编译原理

一、 准备

1. 为什么要学习编译原理的课程

a) 考研

b) 编译器开发

c) 形式语言翻译程序开发

d) 加深对 Java 知识的理解，为工作或者面试加分

对象初始化顺序，重载，多态等知识点会有更加深刻的理解。

2. 需要的基础知识

a) 熟悉一门编程语言

具体什么语言不限，课程中使用 java 语言进行演示讲解。

b) 数据结构和算法

数据结构：链表，集合，有序集合，哈希表，数组等。

算法：搜索（深度优先和广度优先）

c) 设计模式

简单工厂

抽象工厂

责任链

Facde

状态模式

d) 数学

离散数学和高等数学

3. 如何学习编译原理课程

完成好每一次留下的作业

二、 编译原理理论

1. 概述

a) 编译原理理论是干什么的

编译原理是指导编译程序的开发，而编译程序把高级语言源代码 编译 成机器语言 的程序。

b) 程序设计语言主要分以下 3 类:

- 机器语言

机器语言指不经翻译即可为机器直接理解和接受的程序语言或指令代码, 也称第一代计算机语言。

- 汇编语言

汇编语言, 用一些容易理解和记忆的字母, 单词来代替一个特定的指令, 比如: 用“ADD”

代表加减运算，“MOV”代表数据传递等，也称第二代计算机语言。汇编语言的执行，由汇编编译程序，将汇编源程序编译成机器语言后，再由计算机执行。

- 高级语言

高级语言的执行，需要由高级语言编译程序将源代码编译成机器语言，然后再由在计算机执行。

c) 程序设计语言转换

i. 翻译

把某种语言的源代码，在不改变语义的条件下，转换成另一种语言源代码。比如可以把 java 源代码翻译成 C#的源代码，反之也行。

ii. 编译

把高级语言转化为机器代码，这就是我们此次课程重点要学习的内。

iii. 解释

对高级语言的逐条进行翻译，并让计算机执行，马上得到这句的执行结果，然后再解释执行下一句。Javascript 就是一种常见的解释执行的高级语言。

2. 文法表示

编译原理课程中，所有的词法和语法规则，我们都用文法来表示，所以我们先来学习文法的相关知识。

a) Chomsky 文法定义

开始学习文法之前，我看看 java 语言中 package 语句的语法结构。

1. 没有包名；
2. package com;（package 后面只跟一个标识符，然后分号）
3. package com.msb;（package 后面只跟一个标识符，然后跟 1 个点加标识符，然

后分号）

4. package com.msb.compiler;（package 后面只跟一个标识符，然后点和标识符重复两次，然后分号）
5. package com.msb.compiler.A;（package 后面只跟一个标识符，然后点和标识符重复 3 次，然后分号）

能不能根据上面的描述，推导出 package 语句语法结构的一般形式。

1. 没有包名
2. package Identifier(.Identifier)*;

有没有更加严谨，更加规范，大家都认可的方式，来定义这种的语法结构。答案就是 Chomsky 文法定义。Chomsky 是美国一个伟大的哲学家，他发表的《句法结构》被认为是 20 世纪，理论语言学，研究最伟大的贡献。

Chomsky 文法定义如下：从形式上说文法 G 是一个四元式(V_N ,V_T ,P, S)，通俗的说，文法 G 式由 4 个部分构成。

对于识别 java package 语句的文法，如下所示：

G = (V_N ,V_T ,P, S)

V_N = { Package, PackageName, PackageNameFollow, Identifier }

V_T = { package , com,msb,compiler, . , ; , ε}

S= Package

P={

Package -> ε

Package -> package PackageName ;

PackageName –> Identifier

PackageName -> Identifier PackageNameFollow

PackageNameFollow -> . Identifier

PackageNameFollow -> . Identifier PackageNameFollow

Identifier -> com|msb|compiler|……

}

i. 非终结符

非终结符表示一定语法概念的词，可以进行扩展和分割，非终结符集合用 V_N 表示。

V_N = { Package, PackageName, PackageNameFollow, Identifier }

非终结符一般使用大写字母表示，

ii. 终结符

语言中不可再分割的语法单位，终结符集合用 V_T 表示。

V_T = { package , com,msb,compiler,　.　,　;　,　ε}

非终结符一般使用非大写字母表示

iii. 开始符号

表示所定义的语法范畴的开始，开始符号式非终结符。用 S 表示

S = Package

iv. 产生式

是用来定义符号串之间关系的一组(语法)规则。

Package -> package PackageName ;

Package -> ε

PackageName -> Identifier PackageNameFollow

PackageNameFollow -> . Identifier PackageNameFollow

PackageNameFollow -> ε

Identifier -> com | msb | compiler

以上式 6 条产生式。

v. 推导

推导是从开始符号开始，通过使用产生式的右部替换左部，最终能产生语言的一个句子

的过程。

最左推导：每次使用最左的非终结符进行推导。

最右推导：每次使用最右的非终结符进行推导。

最左推导和最右推导称为规范推导，其他形式的推导都是非规范推导。

对于一个合法的包名

package com.msb.compiler;的推导过程

1. Package -> package PackageName ;

2. Package -> ε

3 .PackageName -> Identifier PackageNameFollow

4. PackageNameFollow -> . Identifier PackageNameFollow

5. PackageNameFollow -> ε

6. Identifier -> com | msb | compiler

最左推导

Package

-> package PackageName ; （1）

-> package Identifier PackageNameFollow; （3）

-> package com PackageNameFollow; （6）

-> package com . Identifier PackageNameFollow; （4）

-> package com.msb PackageNameFollow; （6）

-> package com . msb . Identifier PackageNameFollow; （4）

-> package com . msb . compiler PackageNameFollow; （6）

-> package com . msb . compiler;（5）

最终就识别了这个句子。

最右推导

1. Package -> package PackageName ;

2. Package -> ε

3 .PackageName -> Identifier PackageNameFollow

4. PackageNameFollow -> . Identifier PackageNameFollow

5. PackageNameFollow -> ε

6. Identifier -> com | msb | compiler

Package

-> package PackageName ; （1）

-> package Identifier PackageNameFollow; （3）

-> package Identifier . Identifier PackageNameFollow; （4）

-> package Identifier . Identifier . Identifier PackageNameFollow; （4）

-> package Identifier . Identifier . Identifier; （5）

-> package Identifier . Identifier. compiler; （6）

-> package Identifier. msb . compiler;（6）

-> package com.msb.compiler;（6）

最终就识别成功。

对于一个不合法的包名，大家试一试能不能从上面的文法推导出来。

package ;

package .com;

package com.;

package com.msb.;

vi. 规约

归约是推导的逆过程，即从原始的句子开始，通过规则的左部取代右部，最终达到开始符号的过程。

最左规约：每次使用最左的符号进行规约，最左规约是最右推导的逆过程。

最右规约：每次使用最右的符号进行规约，最右规约是最左推导的逆过程。

最左归约和最右归约称为规范归约，其他形式的规约是非规范规约。

package com. msb.compiler;的规约过程

1. Package -> package PackageName ;

2. Package -> ε

3 .PackageName -> Identifier PackageNameFollow

4. PackageNameFollow -> . Identifier PackageNameFollow

5. PackageNameFollow -> ε

6. Identifier -> com | msb | compiler

最右规约

package com . msb . compiler;

<- package com . msb . compiler PackageNameFollow; （5）

<- package com . msb . Identifier PackageNameFollow; （6）

<- package com.msb PackageNameFollow; （4）

<- package com . Identifier PackageNameFollow;（6）

<- package com PackageNameFollow;（4）

<- package Identifier PackageNameFollow;（6）

<- package PackageName ;（3）

<- Package（1）

最左规约：

1. Package -> package PackageName ;

2. Package -> ε

3 .PackageName -> Identifier PackageNameFollow

4. PackageNameFollow -> . Identifier PackageNameFollow

5. PackageNameFollow -> ε

6. Identifier -> com | msb | compiler

package com.msb.compiler;

<- package Identifier. msb . compiler; （6）

<- package Identifier . Identifier. compiler; （6）

<- package Identifier . Identifier . Identifier;（6）

<- package Identifier . Identifier . Identifier PackageNameFollow;（5）

<- package Identifier . Identifier PackageNameFollow;（4）

<- package Identifier PackageNameFollow;（4）

<- package PackageName ;（3）

<- Package （1）

规约成功，最终一定到达的文法的开始符号，如果不能规约的文法的开始符号，就规约失败，同样对于以下非法的 package 语句，大家尝试一下规约过程。

package ;

package .com;

package com.;

package com.msb.;

vii. 句型

句型是从文法的开始符号 S 开始，每步推导（包括 0 步推导）所得到的字符串。

Package

-> package PackageName ;

-> package identifier PackageNameFollow;

-> package com PackageNameFollow;

-> package com . identifier PackageNameFollow;

-> package com.msb PackageNameFollow;

-> package com . msb . identifier PackageNameFollow;

-> package com . msb . compiler PackageNameFollow;

-> package com . msb . compiler;

viii. 句子

是仅含终结符的句型

package com . msb . compiler;

ix. 语言

语言是由开始符号 S 开始通过 1 步或 1 步以上推导所得的句子的集合。

x. 文法规则的递归定义

非终结符的定义中包含了非终结符自身，有了这种递归，语言就变成无穷集合。

PackageNameFollow -> . identifier PackageNameFollow

xi. 元语言符号

用来说明文法符号之间关系的符号，如，“->”和“|”称为元语言符号。

PackageNameFollow -> . identifier PackageNameFollow

PackageNameFollow -> ε

以上的两个产生式，也可以改写成如下一条产生式：

PackageNameFollow -> . identifier PackageNameFollow | ε

b) 文法分类

0 型文法

0 型文法又叫短语文法或无限制文法，只要求产生式的左部，至少包含一个非终结符。

1 型文法

1 型文法又称为长度增加文法、上下文有关文法，1 型文法有以下几个限制条件：

a. 对文法 G 的中每一条产生式 α -> β ,除了 S -> ε 外均有$|\beta|>=|\alpha|$;

b. 若有 S -> ε ,规定 S 不得出现在产生式右部。

举例：

S -> aAa

aA -> Ba

AB->Cdd

都是合法的 1 型文法产生式。

而对于 AB->a 这样的产生式，就不是 1 型文法产生式，构成的文法就不是 1 型文法

2 型文法

2 型文法也称为上下文无关文法，要求产生式左部一定是一个非终结符。

S->AB

A->a

B->b

都是合法的 2 型文法产生式。

而对于 aA -> Ba，AB->Cdd 这样的产生式，就不是 2 型文法产生式。

3 型文法

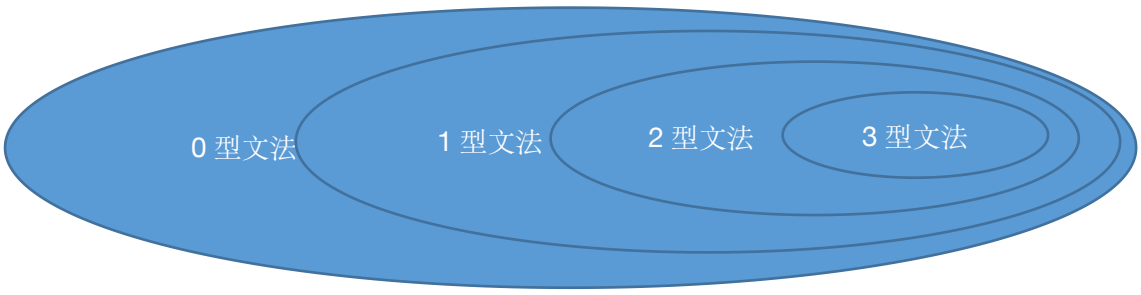
3 型文法也称为正规文法 RG。正规文法又分左线性文法或右线性文法，要求 P 中产生式具有形式

