误为错误类型 B），否则将影响评分。注意，输入文件中可能会包含一个或者多个错误（但输入文件的同一行中保证不出现多个错误），程序需要将这些错误全部报告出来，每一条错误提示信息在输出中单独占一行。

对于那些没有任何词法或语法错误的输入文件，程序需要将构造好的语法树按照先序遍 历的方式打印每一个节点的信息，这些信息包括：

（ 1 ）如果当前节点是一个语法单元并且该语法单元没有产生 ε（即空串），则打印该语法单元的名称以及它在输入文件中的行号（行号被括号所包围，并且与语法单元名之间有一个空格）。某个语法单元在输入文件中的行号是指该语法单元产生出的所有词素中的第一个在输入文件中出现的行号。

（2 ）如果当前节点是一个语法单元并且该语法单元产生了 ε ,则无须打印该语法单元的信息。

（3 ）如果当前节点是一个词法单元，则只要打印该词法单元的名称，而无须打印该词法单元的行号。

1 ）如果当前节点是词法单元 ID，则要求额外打印该标识符所对应的词素。

2 ）如果当前节点是词法单元 TYPE，则要求额外打印说明该类型是 int 还是 float。

3 ）如果当前节点是词法单元 INT 或者 FLOAT，则要求以十进制的形式额外打印该数字所对应的数值。

4）词法单元所额外打印的信息与词法单元名之间以一个冒号和一个空格隔开。

每一条词法或语法单元的信息单独占一行，而每个子节点的信息相对于其父节点的信息来说，在行首都要求缩进 2 个空格。具体输出格式可参见后续的样例。

2.3.4 验证环境

所编写的程序将在如下环境中被编译并运行：

- GNU Linux Release: Ubuntu 20.04, kernel version 5.13.0-44-generic；
- GCC version 7.5.0；
- GNU Flex version 2.6.4；
- GNU Bison version 3.5.1。

一般而言，只要避免使用过于冷门的特性，使用其他版本的 Linux 或者 GCC 等，也基本上不会出现兼容性方面的问题。注意，实践内容一的检查过程中不会去安装或尝试引用各类方便编程的函数库（如 glib 等），因此请不要在程序中使用它们。

2.3.5 提交要求

实践内容一要求提交如下内容：

（1）Flex、Bison 以及 C 语言的可被正确编译运行的源程序。

（2）一份 PDF 格式的实验报告，内容包括：

1) 你的程序实现了哪些功能？简要说明如何实现这些功能。清晰的说明有助于助教对你的程序所实现的功能进行合理的测试。
2) 你的程序应该如何被编译？可以使用脚本、makefile 或逐条输入命令进行编译，请详细说明应该如何编译程序。无法顺利编译将导致助教无法对程序所实现的功能进行任何测试，从而丢失相应的分数。
3) 实验报告的长度不得超过三页！所以实验报告中需要重点描述的是程序中的亮点，是开发人员认为最个性化、最具独创性的内容，而相对简单的、任何人都可以做的内容则可不提或简单地提一下，尤其要避免大段地向报告里贴代码。实验报告中所出现的最小字号不得小于 5 号字（或英文 11 号字）。

2.3.6 样例（必做部分）

实践内容一的样例包括必做部分与选做部分，分别对应于实践要求中的必做内容和选做要求。请仔细阅读样例，以加深对实践要求以及输出格式要求的理解。这节列举必做内容样例。

样例 1：

输入（行号是为标识需要，并非样例输入的一部分，后同）：

```
1  int main()
2  {
3    int i = 1;
4    int j = ~i;
5  }
```

输出：

这个程序存在词法错误。第 4 行中的字符“~”没有在我们的 C−−词法中被定义过，因此程序可以输出如下的错误提示信息：

```
Error type A at Line 4: Mysterious character "~".
```

样例 2：

输入：

```
1  int main()
2  {
3    float a[10][2];
4    int i;
5    a[5,3] = 1.5;
6    if (a[1][2] == 0) i = 1 else i = 0;
7  }
```

输出：

这个程序存在两处语法错误。其一，虽然我们的程序中允许出现方括号与逗号等字符，但二维数组正确的访问方式应该是 a[5][3]而非 a[5,3]。其二，第 6 行的 if-else 语句在 else 之前少了一个分号。因此程序可以输出如下的两行错误提示信息：

```
Error type B at Line 5: Missing "]".
Error type B at Line 6: Missing ";".
```

样例 3：

输入：

```
1  int inc()
2  {
3    int i;
4    i = i + 1;
5  }
```

输出：

这个程序非常简单，也没有任何词法或语法错误，因此你的程序需要输出如下的语法树 节点信息：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        TYPE: int
      FunDec (1)
        ID: inc
        LP
        RP
      CompSt (2)
        LC
        DefList (3)
          Def (3)
            Specifier (3)
              TYPE: int
            DecList (3)
              Dec (3)
                VarDec (3)
                  ID: i
            SEMI
        StmtList (4)
          Stmt (4)
            Exp (4)
              Exp (4)
                ID: i
              ASSIGNOP
              Exp (4)
                Exp (4)
                  ID: i
                PLUS
                Exp (4)
                  INT: 1
              SEMI
        RC
```

样例 4：

输入：