$A\rightarrow \alpha B$，　$A\rightarrow \alpha$，（右线性）或者

$A\rightarrow B\alpha$，　$A\rightarrow \alpha$，（左线性）其中 $A，B\in V_N$，　$\alpha \in V_T^*$

3 型文法中的产生式要么都是右线性产生式，要么是左线性产生式，不能既有左线性产生式，又有右线性产生式；若所有产生式均是左线性，则称为左线性文法；若所有产生式均是右线性，则称为右线性文法

这些概念了解一下就可以了，4 种文法类型的包含关系，如下图：



c) 语法树和文法二义性

语法树用来表示句子结构的树，使用语法树标识，可以使语法分析过程直观、形象。

如果文法的一个句子存在对应的两棵或两棵以上的语法树，则该句子是二义的，包含二

义性句子的文法是二义文法。

说明：

1）二义性会给语法分析带来不确定性，

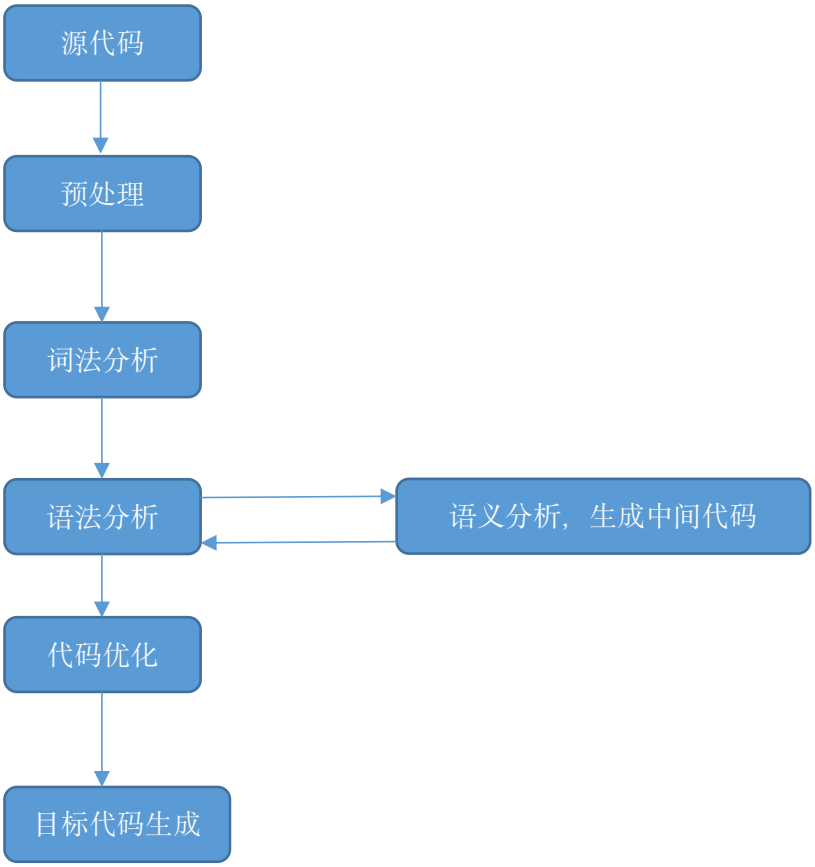
2）文法的二义性是不可判定的，即不存在算法，能够在有限步数内确切判定一个文法是否为二义文法。若要证明是二义性，只要举出一例即可。

3. 编译过程

a) 编译程序的工作流程

为了使 编译程序的 逻辑结构更加清晰，本次课程采用多遍扫描的方式进行讲解。

源代码 -> 预处理 -> 词法分析 -> 语法分析 -> 语义分析生成中间代码 -> 中间代码优化 -> 目标代码生成

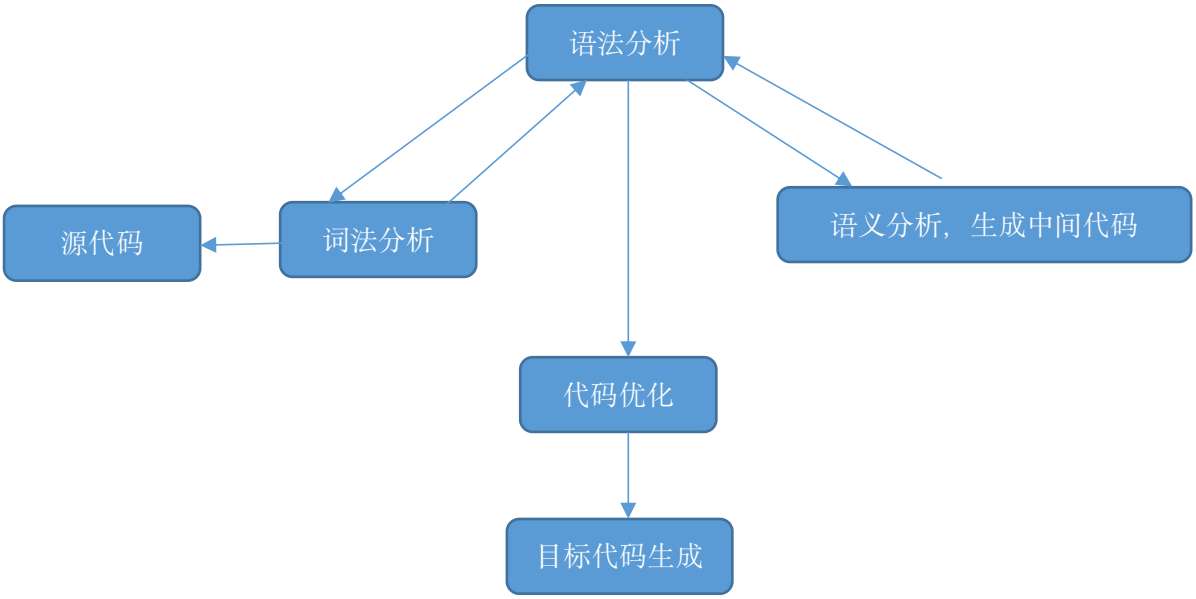


多遍扫描实现的优点：编译程序的逻辑结构非常清晰，每一个编译过程只做一件事，各个过程互不影响，互相无感知。

缺点：编译时间较长。

为了提高编译效率，还有一遍扫描的处理过程。

一遍扫描，以语法分析为中心，需要一个新的单词，就调用词法分析获取，词法分析就返回一个 Token 给语法分析。语法分析每识别一个的句子，就调用相应的语义分析子程序，生成中间代码，也返回给语法分析程序。



一遍扫描的实现方式优点：编译效率高

缺点：编译程序的逻辑结构会变复杂。

如果想把多遍扫描的实现方式，重构成一遍扫描的方式，也不困难。系列课程结束后，有兴趣的同学可以尝试一下，课上不做讲解。

听完上面的编译过程，可能大家对此还是很陌生。

通常，在遇到我们不熟悉的事物的时候，我们会用熟悉的，类似的，已知的，事物进行类比，来帮助我们理解。

编译过程，和英语翻译成汉语的过程非常类似，我们来对比一下。

b) 英语翻译成汉语过程

编译过程		翻译过程	
源程序	//包名定义 package /*多行注释*/ com.msb.compiler ;	英文源文	You are beautiful !
预处理	package com.msb.compiler;	去掉无用的字符	You are beautiful!
词法分析	package 关键字 com 标识符 . 分界符 msb 标识符 . 分界符 compiler 标识符 ; 分界符	查字典，找到每一个单词的中文含义，并设置单词类型, 名词, 动词，形容词	You 你（第二人称代词） are 是（be 的第二人称单复数现在时，第一、三人称复数现在时） beautiful 美丽的，漂亮的（形容词）
语法分析	java 包名的定义语法结构： 包 名 为 ： com.msb.compiler	分析整个句子的语法结构，主谓宾，主系表, 还是其他的结构, 时态是一般现在时, 还是一般过去时，还是一般将来时。	主系表结构, 时态是一般现在时：
语义分析生成中间代码	（definepackage, com.msb.compiler null,null）	根据句子的含义进行初步翻译	你是漂亮的!
中间代码优化	按照等价变换的原则，对语义分析的中间代码进行转化，来生成执行效率更高的目标代码。	对翻译内容进行润色，使句子的表达更加生动形象。（beautiful 直接翻译成美丽不能表达情	你有沉鱼落雁之容,闭月羞花之貌。

		感, 翻译 成鱼落雁闭月羞花)	
目标代码生成	Java 源代码编译完的目标代码是 class 文件	得到翻译结果	你有沉鱼落雁之容,闭月羞花之貌。

c) 预处理

输入：源代码

去掉所有对编译无有的字符：

注释（单行注释//和多行注释/**/）

多余的空白字符（空格，回车和换行，缩进），

但是需要注意三点：

1.字符串，双引号里面包含的任何东西，都是程序的一部分，不能去掉；

2.单个字符 char，单引号里面包含的内容，也是程序的一部分，同样不能去掉。

3.转义字符的处理（\r\n\t \"）。

d) 词法分析

词法分析依照词法规则，识别出正确的单词，提供给语法分析程序使用。

输入：预处理后的源代码

输出：单词（或者叫 Token 串）

单词是高级语言中 具有实际意义的 最小语法单位，它由字符构成。

单词需要分类，一般分 5 大类：关键字，标识符，常量，运算符，分界符。这是最常见，也是最简单的分类方法。

当然，也可以有更加精确的细分，常量可以再分为数字常量和字符串常量，数字常量还可以细分整数常量和浮点数常量，整数常量还可以细分为 int 型整数常量和 long 型整数常量，浮点数常量还可以细分为 float 单精度浮点数常量和 double 双精度浮点数常量。

运算符又可以分为条件运算符，数值运算符和位运算符等等。

具体细分到什么程度，可以有开发着自己决定。词法分析细分好了类，语法分析就可以直接使用。词法分析不细分，就由语法分析来分类。

举例：

```
package com;
public final class Main {
    private static char V4 = 'a';
    public static void main(String[] args) {
        System.out.println(1 + 2020);
    }
}
```

对于以上源程序，有那些单词构成？

关键字：package public final class private static char void

标识符：com Main args System out println V4

常量：'a' 1 2020

运算符：+

分界符： ( ) { } , ;

e) 语法分析

根据语法规则，把词法分析的单词，识别成的语法单位：短语、语句、过程、程序。

输入：词法分析的单词串

输出：语法树

语言的规则，又称为文法，规定单词如何构成短语、语句、过程和程序。

语法分析方法有两种方式：

自上而下分析方法（推导），

还有一种自下而上分析方法：规约

f) 语义分析和中间代码生成

对语法分析识别出的短语、语句、过程、程序，分析其含义，进行初步翻译，产生介于源代码和目标代码之间的一种代码。

常见的中间代码形式： 四元式、三元式、逆波兰式

最常用的中间代码形式是四元式，我们这只讲四元式，其他形式，如果大家有兴趣，可以自己去了解。

对于 x = a + b * c - d 语句，语义分析后生成如下的四元式

操作符	左操作数	右操作数	中间结果
*	b	c	T1
+	a	T1	T2
-	T2	d	T3
=	T3	x	null

g) 中间代码优化

按照等价变换的原则，对语义分析的中间代码进行转化，以生成执行效率更高的目标代码。

常用的优化方法：

公共子表达式提取

举例：

```
int a = 1,b = 2,c =3;
int e = a + b + c;
int d = a + b + d;
```

等级转化：

```
int a,b,c =1,2,3;
int T1 = a + b;
int e = T1 + c;
int d = T1 + d;
```

合并已知量

删除无用语句

循环优化

h) 目标代码生成

把经过优化的中间代码转化成特定机器上的低级语言代码。

目标代码的形式

- 绝对指令代码: 可立即执行的目标代码。
- 汇编指令代码：汇编语言程序，需要通过汇编程序 编译后 后才能运行。
- 可重定位指令代码：先将各目标模块连接起来，确定变量、常数在主存中的位置，装入主存后才能成为可以运行的绝对指令代码

上面的几种形式，大家简单了解一下就可以了，本次课程要生成的目标代码是 java 的 class 文件。

4. 词法分析

a) 预处理

1. 去掉单行和多行注释
2. 去掉多余的空白字符
3. 处理转义字符

过程化处理

过程化处理方式，比较简单，我们直接上代码。

状态模式处理

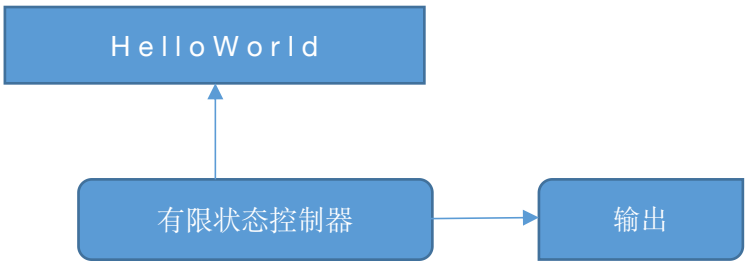
使用设计模式中的状态模式进行处理。

b) 自动机处理

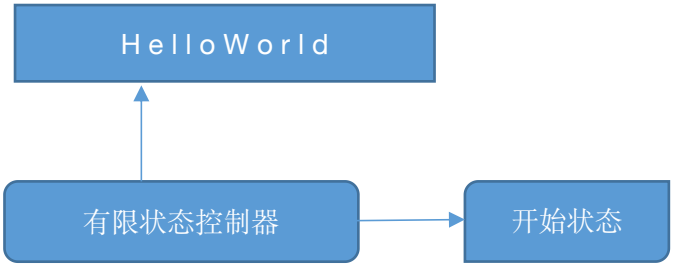
i. 有限自动机(Finite Automation,FA)

有限自动机是具有离散输入输出系统的数学模型。它具有有限数目的内部状态，系统可以根据当前所处的状态和面临的输入字符决定系统的后继行为。其当前状态概括了过去输入处理

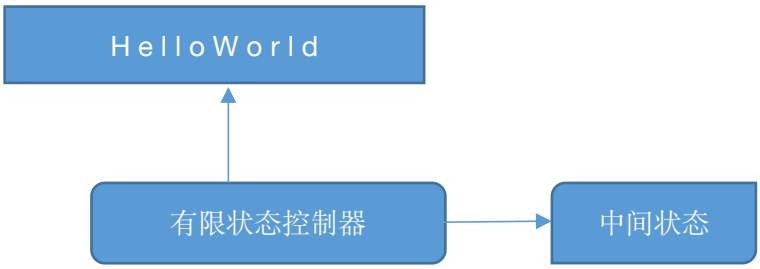
的信息。



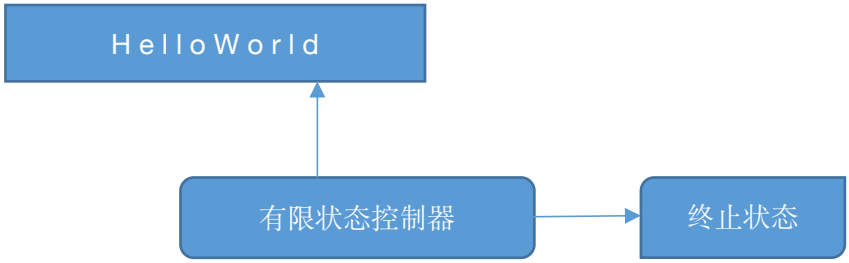
说明：状态分为初始状态、中间状态和终止状态。终止状态可以有若干个，而初始状态一般只有一个。
程序开始：



不断读入字符



读到结尾



只要当数据全部读完，且此时状态为终止状态，则说明此输入串正确，其他情况都说明输入串不能识别。

ii. 确定有限自动机 DFA(Deterministic FA)

定义：

确定有限自动机是一个五元式 $M(S, \Sigma, f, S_0, Z)$

- S：有限状态集
- Σ ：有限字母表
- f ：状态转换关系，$f(S_1, a) = S2$ 其中 $S_1 \in S$， $S_2 \in S$ ，$a \in \Sigma$
- S_0 ：唯一的初态，$S_0 \in S$
- Z ：终止状态集，$Z \subseteq S$

说明：状态转换关系中，对于给定的当前状态 S_1 和当前读入的符号 a，有唯一确定的下一状态 S_2。

状态转换关系可以有两种方式标识：一种是状态转换图，另外一种是状态转换矩阵。

对于预处理程序的状态转换矩阵如下：

S = { Start , String ,转义， End，注释 S ，单行注释，多行注释，多行注释结束}

Σ ={“ , \ , rnbt”\ , ^”\, / , * , \n , ^\r\n\t\b}

S_0 = { start }

Z ={End}

	”	\	rnbt"\	^”\	/	*	^*	\n	^\n
Start	String			End	注释 S				
String	End	转义		String					
转义			String						
End									
注释 S					单行注释	多行注释			
单行注释								End	单行注释
多行注释						多行注释结束	多行注释		
多行注释结束					End				

FA 的识别：

串 $\alpha \in \Sigma^*$为 DFA $M=(S, \Sigma, f, S_0, Z)$ 所识别，当且仅当$(S_0, \alpha) \vdash^* (S, \varepsilon)$,且 $s \in Z$

使用上面定义的 DFA 来识别“1\t2\n3”

状态	α	⊢（一步动作）
Start	“1\t2\n3”	String
String	1\t2\n3”	String
String	\t2\n3”	转义
转义	t2\n3”	String
String	2\n3”	String
String	\n3”	转义
转义	n3”	String
String	3”	String
String	“	End
End	ε	识别

说明：

如果读完输入串，状态不停在终止状态，即： $(s\ 0, \alpha) \vdash^* (s', \varepsilon)$,且 $s' \notin Z$

或者

在读过程中出现不存在的映射，使自动机无法继续动作。

出现以上两种情况，说明 FA 不能识别该串。

举例：使用上面定义的 FA 来识别“1\t2\n3

状态	α	⊢
Start	“1\t2\n3	String
String	1\t2\n3c	String
String	\t2\n3	转义
转义	t2\n3	String
String	2\n3	String
String	\n3	转义
转义	n3	String
String	3	String

String	ε	α 读完，而 String 不在终态集，DFA 不能识别该串

使用上面定义的 DFA 来识别“1\t2\s3”

状态	α	⊢
Start	“123\t\s”	String
String	1\t2\s3	String
String	\t2\s3	转义
转义	t2\s3	String
String	2\s3	String
String	\s3	转义
转义	s3	出错，转义状态读到 s，找不到后续状态，无法识别，\s 是不合法的转义字符。

iii. 不确定有限自动机 NFA(Non-deterministic FA)

定义：

不确定有限自动机是一个五元式 $M(S, \Sigma, f, S_0, Z)$

- 其中 S：有限状态集
- Σ ：有限字母表
- f ：状态转换关系，$f(S_1, a) = \{S_2, S_3, S_4, ...\}$ 其中 $S_1 \in S$， $S_2 \in S$，$S_3 \in S$，$S_4 \in S$ ，$a \in \Sigma$
- S_0 ：唯一的初态，$S_0 \in S$
- Z ：终止状态集，$Z \subseteq S$

说明：状态转换关系中，对于给定的当前状态 S_1 和当前读入的符号 a，有多个后继状态。

DFA 是 NFA 的一个特例。

举例：

M(S, Σ , f , S_0 , Z)

S = {S0，S1，S2，S3，S4 , S5}

Σ={0 , 1， ε }

f 状态转换矩阵如下表

S_0 = S0

Z = {S5}

	0	1	ε
S0			S1，S4
S1	S2	S2	
S2	S1,S3	S1	
S3			S5
S4		S2	
S5			

S2 状态，读入 0，可以跳转到两个状态 S1 和 S3，所以上面的 FA，是 NFA，而不是 DFA。

iv. NFA 转 DFA

自动机等价：任何两个有限自动机 M_1 和 M_2，若它们识别的语言相同则称 M_1 和 M_2 等价。

子集算法可以把一个 NFA 转换为 DFA。

什么要把 NFA 转 DFA，转完有什么意义？在识别的过程中，计算机只能做确定的事情，对于上面的 NFA，识别过程中，如果 S2 状态读入 0，到底是跳转到 S1 状态还是 S3 状态，不能确定，给识别带来困难，这个是时候需要把 NFA 进行确定化。

先说一下 ε -closure 的求法：

(a)若 s ∈ I，则 s ∈ ε -closure(I)

(b)若 s ∈ I，那么从 s 出发经过任意段的 ε 弧所能达到的任意状态 s’都属于 ε -closure(I)

ε -closure	结果
ε -closure（S0）	{S0,S1,S4}
ε -closure（S1）	{S1}
ε -closure（S2）	{S2}
ε -closure（S3）	{S3,S5}
ε -closure（S4）	{S4}
ε -closure（S5）	{S5}

子集算法：

- 子集可以将一个 NFA M=(S, Σ ,f, S_0 ,Z)构造一个等价的

  DFA M'=(Q, Σ ,f', I_0 ,F)

算法过程如下：

1.取 I_0 = ε -closure (S_0),

2.若状态集 Q 中有状态 I_i ={ S_0, S_1, S_2 ,……S_j} , S_k∈S , 0 ≤ k≤ j;并且

f(I_i,a)

= f(S_0 ,a) ∪f(S_1 ,a) ∪f(S_2 ,a)……∪f(S_j,a)

={ S_0 , S_1, S_2 ,……S_t } = I_t

若 closure (I_t) 不在 Q 中，则将 closure (I_t)加入 Q。

3.重复步骤 2，直到 Q 中无新状态加入为止。

4.取终态 F={I | I ∈ Q,且 I ∩ Z <>Φ}

举例：对以下得 NFA 进行确定化

M(S, Σ , f , S_0 , Z)