```
struct Complex
{
  float real, image;
};
int main()
{
   struct Complex x;
   y.image = 3.5;
}
```

输出：

这个程序虽然包含了语义错误（即使用了未定义的变量 y），但不存在任何词法或语法错误，因此程序不能报错而是要输出相应的语法树节点信息。至于把该语义错误检查出来的任务，我们则放到实践内容二中去做。本样例输入所对应的正确输出应为：

```
Program (1)
  ExtDefList (1)
    ExtDef (1)
      Specifier (1)
        StructSpecifier (1)
          STRUCT
          OptTag (1)
            ID: Complex
          LC
          DefList (3)
            Def (3)
              Specifier (3)
                TYPE: float
              DecList (3)
                Dec (3)
                  VarDec (3)
                    ID: real
                COMMA
                DecList (3)
                  Dec (3)
                    VarDec (3)
                      ID: image
              SEMI
          RC
      SEMI
    ExtDefList (5)
      ExtDef (5)
        Specifier (5)
          TYPE: int
        FunDec (5)
          ID: main
          LP
          RP
        CompSt (6)
          LC
          DefList (7)
            Def (7)
              Specifier (7)
                StructSpecifier (7)
                  STRUCT
                  Tag (7)
                    ID: Complex
              DecList (7)
                Dec (7)
                  VarDec (7)
                    ID: x
              SEMI
          StmtList (8)
            Stmt (8)
              Exp (8)
                Exp (8)
                  Exp (8)
```

```
53                  ID: y
54                DOT
55                ID: image
56              ASSIGNOP
57              Exp (8)
58                FLOAT: 3.500000
59            SEMI
60          RC
```

2.3.7 样例（选做部分）

本节列举选做要求样例。

样例 1：

输入：

```
1  int main()
2  {
3    int i = 0123;
4    int j = 0x3F;
5  }
```

输出：

如果程序需要完成要求 2.1，该样例输入不包含任何词法或语法错误，其对应的输出为：

```
1  Program (1)
2    ExtDefList (1)
3      ExtDef (1)
4        Specifier (1)
5          TYPE: int
6        FunDec (1)
7          ID: main
8          LP
9          RP
10       CompSt (2)
11         LC
12         DefList (3)
13           Def (3)
14             Specifier (3)
15               TYPE: int
16             DecList (3)
17               Dec (3)
18                 VarDec (3)
19                   ID: i
20                 ASSIGNOP
21                 Exp (3)
22                   INT: 83
23             SEMI
24           DefList (4)
25             Def (4)
26               Specifier (4)
27                 TYPE: int
28               DecList (4)
29                 Dec (4)
30                   VarDec (4)
31                     ID: j
32                   ASSIGNOP
```

```
33                 Exp (4)
34                   INT: 63
35             SEMI
36         RC
```

样例 2：

输入：

```
1  int main()
2  {
3    int i = 09;
4    int j = 0x3G;
5  }
```

输出：

如果程序需要完成要求 2.1，该样例程序中的“09”为错误的八进制数，其会因被识别为十进制整数“0”和“9”而引发语法错误；同样，“0x3G”为错误的十六进制数，其也会因被识别为十进制整数“0”和标识符“x3G”而引发语法错误。因此程序可以输出如下的错误提示信息：

```
Error type B at Line 3: Missing ";".
Error type B at Line 4: Missing ";".
```

程序也可以直接将“09”和“0x3G”分别识别为错误的八进制数和错误的十六进制数，此时程序也可以输出如下的错误提示信息：

```
Error type A at Line 3: Illegal octal number '09'.
Error type A at Line 4: Illegal hexadecimal number '0x3G'.
```

样例 3：

输入：

```
1  int main()
2  {
3    float i = 1.05e-4;
4  }
```

输出：

如果程序需要完成要求 2.2，该样例程序不包含任何词法或语法错误，其对应的输出为：

```
 1  Program (1)
 2    ExtDefList (1)
 3      ExtDef (1)
 4        Specifier (1)
 5          TYPE: int
 6        FunDec (1)
 7          ID: main
 8          LP
 9          RP
10        CompSt (2)
11          LC
12          DefList (3)
13            Def (3)
14              Specifier (3)
```

```
15              TYPE: float
16            DecList (3)
17              Dec (3)
18                VarDec (3)
19                  ID: i
20                ASSIGNOP
21                Exp (3)
22                  FLOAT: 0.000105
23            SEMI
24        RC
```

样例 4：

输入：