S = {S0，S1，S2，S3，S4 , S5}

Σ ={0 , 1， ε }

f 状态转换矩阵如下表

S_0 = S0

Z = {S5}

	0	1	ε

S0			S1，S4
S1	S2	S2	
S2	S1,S3	S1	
S3			S5
S4		S2	
S5			

编号	新状态集 Q	0	1
I0	ε -closure (S_0) ={S0,S1,S4}	f(I0,0) =f(S0,0) ∪ f(S1,0) ∪ f(S4,0) =Φ∪{S2}∪Φ ={S2} ε -closure (S2)={S2}	f(I0,1) =f(S0,1) ∪ f(S1,1) ∪ f(S4,1) =Φ∪{S2}∪{S2} ={S2} c
I1	{S2}	f(I1,0) =f(S2,0) ={S1,S3} ε -closure ({S1,S3})= ε -closure (S1) ∪ ε -closure (S3) ={S1}∪{S3,S5} ={S1,S3,S5}	f(I1,1) =f{S2,1} ={S1} ε -closure (S1)={S1}
I2	{S1,S3,S5}	f(I2,0) =f(S1,0) ∪ f(S3,0) ∪ f(S5,0) ={S2}∪Φ∪Φ ={S2} ε -closure (S2)={S2}	f(I2,1) =f(S1,1) ∪ f(S3,1) ∪ f(S5,1) ={S2}∪Φ∪Φ ={S2} ε -closure (S2)={S2}

I3	{S1}	f(I3,0) =f(S1,0) ={S2} ε -closure (S2)={S2}	f(I3,1) =f(S1,1) ={S2} ε -closure (S2)={S2}

设置新的状态，重新编号

I0 = {S0,S1,S4}

I1 = {S2}

I2 = {S1,S3,S5}

I3 = {S1}

转换后的 DFA 为

M(S, Σ , f , S_0 , Z)

S = {I0，I1, I2 ,I3}

Σ ={0 , 1}

f 状态转换矩阵如下表

S_0 = {I0}

Z = { I2 }

	0	1
I0	I1	I1
I1	I2	I3
I2	I1	I1

I3	I1	I1

v. DFA 最小化

为什么要进行 DFA 最小化：减小 DFA 状态状态表的存储空间，以达到节省内存目的。

DFA 最小化算法：

1.把状态集 Q 划分为新的集合 P，P={终态集，非终态集}

2 任取 P 一个子集 Pi={I1 ,I2 ,……Ik },若存在某读入字符 a，使 f(Pi ,a)的结果不是全部包含在 Pj 的某个子集中，

则说明 Pi 中有不等价的状态，还要进一步划分（新创建一个状态 Pk，把 f(Ii,a)不包含在 Pj 的状态划分出去）。

3 重复步骤 2，直到没有可以新划分的子集，算法结束。

举例，对于下面的 DFA，进行最小化

M(S, Σ , f , S_0 , Z)

S = {I0,I1, I2 ,I3}

Σ ={0 , 1}

f 状态转换矩阵如下表

S_0 = {I0}

Z = {I2}

	0	1
I0	I1	I1
I1	I2	I3
I2	I1	I1
I3	I1	I1

编号	新分组 index	新分组	0	1
P2	2	终态集={I2}	F(I2,0) =I1	F(I2,1) =I1
	1	非 终 态 集	F(I0,0)=I1	

		={I0,I1,I3}	F(I1,0)=I2 F(I3,0)=I1	
P0	1	非终态集 ={I0 ,I3}	F(I0,0)=I1 F(I3,0)=I1	F(I0,1)=I1 F(I3,1)=I1
P2	3	非终态集 1={I1}	F(I1,0)=I2	F(I1,1)=I3

设置新的状态，重新编号

P0 = {I0,I3 }

P1={I2 }

P2={I1}

最小化后的 DFA 为

M(S, Σ , f , S_0 , Z)

S = {P0,P1,P2 }

Σ ={0 , 1}

f 状态转换矩阵如下表

	0	1
P0	P2	P2
P1	P2	P2
P2	P1	P0

S_0 = {P0}

Z = { P1 }

vi. DFA 应用

1.词法分析预处理

2.词法分析

3.Json 反序列化

```
[
    true,
    false,
    1,
```

```
        2,
        null,
        {
            "abc":{
                "cd":12
            }
        },
        [
            1,
            2
        ]
    ]
```

5. 语法分析

语法分析是编译程序的最核心部分，也是比较难的部分。

语法分析的任务是：根据语法规则，把词法分析结果（一个个单词），识别成的语法单位：短语、语句、过程、程序。

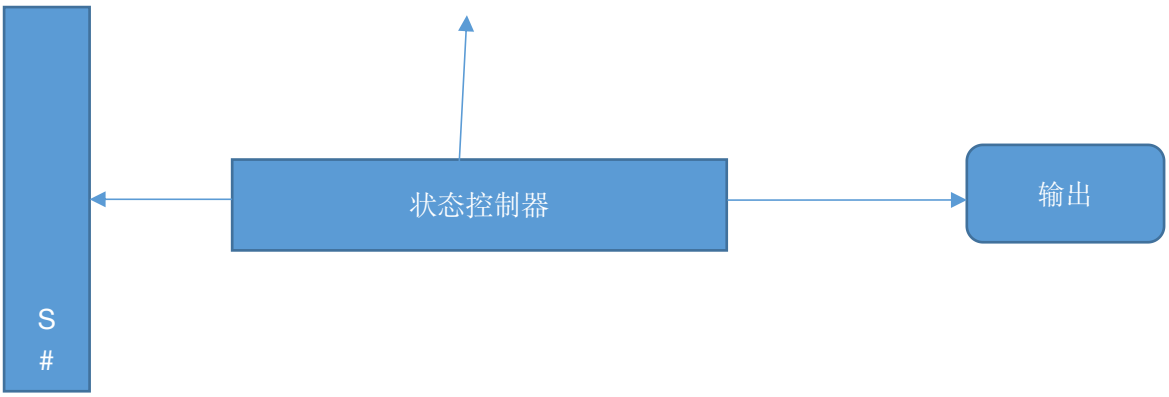
输入：词法分析的单词串

输出：语法树

语法分析的理论基础 ：上下文无关文法和下推自动机

a) 下推自动机 PDA（push down automation）

```
package com.mab.compiler
```



1）PDA 和 FA 的模型相比，多了一个下推栈。

2）PDA 的动作由三个因素来决定：当前状态、读头所指向符号、下推栈栈顶符号。

3）一个输入串能被 PDA 所接受，仅当输入串读完，下推栈变空；或输入串读完，控制器到达某些终态。

4）正规文法和有限自动机 FA 仅适合于描述和识别高级语言的各类单词。而语句规则可用上下文无关文法来描述，而下推自动机又恰好能识别上下文无关文法所能描述的语言，因此上下文无关文法及其对应的下推自动机就成为编译技术中语法分析的理论基础。

b) 自上而下语法分析

从开始符号开始，使用不同产生式进行推导，最后与目标输入串相匹配。

i. 一般分析方法

算法过程

（1）若栈顶符号 x 是非终结符，查询语法表（文法产生式），找出一个以 x 作为左部的产生式，x 出栈，并将其右部反序入栈，且输出带记下产生式编号 —— 推导。

（2）若栈顶符号 x 是终结符，且读头下的符号也是 x，则 x 出栈，读头指向下一个符号—— 匹配。

（3）若栈顶符号 x 是终结符，但读头下的符号不是 x，则匹配失败。这说明可能前面推导时选错了候选式，退回到上次推导现场(包括栈顶符号、读头的指针和输出带上信息)—— 回溯。

举例

G = (VN ,VT ,P, S)

V_N = { Package, PackageName, PackageNameFollow, Identifier}

V_T = { package , com, msb, compiler , . , ; , ε}

P={

1：Package -> package PackageName ;

2：Package -> ε

3：PackageName -> Identifier PackageNameFollow

4：PackageNameFollow -> . Identifier PackageNameFollow

5：PackageNameFollow -> ε

6：Identifier -> com

7：Identifier -> msb

8：Identifier -> compiler

}

识别 package com.msb.compiler;

动作	栈	识别串
初始化	# Package	package com.msb.compiler;#
使用 1 产生式推导	# ; PackageName package	package com.msb.compiler;#
匹配 package	# ; PackageName	com.msb.compiler;#
使用 3 产生式推导	# ; PackageNameFollow Identifier	com.msb.compiler;#
使用 6 产生式推导	# ; PackageNameFollow com	com.msb.compiler;#
匹配 com	# ; PackageNameFollow	.msb.compiler;#
使用 4 产生式推导	# ; PackageNameFollow Identifier .	.msb.compiler;#
匹配.	# ; PackageNameFollow Identifier	msb.compiler;#
使用 7 产生式推导	# ; PackageNameFollow msb	msb.compiler;#
匹配 msb	# ; PackageNameFollow	.compiler;#
使用 4 产生式推导	# ; PackageNameFollow Identifier .	.compiler;#
匹配.	# ; PackageNameFollow Identifier	compiler;#

使用 8 产生式推导	# ; PackageNameFollow compiler	compiler;#
匹配 compiler	# ; PackageNameFollow	;#
使用 5 产生式推导	# ;	;#
匹配;	#	#
识别成功		

存在问题

1） 如果文法存在左递归，语法分析会无限循环下去；

2） 若一个非终结符存在多个候选式，选择哪个产生式进行推导完全是盲目的，如果对所有可能满足条件的产生式进行搜索，搜索不成功就会进行回溯，回溯会引起的大量时间和空间的消耗，严重影响编译程序的效率；

3） 如果被识别的语句是错的，算法无法指出错误的确切位置。

ii. LL(1)分析方法

为了解决上面的问题，LL（1）分析方法，就应运而生，我们来看看 LL(1)是如何解决一般分析方法产生的问题。

左递归消除

什么是左递归

直接左递归：

$P \rightarrow Pa$

举例：

#P

#aP

#aaP

#aaaP

间接左递归：文法存在产生式 $P \rightarrow Aa$, $A \rightarrow^{+} Pb$

举例：

A -> Ba

B -> Ca

C -> Aa | a

A -> Ba

-> Caa

-> Aaaa

直接左递归消除：

设有文法 $G=(V_N, V_T, P, S)$，其中产生式 P 为

$P \rightarrow P\alpha | \beta$ ，$\alpha \in V^+$ ，$\beta \in V^+$ 且 β 不以 P 开头（如果 β 以 P 开头，只需要提取 P 公因子即可）

将它转换为 等价式：

$P \rightarrow \beta P'$， $P' \rightarrow \alpha P' | \varepsilon$

证明等价性：

$P \rightarrow P\alpha | \beta$ 表示的语言是以 β 开始，后面跟 0 个或者多个 α

β	$P \rightarrow \beta$
$\beta\ \alpha$	P $\rightarrow P\alpha$ $\rightarrow \beta\ \alpha$
$\beta\ \alpha\ \alpha$	P $\rightarrow P\alpha$ $\rightarrow P\alpha\ \alpha$ $\rightarrow \beta\ \alpha\ \alpha$

$P \rightarrow \beta P'$， $P' \rightarrow \alpha P' | \varepsilon$ 表示的语言是以 β 开始，后面跟 0 个或者多个 α

β	P

	→β P’ →β
β α	P →β P’ →β α P’ →β α
β α α	P →β P’ →β α P’ →β α α P’ →β α α

消除文法的 ε

无 ε 产生式的上下文无关文法要满足条件

1. 产生式 P 中要么不含有 ε 产生式，要么只有 S → ε
2. 若 S →ε，则 S 不出现在任何产生式右部。

消除 ε 算法过程：

G=(V_N ,V_T ,P,S) (有 ε 产生式的上下文无关文法) ->

G’=(V'_N ,V'_T ,P’,S’) (无 ε 产生式的上下文无关文法)