```
1  int main()
2  {
3    float i = 1.05e;
4  }
```

输出：

如果程序需要完成要求 2.2，该样例程序中的“1.05e”为错误的指数形式的浮点数，其会因被识别为浮点数“1.05”和标识符“e”而引发语法错误。因此程序可以输出如下的错误提示信息：

```
Error type B at Line 3: Syntax error.
```

程序也可以直接将“1.05e”识别为错误的指数形式的浮点数，此时程序也可以输出如下的错误提示信息：

```
Error type A at Line 3: Illegal floating point number "1.05e".
```

样例 5：

输入：

```
1  int main()
2  {
3     // line comment
4     /*
5    block comment
6     */
7     int i = 1;
8  }
```

输出：

如果程序需要完成要求 2.3，该样例程序不包含任何词法或语法错误，其对应的输出为：

```
1  Program (1)
2    ExtDefList (1)
3      ExtDef (1)
4        Specifier (1)
5          TYPE: int
6        FunDec (1)
7          ID: main
8          LP
9          RP
```

```
10      CompSt (2)
11        LC
12        DefList (7)
13          Def (7)
14            Specifier (7)
15              TYPE: int
16            DecList (7)
17              Dec (7)
18                VarDec (7)
19                  ID: i
20                ASSIGNOP
21                Exp (7)
22                  INT: 1
23            SEMI
24        RC
```

注意，助教检查程序的时候，所使用的测试用例中“//”注释的范围与“/*…*/”注释的范围不会重叠（以减少问题的复杂性），因此程序不需要专门处理“//”注释的范围与“/*…*/”注释的范围相重叠的情况。

样例 6：

输入：

```
1  int main()
2  {
3    /*
4    comment
5    /*
6    nested comment
7    */
8    */
9    int i = 1;
10 }
```

输出：

样例输入中程序员主观上想使用嵌套的“/*…*/”注释，但 C--语言不支持嵌套的“/*…*/”注释。如果程序需要完成要求 2.3，该样例输入中第 8 行的“*/”会因被识别为乘号“*”和除号“/”而引发语法错误（只需要为每行报告一个语法错误），程序可以输出如下的错误提示信息：

```
Error type B at Line 8: Syntax error.
```

2.4 本章小结

本章介绍了编译器工作流程中的前两个关键步骤，即词法分析和语法分析。词法分析将源代码从字符流转换为词法单元序列。语法分析则负责检验该词法单元序列是否符合语法规范。在词

法分析的理论部分，我们重点讨论了正则表达式的基本概念和原理，并分析了非确定的有限状态机和确定的有限状态机之间的转换和技术特点。在语法分析的理论部分，我们介绍了上下文无关文法，以及自顶向下和自底向上两大类语法分析算法，以形成完整的编译器词法和语法分析步骤。此外，本章中也讨论了对应的技术实践内容，通过对该部分的学习，可以编写一个程序对使用 C--语言书写的源代码进行基本的词法分析和语法分析，并打印出相应的分析结果。

习题

2.1 将下列正则表达式转换成 NFA 和 DFA：

（1）$(a^* \mid b^*)b\,(ba)^*$

（2）$(a^*b)^*ba(a \mid b)^*$

2.2 构造表示“标识符”的正则表达式，其中标识符的定义为：以字母开头的字母数字串，标识符可以有后缀，其后缀是用“-”或者“.”隔开的字母数字串。将所构造的正则表达式转换成 DFA，画出相应的转换图。

2.3 假设“标识符“的定义如下：小写字母开头的（大小写）字母数字串，并且不包含连续的大写字母。设计一个接受该“标识符”的 DFA（转换图形式）。

2.4 对下面的文法 G：

$T \rightarrow FT'$

$T' \rightarrow T \mid \varepsilon$

$F \rightarrow PF'$

$F' \rightarrow *F' \mid \varepsilon$

$P \rightarrow (T) \mid a \mid b \mid \wedge$

（1）计算这个文法的每个非终结符的 FIRST 集和 FOLLOW 集。

（2）证明这个文法是 LL(1)的。

2.5 为下面的文法构造预测分析表。你可能先要对文法进行提取左公因子或消除左递归的操作，并计算该文法的 FIRST 和 FOLLOW 集合。

$S \rightarrow (L) \mid a\ S \mid a$

$L \rightarrow L, S \mid S$

2.6 某语言的增广文法 G 为：

（1）$S' \rightarrow T$

（2）$T \rightarrow aBd \mid \varepsilon$

（3）$B \rightarrow Tb \mid \varepsilon$

证明 G 不是 LR(0)文法而是 SLR(1)文法，并给出 SLR(1)分析表。

2.7 给定文法 G[S]： $S \rightarrow (S) \mid a$ *（知识拓展）*

（1）构造识别文法 G[S]可行前缀的 LR(1)项目的 DFA；

（2）构造 LR(1)分析表；

（3）构造 LALR(1)分析表。