（1） 由文法 G 找出所有经过若干步推导能推出 ε 的非终结符，放在 V_0 集合中；

（2） 若 V_0 集合中的某元素出现在某产生式的右端，则分别 ε 和其原型代入，并将新产生式加入 P’；

（3） 不满足上一条的 P 中其他产生式除去 ε 产生式后也加入 P’；

（4） 如果 P 中有产生式 S → ε，将它在 P’ 中改为 S’ → ε | S，S’ 是新的开始符号，把它加入 V_N， 形成 V'_N

举例：

G=({S},{a,b},P,S)

P ={

S →ε

S →aSbS

S →bSaS

}

第一步：计算 V_0	V_0={S}
第二步：	对于产生式 S →aSbS 产生的右部有两个 S 可以有 4 种代入组合(S,S) (S, ε)(ε , S) (ε , ε) S →aSbS S →aSb S →abS S →ab
	对于产生式 S →bSaS 产生的右部有两个 S 可以有 4 种代入组合(S,S) (S, ε)(ε , S) (ε , ε) S →bSaS S →bSa S →baS S →ba
第三步：	因为 S → ε S’ → ε | S

G’=(V’$_N$,V’$_T$,P’,S’)

V’$_N$ = { S’ , S}

V’$_T$ = {a,b}

P’ = {

S’ → ε | S

S →aSbS

S →aSb

S →abS

S →ab

S →bSaS

S →bSa

S →baS

S →ba

}

间接左递归消除

消除算法对文法要求：不能有 P -> P 的产生式，无 ε 产生式的上下文无关文法。

算法过程

1. 删除文法中 G 中 P -> P 的产生式；
2. 把文法 G 的所有非终结符按出现的先后顺序进行从小到大进行编号；
3.

```
for(int i = 1;i <= N;i++){
    for(int j = 1;j <= i – 1;j++){
        对于 Pi →Pj γ 的产生式，将 Pj → δ1 | δ2 | δ3|……| δn 代入 Pi →Pj γ
        得到 Pi → δ1 γ | δ2 γ | δ3 γ |……| δn γ
    }
}
```

4. 删去从文法开始符号不可达的非终结符产生式。

举例：

对于以下文法，如何消除间接左递归

G = {{R,Q,S},{a,b,c},P,S}

P={

R -> Sa | a

Q -> Rb | b

S -> Qc | c

}

S -> Qc -> Rbc -> Sabc

第一步：编号	R 编号为 1 Q 编号为 2 S 编号为 3	R -> Sa | a Q -> Rb | b S -> Qc | c
第二步：循环代入	i = 1,j = 1 不满足跳出	R -> Sa | a Q -> Rb | b S -> Qc | c
	i = 2,j = 1 找到形如 Q –> R 的产生式 Q -> Rb 代入 R Q -> Rb ->(Sa | a)b ->Sab | ab	R -> Sa | a Q -> Sab | ab | b S -> Qc | c
	i =2 , j=2 跳出循环	R -> Sa | a Q -> Sab | ab | b S -> Qc | c
	i =3，j = 1 找到形如 S -> R 的产生式，没有找到，循环继续	R -> Sa | a Q -> Sab | ab | b S -> Qc | c
	i =3，j = 2 找到形如 S -> Q 的产生式 S -> Qc 代入 Q S -> Qc ->(Sab | ab | b)c ->Sabc | abc | bc	R -> Sa | a Q -> Sab | ab | b S -> Sabc | abc | bc | c
	i =3，j = 3 跳出循环	R -> Sa | a Q -> Sab | ab | b S -> Sabc | abc | bc | c
第三步：删去从文	S 只能产生 S，不能产生 Q,R，所	S -> Sabc | abc | bc | c

法开始符号不可达的非终结符产生式	以 Q 和 R 属于不可达的非终结符，把 Q 和 R 为左端的产生式删除	
第四步：使用直接左递归的规则，进行文法改写	P → P α | β 改写为 P →β P’， P’ → α P’ | ε	P = S α = abc β = abc | bc | c 改写后： S -> (abc |bc | c) S’ S’ -> abc S’ | ε

回溯消除

回溯是指，在搜索过程中，寻找问题的解，当发现已不满足求解条件时，就“回溯”返回，尝试别的路径。回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当搜索到某一步时，发现原先选择并不最优或达不到目标，就退回一步重新选择，这种走不通就退回再走的技术为回溯法。

语法分析回溯产生的根本原因

进行推导时，若产生式中，一个非终结符存在多个候选式，选择哪个候选式进行推导存在不确定性。

G = (VN ,VT ,P, S)

VN = { Package, PackageName, PackageNameFollow, Identifier}

VT = { package , com, msb, compiler , . , ; , ε}

P={

1：Package -> package PackageName ;

2：Package -> ε

3：PackageName -> Identifier PackageNameFollow

4：PackageNameFollow -> . Identifier PackageNameFollow

5：PackageNameFollow -> ε

6：Identifier -> com

7：Identifier -> msb

8：Identifier -> compiler

}

当出现非终结符 Package 时，选择 1 号产生式还是 2 号产生式进行推导不确定；

当出现非终结符 PackageNameFollow 时，选择 4 号产生式还是 5 号产生式进行推导也是不确定；

消除回溯思路

对文法的任何非终结符, 若能根据当前待识别的符号, 准确的选择一个候选式进行推导, 那么回溯就可以消除。

注：之所以会产生回溯是因为在推导匹配的过程中存在虚假匹配。

消除回溯方法

提取公共左因子

对于产生式：

P -> aB 和 P –> aC 的情况，当前待识别的符号为 a 时，不能确定使用哪个产生式进行推导，这时需要对文法进行改写，提前公共左因子

P –> aB | aC

P-> aP’

P’->B|C

当前待识别的符号为 a 时，确定的使用 P-> aP’产生式进行推导（B 和 C 都不能推动出以 a 开始的串，如果 B 或者 C 以 a 开头，需要再提取公共左因子）。

预测分析

根据待识别符号选择候选式，使其第一个符号与读头下符号相同，或该候选式可推导出的第一个符号与读头下符号相同。这相当于向前看了一个符号，所以称为预测。

注：使用了预测之后，选择候选式不再是盲目的了，所以也就无需回溯。

计算产生式 α 首符集 First(α)

计算终结首符集 First(α) 设 α -> $X_1 X_2 X_3 X_4 \cdots\cdots X_n$ 其中 $X_i \in (V_N \cup V_T)$ $1 <= i <= n$, First(α)集合包括了 α 的所有可能推导的开头终结符或可能的 ε 。

数学表达：$First(\alpha) = \{ a \mid \alpha \rightarrow^{*} a\cdots\cdots , a \in V_T \}$

算法：

如何求 First(α)，算法分两步，第一步求 First (X_i)，第二步再求 First(α)

求 First (X_i)过程：

1.如果 $X_i \in V_T$，那么 First (X_i) = X_i

举例：

First(package) = { package }

First(;) = { ; }

2. 如果 $X_i \in V_n$ ，

规则 1：如果有一条产生式是 $X_i \rightarrow a\cdots\cdots$, $a \in V_T$，那么 $a \in$ First (X_i)；

规则 2：如果有一条产生式是 $X_i \rightarrow \varepsilon$，那么$\varepsilon \in$ First (X_i)。

举例

First(Package)

对于 Package -> package PackageName ;这条产生式，package$\in V_T$，按照规则 1，package$\in$ First (Package)

对于 Package -> ε 这条产生式，按照规则 2，ε$\in$ First (Package)

α -> $X_1 X_2 X_3 X_4\cdots\cdots X_n$ 求 First (α)过程

α -> $X_1 X_2 X_3 X_4\cdots\cdots X_n$ ，$X_i \in (V_N \cup V_T)$ 1 <= i <= n

```
First(α) = {};
for(int i = 1;i <= n;i++) {
    if( ε ∈ First( Xi )){
        First(α) = First(α) ∪ { First( Xi ) - ε }
        if(i == n){
            First(α) = First(α) ∪ ε
        }
    }
    else{
        First(α) = First(α) ∪ First(Xi)
        break;
    }
}
```

举例：

例 1：

G = (V_N ,V_T ,P, S)

V_N = { S,T}

V_T = { (,), ε }

P={

1: S -> T

2: T -> T (T)

3: T -> ε

}

非终结符	First
T	T -> ε 得出 $\varepsilon \in$ First (T) 所以 Fisrt(T) = { ε } T -> T (T) 看一个符号 T ,因为 $\varepsilon \in$ First (T) 所以 First (T) = First (T) $\cup$ { First(T) - ε } = { ε } 接着看第二个符号(, ($\in$ First (( ) First (T) = First (T) $\cup$First (( ) = { (, ε } 所以 First (T) = { (, ε }
S	S -> T $\varepsilon \in$ First (T) First(S) = First(S) $\cup$ { First(T) - ε } First(S) = { (} $\varepsilon \in$ First (T) First (S) = First(S) $\cup \varepsilon$ First (S) = { (, ε }

例 2：

G = (V_N ,V_T ,P, A)

V_N = { A,B,C,D,E}

V_T = { a,b,c,d,f,g}

P={

1: A -> B C c

2: A -> g D B

3: B -> b C D E

4: B -> ε

5: C -> D a B

6: C -> c a

7: D -> d D

8: D -> ε

9: E ->g A f

10: E ->c

}

非终结符	First
A	A -> B C c ε ∈ Fisrt(B) Fisrt(A) = Fisrt(A) ∪ { First(B) - ε } ={b} Fisrt(C)中没有 ε Fisrt(A) = Fisrt(A) ∪Fisrt(C) ={a,b,c,d} A -> g D B 计算 g ∈ Fisrt(A) 最终：Fisrt(A) = {a,b,c,d,g}
B	B -> b C D E B -> ε Fisrt(B) = {b, ε }
C	C -> D a B ε ∈ Fisrt(D) Fisrt(C) = Fisrt(C) ∪ { First(D) - ε } = {d} Fisrt(C) = Fisrt(C) ∪ Fisrt(a) ={d,a} C -> c a 计算出 c ∈ Fisrt(C) 最终：Fisrt(C) = {a,c,d}
D	D -> d D 计算出 d ∈ Fisrt(D) D -> ε 计算出 ε ∈ Fisrt(D) Fisrt(D) = {d, ε }
E	E ->g A f 计算出 g ∈ Fisrt(E) E ->c 计算出 c ∈ Fisrt(E) Fisrt(E) = {g , c }

例 3：

G = (V_N ,V_T ,P, S)

V_N = { Package, PackageName, PackageNameFollow}

V_T = { package , identifier , . , ; , ε}

P={

1: Package -> package PackageName ;

2: Package -> ε

3: PackageName -> identifier PackageNameFollow

4: PackageNameFollow -> . identifier PackageNameFollow

5: PackageNameFollow -> ε

}

非终结符	First
Package	Package -> package PackageName ; Package -> ε Fisrt(Package) = { package , ε }
PackageName	PackageName -> identifier PackageNameFollow Fisrt(PackageName) = { identifier}
PackageNameFollow	PackageNameFollow -> . identifier PackageNameFollow PackageNameFollow -> ε Fisrt(PackageNameFollow) = { . , ε}

使用终结首符集 First(α)进行预测分析

设 A 是非终结符，且 A→α|β，当 A 出现栈顶，且输入符号为 a 时，应如何选择 A 的候选式进行推导？这里分为四种情况：

(a)若 a∈First(α),而 a∉First(β)，则选 A→α

(b)若 a ∉ First(α),而 a ∈ First(β)，则选 A→β

(c) 若 a ∉ First(α),且 a ∉First(β),则表示输入有错

(d)若 a∈First(α),且 a ∈ First(β),则表示终结首符集相交，需改写文法，进行公因子提取。

注：对 First(α)={ ε }情况，留待讨论。

计算非终结符 A 的随符集 Follow(A)

Follow(A)集合是指在所有句型中紧跟 A 之后的终结符所组成的集合。

数学定义

Follow(A) $= \{ a|S \rightarrow^{+} \cdots Aa \cdots, a \in V_T \}$

算法

1)对文法开始符号 S，将 # 加入到 Follow(S)中；

2)若 B $\rightarrow \alpha A \beta$ 是文法 G 的一个产生式，则将 { First(β) - ε } 加入到 Follow(A)中;

3)若 B $\rightarrow \alpha A$ 是文法 G 的一个产生式，则将 Follow(B)加入到 Follow(A)中；

4)或 B $\rightarrow \alpha A \beta$ 是文法 G 的一个产生式,且 $\beta \rightarrow \varepsilon$ ，则将 Follow(B)加入到 Follow(A)中

例 1：

G = (V_N ,V_T ,P, S)

V_N = { S,T}

V_T = { (,), ε }

P={

1: S -> T

2: T -> T (T)

3: T -> ε

}

非终结符	First
T	{ (, ε }
S	{ (, ε }

非终结符	Follow
T	T -> T (T) 规则 2 第一个 T 后面 β = (T) { First(β) - ε }加入到 Follow(T) First(β) = { (} Follow(T) = { (} 第二个 T 后面 ） β =) { First(β) - ε }加入到 Follow(T) First(β) = {) } Follow(T) = { ( ) }

	S -> T 规则 3　Follow(S) 加入到 Follow(T) Follow(S) = {#} Follow(T) = {#，) , (}
S	S 开始符号 , # ∈ Follow(S) Follow(S) = { # }

例 2：

G = (V_N ,V_T ,P, A)

V_N = { A,B,C,D,E}

V_T = { a,b,c,d,f,g}

P={

1:　A -> B C c

2:　A -> g D B

3:　B -> b C D E

4:　B -> ε

5:　C -> D a B

6:　C -> c a

7:　D -> d D

8:　D -> ε

9:　E ->g A f

10:　E ->c

}

非终结符	First
A	{a,b,c,d,g}
B	{b，ε }
C	{a,c,d}
D	{d，ε }
E	{g , c }

1)对文法开始符号 S，将 # 加入到 Follow(S)中；

2)若 B →α A β 是文法 G 的一个产生式，则将 { First(β) - ε } 加入到 Follow(A)中;

3)若 B →α A 是文法 G 的一个产生式，则将 Follow(B)加入到 Follow(A)中；

4)或 B →α A β 是文法 G 的一个产生式,且β →ε ，则将 Follow(B)加入到 Follow(A)中

非终结符	Follow
A	A 是开始符号 Follow(A) = {#} E -> g A f Follow(A) = Follow(A) ∪ Fisrt(f) = { f , # }
B	A -> B C c Follow(B) = Follow(B) ∪ {Fisrt(C c) - ε } {Fisrt(C c) - ε } = Fisrt(C) = { a,c,d } Follow(B) = { a,c,d } A -> g D B Follow(B) = Follow(B) ∪Follow(A) = { a,c,d }∪{ f , # } = { a,c,d ,f , #} C -> D a B Follow(B) = Follow(B) ∪Follow(C) = { a,c,d ,f , #} ∪{c,d,g} ={a,c,d,f,g,#}
C	A -> B C c Follow(C) = Follow(C) ∪ Fisrt(c) = {c} B -> b C D E Follow(C) = Follow(C) ∪ { Fisrt(DE) – ε } Fisrt(DE) = { Fisrt (D) - ε } ∪{ Fisrt (E) - ε } = {d,g,c} Follow(C) = {c} ∪ {d,g,c} ={c,d,g}
D	A -> g D B Follow(D) = Follow(D) ∪{Fisrt(B) - ε } = {b} ε ∈ Fisrt(B) Follow(D) = Follow(D) ∪Follow(A) ={ b ,f , #} B -> b C D E Follow(D) = Follow(D) ∪{Fisrt(E) - ε } = {b,c,f,g,#} C -> D a B Follow(D) = Follow(D) ∪{Fisrt(a B) - ε } ={b,c,f,g,#} ∪{ a }

	= {a,b,c,f,g,#} D -> d D Follow(D) = Follow(D) ∪Follow(D) = {a,b,c,f,g,#}
E	B -> b C D E Follow(E)= Follow(E) ∪Follow(B) = {a,c,d,f,g,#}

例 3：

G = (V_N ,V_T ,P, Package)

V_N = { Package, PackageName, PackageNameFollow}

V_T = { package , identifier , . , ; , ε}

P={

1: Package -> package PackageName ;

2: Package -> ε

3: PackageName -> identifier PackageNameFollow

4: PackageNameFollow -> . identifier PackageNameFollow

5: PackageNameFollow -> ε

}

非终结符	First
Package	{ package , ε }
PackageName	{ identifier}
PackageNameFollow	{ . , ε}

1)对文法开始符号 S，将 # 加入到 Follow(S)中；

2)若 B →α A β 是文法 G 的一个产生式，则将 { First(β) - ε } 加入到 Follow(A)中;

3)若 B →α A 是文法 G 的一个产生式，则将 Follow(B)加入到 Follow(A)中；

4)或 B →α A β 是文法 G 的一个产生式,且 β →ε，则将 Follow(B)加入到 Follow(A)中

非终结符	Follow
Package	{#}
PackageName	Package -> package PackageName ; Follow(PackageName) = Fisrt(;) = { ; }
PackageNameFollow	PackageName -> identifier PackageNameFollow

	Follow(PackageNameFollow) = Follow(PackageNameFollow) ∪Follow(PackageName) = { ; } PackageNameFollow -> . identifier PackageNameFollow Follow(PackageNameFollow) = Follow(PackageNameFollow) ∪Follow(PackageNameFollow) = { ; }

构造预测分析表

$G = (V_N, V_T, P, A)$

V_N = { A,B,C,D,E}

V_T = { a,b,c,d,f,g}

P={

1: A -> B C c

2: A -> g D B

3: B -> b C D E

4: B -> ε

5: C -> D a B

6: C -> c a

7: D -> d D

8: D -> ε

9: E ->g A f

10: E ->c

}

	a	b	c	d	f	g	#
A	A -> B C c	A -> B C c	A -> B C c	A -> B C c		A -> g D B	
B	B -> ε	B -> b C D E	B -> ε	B -> ε	B -> ε	B -> ε	B -> ε

C	C -> D a B		C -> c a	C -> D a B			
D	D -> ε	D -> ε	D -> ε	D -> d D	D -> ε	D -> ε	D -> ε
E			E ->c			E ->g A f	

思路：

1) 若 A → α 是一个产生式, a∈ First(α),那么当 A 是下推栈栈顶符号且将读入 a 时, 选择 α 取代 A 进行后续匹配成功的希望最大。故 M[A , a]元素为 A → α 。

2）若 A → α ，而 α = ε ，或 α →+ ε ；当 A 是栈顶符号且将读入 a 时，若 a∈ Follow(A),则栈顶的 A 应被 ε 匹配, 此时读头不前进, 让 A 的随符与读头下的符号进行匹配, 这样进行匹配，成功的可能性最大。故 M[A,a]元素为 A → α 。

算法：

1）若 A →α 是一个产生式，对于每一个元素 a∈ First(α)，应在 M[A,a]中填入 A → α；

2）若 A →α 是一个产生式，而 ε ∈ First(α)，对于每一个 a∈ Follow(A)，应在 M[A,a]中填入 A →α 。

3）把所有无定义的 M[A,a]都填上出错标志。

注意：

1）用此算法可以为任意文法 G 构造其分析表 M。

2) 若是二义文法或没有消除左递归和提取左因子的文法, 构造出的 M 包含有重定义项。即，它们的 M[A,a]中填有一个以上的产生式。

例 1：

G = (V_N ,V_T ,P, S)

V_N = { S,T}

V_T = { (,), ε }

P={

1: S -> T

2: T -> T (T)

3: T -> ε

}

非终结符	First
T	{ (, ε }
S	{ (, ε }

非终结符	Follow
T	{#,) , (}
S	{ # }

	(	)	#
T	T -> T (T) T -> ε	T -> ε	T -> ε
S	S -> T		S -> T

1. 对于 S -> T, Fisrt(T) = { (, ε } ,

   对于(, M[s,(] = S -> T,

   而 $\varepsilon \in$ First(T) , Follow(S) = {#} , M[S,#] = S -> T

2. T -> T (T), Fisrt(T (T)) = { Fisrt(T)- ε } $\cup$ { (} = { (}

   对于(, M[T, ( ] = T -> T (T)

3. T -> ε, Follow(T) = {#,) , (}

例 2:

G = (V_N ,V_T ,P, A)

V_N = { A,B,C,D,E}

V_T = { a,b,c,d,f,g}

P={

1: A -> B C c

2: A -> g D B

3: B -> b C D E

4: B -> ε

5: C -> D a B

6: C -> c a

7: D -> d D

8: D -> ε

9: E ->g A f

10: E ->c

}

非终结符	First
A	{a,b,c,d,g}
B	{b, ε }
C	{a,c,d}
D	{d, ε }
E	{g , c }

非终结符	Follow
A	{ f , # }
B	{a,c,d,f,g,#}
C	{c,d,g}
D	{a,b,c,f,g,#}
E	{a,c,d,f,g,#}

	a	b	c	d	f	g	#
A	A -> B C c	A -> B C c	A -> B C c	A -> B C c		A -> g D B	
B	B -> ε	B -> b C D E	B -> ε	B -> ε	B -> ε	B -> ε	B -> ε
C	C -> D a B		C -> c a	C -> D a B			
D	D -> ε	D -> ε	D -> ε	D -> d D	D -> ε	D -> ε	D -> ε
E			E ->c			E ->g A f	

1: A -> B C c

Fisrt(B C c) = { Fisrt(B) - ε } $\cup$Fisrt(C) = {b} $\cup$ {a,c,d} = {a,b,c,d}

2: A -> g D B

Fisrt(g D B)={g}

3: B -> b C D E

Fisrt(b C D E)={b}

4: B -> ε

Follow(B) ={a,c,d,f,g,#}

5: C -> D a B

Fisrt(D a B) = { Fisrt(D) - ε } ∪Fisrt(a) = {a,d}

6: C -> c a

Fisrt(c a) = {c}

7: D -> d D

Fisrt(d D) = {d}

8: D -> ε

Follow(D) ={a,b,c,f,g,#}

9: E ->g A f

Fisrt(g A f) = {g}

10: E ->c

Fisrt(c) = {c}

例 3:

G = (V_N ,V_T ,P, S)

V_N = { Package, PackageName, PackageNameFollow}

V_T = { package , identifier , . , ; , ε}

P={

0: Package -> package PackageName ;

1: Package -> ε

2: PackageName -> identifier PackageNameFollow

3: PackageNameFollow -> . identifier PackageNameFollow

4: PackageNameFollow -> ε

}

非终结符	First

Package	{ package , ε }
PackageName	{ identifier}
PackageNameFollow	{ . , ε}

非终结符	Follow
Package	{#}
PackageName	{ ; }
PackageNameFollow	{ ; }

	package	identifier	.	;	#
Package	0				1
PackageName		2			
PackageNameFollow			3	4	

1. Package -> package PackageName ; （0）

   Fisrt(package PackageName ;) = { package }

2. Package -> ε（1）

   Follow(Package) = {#}

3. PackageName -> identifier PackageNameFollow （2）

   Fisrt(identifier PackageNameFollow) = { identifier }

4. PackageNameFollow -> . identifier PackageNameFollow （3）

   Fisrt(. identifier PackageNameFollow) = {.}

5. PackageNameFollow -> ε（4）

   Follow(PackageNameFollow) = {;}

LL(1)文法定义

若文法 G 的预测分析表 M 中不含有多重定义项，则称 G 为 LL (1)文法。

说明：

1)LL(1)文法是无二义的，二义文法一定不是 LL(1)文法。

2) LL 的含义是从左到右扫描输入串，采用最左推导分析句子。

3)数字 1 表示分析句子时需向前看一个输入符号。

4)有 LL(1)就有 LL(k)，LL(k)向前查看 k 个输入符号，选择候选式更加准确，但预测分析表 M 的尺寸会爆炸性增长。

LL(1)进行语法分析

例 1：

	a	b	c	d	f	g	#
A	A -> B C c	A -> B C c	A -> B C c	A -> B C c		A -> g D B	
B	B -> ε	B -> b C D E	B -> ε	B -> ε	B -> ε	B -> ε	B -> ε
C	C -> D a B		C -> c a	C -> D a B			
D	D -> ε	D -> ε	D -> ε	D -> d D	D -> ε	D -> ε	D -> ε
E			E ->c			E ->g A f	

识别 bcaccac#

动作	栈	识别串
初始化	#A	bcaccac#
A -> B C c 推导	#cCB	bcaccac#
B -> b C D E 推导	#cCEDCb	bcaccac#
b 匹配	#cCEDC	caccac#
C -> c a 推导	#cCEDac	caccac#
c 匹配	#cCEDa	accac#
a 匹配	#cCED	ccac#
D -> ε 推导	#cCE	ccac#
E ->c 推导	#cCc	ccac#
c 匹配	#cC	cac#
C -> c a 推导	#cac	cac#

c 匹配	#ca	ac#
a 匹配	#c	c#
c 匹配	#	#
识别成功		

例 2：

G = (V_N ,V_T ,P, S)

V_N = { Package, PackageName, PackageNameFollow}

V_T = { package , identifier , . , ; , ε}

P={

1: Package -> package PackageName ;

2: Package -> ε

3: PackageName -> identifier PackageNameFollow

4: PackageNameFollow -> . identifier PackageNameFollow

5: PackageNameFollow -> ε

}

	package	identifier	.	;	#
Package	1				2
PackageName		3			
PackageNameFollow			4	5	

识别 package com.msb.compiler;#

动作	栈	识别串
初始化	# Package	package com.msb.compiler;#
Package -> package PackageName ; 推导	#; PackageName package	package com.msb.compiler;#
package 匹配	#; PackageName	com.msb.compiler;#

PackageName->identifier PackageNameFollow	#; PackageNameFollow identifier	com.msb.compiler;#
identifier 匹配	#; PackageNameFollow	.msb.compiler;#
PackageNameFollow -> . identifier PackageNameFollow 推导	#; PackageNameFollow identifier .	.msb.compiler;#
. 匹配	#; PackageNameFollow identifier	msb.compiler;#
identifier 匹配	#; PackageNameFollow	.compiler;#
PackageNameFollow -> . identifier PackageNameFollow 推导	#; PackageNameFollow identifier .	.compiler;#
.匹配	#; PackageNameFollow identifier	compiler;#
identifier 匹配	#; PackageNameFollow	;#
PackageNameFollow -> ε 推导	#;	;#
;匹配	#	#
#匹配		
识别成功		

iii. 递归下降分析

定义：

若一个文法 G，不含有左递归，而且每个非终结符的所有候选式的首符集都是两两不相交的，那么就能为 G 中每个非终结符编写一个相应的递归过程。把该文法中所有这样的递归过程组合起来就构成一个不带回溯的自上而下分析程序——递归下降分析程序。

实现思想：

为文法中每个非终结符编写一个递归过程，每个过程的功能是识别由该非终结符推出的串，当某非终结符的产生式有多个候选式时，按 LL(1)形式唯一确定选择哪个候选式进行推导，若遇到某候选式为 ε，认为其自动匹配。把这些递归过程组合起来就构成了文法的递归下降分析程序。

具体实现：

先将文法消除左递归、提取公共左因子，使之成为 LL(1)文法，后将每个非终结符对应一个递归过程，过程体是按照相应产生式的右部符号串顺序编写。每匹配一个终结符，则再读入下一个符号，对产生式右部的每个非终结符，则调用相应的过程。

递归下降分析法的缺点

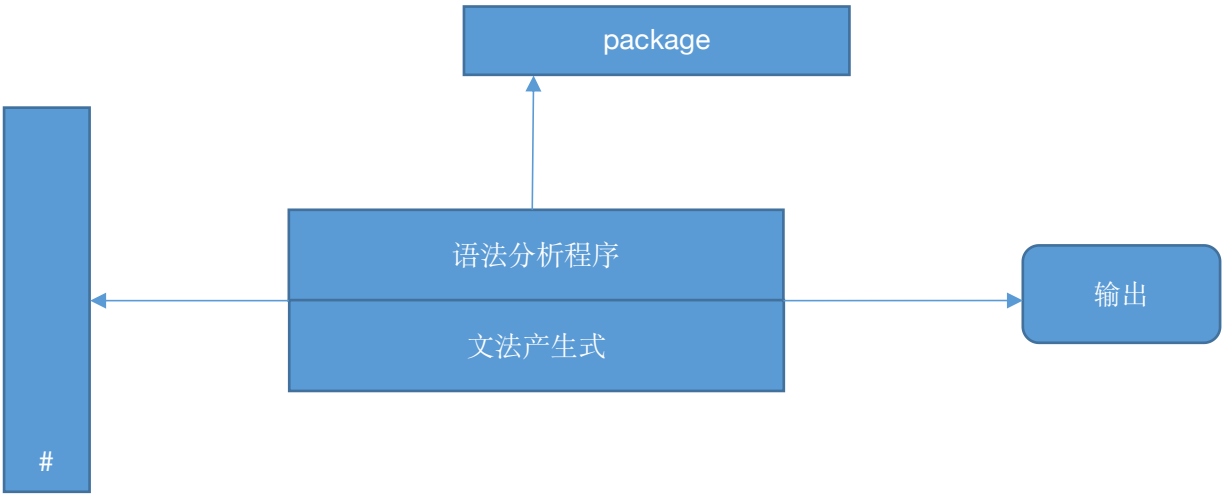
1） 对文法的要求高，必须满足 LL(1)文法。

2） 只要文法有调整，就需要对程序进行改写，扩展性非常差。

3） 高深度的递归调用会影响语法分析的效率，速度慢，占空间多。

c) 自下而上语法分析

从输入串开始，使用不同产生式进行归约，直到文法开始符号。



说明

1）初态时栈内仅有栈底符“ # ”，读头指在最左边的单词符号上。

2）语法分析程序执行的动作：

a）移进 读入一个单词并压入栈内，读头后移；

b）归约 检查下推栈栈顶若干个符号能否进行归约，若能，就以产生式左部替代该符号串，同时输出产生式编号；

c）识别成功 下推栈内只剩下“ # ”和文法开始符号，读头也指向输入串的结束符；

d）识别失败

举例，使用下面的文法：

G = (VN ,VT ,P, S)

VN = { Package, PackageName, PackageNameFollow}

VT = { package , identifier , . , ; , ε}

P={

1: Package -> package PackageName ;

2: Package -> ε

3: PackageName -> identifier PackageNameFollow

4: PackageNameFollow -> . identifier PackageNameFollow

5: PackageNameFollow -> ε

}

识别 package com.msb.compiler;过程。

编号	动作	栈	识别串
1	初始化	#	package com.msb.compiler;#
2	移进 package	# package	com.msb.compiler;#
3	移进 com	# package com	.msb.compiler;#
4	移进.	# package com .	msb.compiler;#
5	移进 msb	# package com . msb	.compiler;#
6	移进.	# package com . msb .	compiler;#
7	移进 compiler	# package com . msb . compiler	;#
8	使用 5 产生式规约	# package com . msb . compiler	;#

		PackageNameFollow	
9	使用 4 产生式规约	# package com . msb PackageNameFollow	;#
10	使用 4 产生式规约	# package com PackageNameFollow	;#
11	使用 3 产生式规约	# package PackageName	;#
12	移进	# package PackageName ;	#
13	使用 1 产生式规约	# Package	#
14	识别成功		

初看，上面的分析过程非常简单，但是仔细分析，其实并不简单。

对于过程 9，是进行移进操作，将分号（;）入栈，还是进行规约操作。

如果进行规约操作，那我们是使用 4 号产生式进行规约，还是使用 3 号产生式进行规约？这种不确定性的分析方法，是不能作为语法分析的工具。

确定的自上而下分析器通常分为两大类：优先分析器和 LR 分析器。

i. LR 分析器

简介：

LR 中的 L 是指自左至右扫描，R 最右推导的逆过程（也就是最左规约）。一般情况下，大多数用上下文无关文法描述的程序设计语言均可用 LR 分析器予以识别。JAVA 语言就可以使用 LR 分析法，进行识别，生成中间代码，最终生成 Class 文件。

LR 分析法的优缺点：

优点

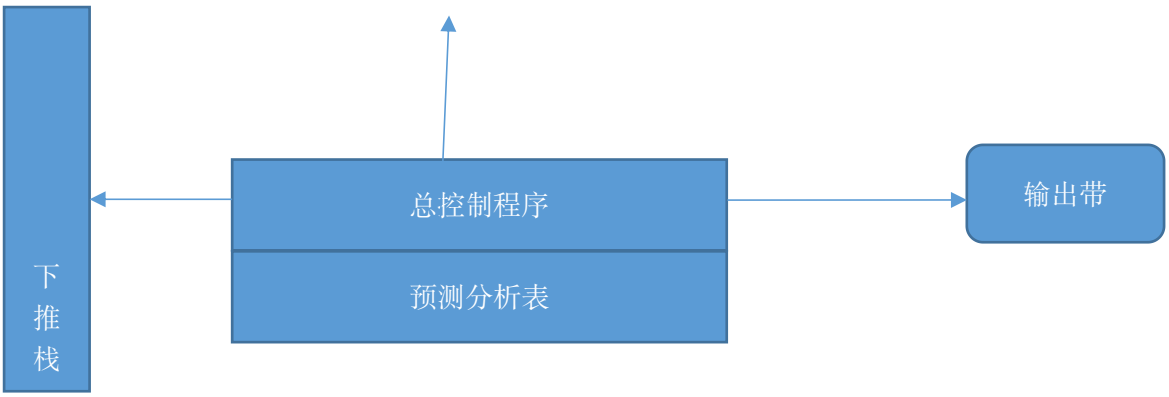
•适应文法范围广，分析效率高；

•在自左向右扫描输入串时，就能发现其中的语法错误，并能准确指出出错位置。

缺点

•若用手工构造分析程序，工作量太大，且容易出错，所以必须使用自动产生这种分析程序的产生器。

LR 分析器的实现模型：



一个 LR 分析器实际上是带有下推栈的确定的有限状态自动机 FA，可将一个“历史”与这个“历史”下的展望信息综合为抽象的一个状态。

下推栈：

下推栈用于存放，分析输入串过程中的所有由“历史”及相应“展望”信息形成的抽象状态。

由于每个状态都概括了从分析开始到归约阶段的全部“历史”和“展望”信息，所以 LR 分析器的每步动作可由栈顶状态和读头下符号唯一确定。

Sm	C
S3	A
……	
S2	b
S1	a
S0	#
状态栏	符号栏

为了便于了解栈顶状态对分析过程的作用，将栈分为两栏：状态栏和符号栏。符号栈仅记录迄今移进－归约所得到的文法符号，由于它们已经被概括在“状态”里了，所以对语法分析不起作用。

初始时符号栈放“＃”（栈底符），状态栈放 S0（初态）。

预测分析表：

	Action				Goto					
	a1	a2	……	an	a1	……	an	A1	an	An
S0										
S1										
S2										
S3										
……										
SN										

表格说明：

1.行 Si 表示状态， ai 表示终结符，Ai 表示非终结符。

2.预测分析表分为 Action 表和 Goto 表。

ACTION 表

ACTION[S_i,a]表示在当前状态 S_i 下，面临读头下的终结符为 a 所应采取的动作。注：该动作有四种可能：移进、归约、出错、接受。

移进：栈顶的状态是 S_i，读头下的元素 ai，查询 Goto 表，S`=GOTO($S_i$ ,ai )，将状态 S`连同读头下符号 ai 一起压栈，栈顶成为(S`,ai)，读头前进一格；

归约：指用某产生式 A →β 进行归约。若 β 的长度为 γ，则弹出栈顶的 γ 个元素，使得栈顶的状态变成 S_j，查询 Goto 表， S`=GOTO($S_j$，A)，将状态 S`连同非终结符 A 一起压栈，栈顶变成(S`,A)，读头不动。

接受：分析成功，退出总控程序。

报错：输入串出错，调用相应出错程序。

转向表 GOTO 表

GOTO[S,X]：若 $X \in V_T$，表示在当前状态下，读入 X 应转向什么状态；若 $X \in V_N$ ，表示当前栈顶句柄归约成 X 后，应转向什么状态。

注：对终结符的移进动作和转向动作可以合并在一起填在动作表中，这样，转向表可以只保留非终结符转向部分。

总控制程序

总控程序的动作根据当前栈顶状态 S_i 和读头下符号 ai 查表决定下一步动作，不管哪一类分析程序，总控程序的动作都一样。

总程序本身很简单，按动作表中填的内容具体实施而已。

LR 分析器的基本思想

在规范归约过程中，一方面记住已移进和归约出的整个符号串，另一方面根据所用的产生式推测未来可能碰到的输入符号。

LR 分析法寻找可归约句柄的依据

- 历史：
  - 记录在下推栈内的符号串移进、归约的历史情况
- 展望：
  - 预测句柄之后可能出现的信息；
- 现实：
  - 读头下的符号。

LR 分析法举例

对于下面的文法：

G = (VN ,VT ,P, S)

VN = { S, E, A,B}

VT = { a, b, c, d}

P={

0: S ->E

1: E -> a A

2: E -> b B

3: A -> c A

4: A ->d

5: B-> c B

6: B->d

}

预测分析表：

	Action					Goto			
	a	b	c	d	#	S	E	A	B
I_0	S1	S2					3		
I_1			S5	S6				4	
I_2			S8	S9					7
I_3					acc				
I_4	r1	r1	r1	r1	r1				
I_5			S5	S6				10	
I_6	r4	r4	r4	r4	r4				
I_7	r2	r2	r2	r2	r2				
I_8			S8	S9					11
I_9	r6	r6	r6	r6	r6				
I_{10}	r3	r3	r3	r3	r3				
I_{11}	r5	r5	r5	r5	r5				

说明：
S_j：移进操作，将识别串当前符号 a，压入下推栈符号栏，并将状态 I_j 压入下推栈状态栏，栈顶变成（I_j, a）；

r_j：规约操作，按照 j 号产生式进行规约；
acc：识别成功
空：出错标志

识别分析过程 a c d #过程

状态栏	符号栏	识别串	动作
I_0	#	acd#	初始化
I_0, I_1	#,a	cd#	ACTION（I_0，a）= S1 移进操作 I_1 入状态栏 a 入符号栏 读头后移
I_0, I_1, I_5	#,a,c	d#	ACTION（I_1，c）=S5 移进操作 I_5 入状态栏 c 入符号栏 读头后移
I_0, I_1, I_5, I_6	#,a,c,d	#	ACTION（I_5，d）=S6 移进操作 I_6 入状态栏 d 入符号栏 读头后移
I_0, I_1, I_5,, I_{10}	#,a,c,A	#	ACTION（I_6，#）= r4 4 号产生式：A ->d 规约 状态栏 I_6 出栈， 符号栏 d 出栈 栈顶式 I_5，产生式右部为 A 查表 Goto（I_5，A）= 10 I_{10} 入状态栏 A 入符号栏 读头保持不变

I_0, I_1, I_4	#,a,A	#	ACTION（S10，#）=r3 3 号产生式：A -> c A 规约，产生式右部长度为 2 状态栏 I_5, I_{10} 出栈， 符号栏 c,A 出栈 栈顶式 I_1，产生式右部为 A 查表 Goto（I_1，A）=4 I_4 入状态栏 A 入符号栏 读头不变
I_0, I_3	#E	#	ACTON（S4，#）= r1 1 号产生式：E -> a A 规约：产生式右部长度为 2 状态栏 I_1, I_4 出栈， 符号栏啊 a,A 出栈 栈顶式 I_0，产生式右部为 E 查表 Goto（I_0，E）= I_3 I_3 符状态栈 E 入符号栈
I_0, I_3	#E	#	ACTION（I_3，#）= ACC 识别成功

	Action					Goto			
	a	b	c	d	#	S	E	A	B
I_0	S1	S2					3		
I_1			S5	S6				4	
I_2			S8	S9					7
I_3					acc				
I_4	r1	r1	r1	r1	r1				
I_5			S5	S6				10	
I_6	r4	r4	r4	r4	r4				

I_7	r2	r2	r2	r2	r2				
I_8			S8	S9					11
I_9	r6	r6	r6	r6	r6				
I_{10}	r3	r3	r3	r3	r3				
I_{11}	r5	r5	r5	r5	r5				

识别分析过程 a b c #过程

状态栏	符号栏	识别串	动作
I_0	#	abc#	初始化
I_0, I_1	#,a	bc#	ACTION（I_0,a）= S1 移进操作 I_1 入状态栏 a 入符号栏 读头后移
I_0, I_1	#,a	bc#	ACTION（I_1, b）=空 出错

ii. LR(0)预测分析表构建

活前缀

前缀：字的前缀是指该字的任意首部。
例如：字 ABC 的前缀有 ε，A，AB，ABC

活前缀：指规范句型的一个前缀，这种前缀不含句柄之后的任何符号。

根据下面的文法识别输入串 abbcde。
1)S→aAcBe
2)A→b
3)A→Ab
4)B→d

为每条产生式的尾部加上用[I]表示的产生式序号
1)S→aAcBe[1]
2)A→b[2]
3)A→Ab[3]
4)B→d[4]
用最右推导方式来识别,推导时把序号也带上。
• S

→aAcBe[1]
→aAcd[4]e[1]
→aAb[3]cd[4]e[1]
→ab[2]b[3]cd[4]e[1]

• 若用最左归约的方式进行识别，则完全是上面的逆过程。
→ab[2]b[3]cd[4]e[1]
→aAb[3]cd[4]e[1]
→aAcd[4]e[1]
→aAcBe[1]
• S

• 规范句型 abbcde 的活前缀有： ε ， a， ab
• 规范句型 aAbcde 的活前缀有： ε ， a， aA， aAb
• 规范句型 aAcde 的活前缀有： ε ， a，aA，aAc， aAcd
• 规范句型 aAcBe 的活前缀有： ε ， a， aA， aAc， aAcB，aAcBe

活前缀有两种类型：
– 1）归态活前缀
活前缀的尾部正好是句柄之尾，这时可以进行归约。归约之后又成为另一句型的活前缀。
– 2）非归态活前缀
句柄尚未形成，需要继续移进若干符号之后才能形成句柄。

LR(0)项目

在文法的每个产生式右部添加一个圆点，就成为 G 的一个 LR(0)项目（简称项目）。
– 特殊说明：
1. 圆点在产生式中的位置不同则是不同项目；
2. 可以把圆点理解为栈内外的分界线，圆点左边的符号在栈内，圆点右边的符号在栈外；
3. 产生式右部符号串的长度为 n，则可以分解为 n + 1 个项目；
4. 产生式 A→ ε 只有一个项目 A→• 。

举例说明：
1) S→aAcBe

对于 S→aAcBe 产生式，一共可以生成 6 个项目
1. S→•aAcBe
2. S→a•AcBe
3. S→aA•cBe
4. S→aAc•Be
5. S→aAcB•e
6. S→aAcBe•

对于 A→b，一共可以产生 2 个项目
A→•b
A→b•

LR(0)项目集规范族构造

1. 拓广文法

   如果文法 G 的开始符号 S 出现在产生式的右部, 则新增加一条的产生式 S`→S, 并将文法的开始符号 S 修改为 S`。这样使得文法的开始符号，不出现在任何产生式右部，从而保证有唯一的接受项目；

2. 项目集闭包

   设 I 是拓广文法 G 的一个项目集，项目集 I 的闭包记做 CLOSURE(I)；

   CLOSURE(I)的，按照下面的规则进行构造：

   A. I 的任何项目都属于 CLOSURE(I);

   B.若项目 A→α•Bβ 属于 CLOSURE(I)，B 是非终结符，那么，对于任何关于 B 的产生式 B→γ，项目 B→•γ 也属于 CLOSURE(I)；

   C. 重复执行步骤 B，直到 CLOSURE(I)不再扩大为止；

3. 状态转化函数 GO 函数

   GO(CLOSURE(I),X) = CLOSURE(J),其中 I,J 都是项目集，X ∈(V_N ∪ V_T)

   其中项目集 I 中有形如 A→α•Xβ 的项目，

   并且项目集 J 中有形如 A→αX•β 的项目。

   说明：

   对于上面的转换关系，是由于项目集 I 中有 A→α•Xβ 项目，项目集 J 中有 A→αX•β 项目，表示识别活前缀又移进一个符号 X。

4. 算法过程

```
constructProjectGroup(){
    C={CLOSURE(S` →•S)}/*初态项目集闭包*/
    DO{
        FOR (对 C 中每个项目集 I 和 G`中每个文法符号 X){
            IF (GO(I,X)非空且不属于 C) {
                把 GO(I,X)加入 C 中
            }
        }
    }
    WHILE (C 仍然在扩大)
}
```

说明:

1)此算法是迭代算法，置了 C 的初态(初态仅包含第一个项目集)后，每经过一次 FOR

语句，就扩大一次 C 中的项目集数，直到项目集数不再增加为止；

2)此算法是从 I_0 开始，按该项目集内的项目顺序依次求出所有后继项目集。由这样一层一层向下生成的所有项目集的方法避免了项目集的遗漏。

3)若在项目集中存在 A→• ε 项目，不应再做 GO 函数转向其他项目集，因为 A→ ε •和 A→• ε 是同一项目， 都是 A→•项目，它本身是归约项目，不存在后继项目。

4)由这个项目集规范族 C 中各个状态及状态转换函数 GO ，可构造一张识别活前缀的 DFA 图。

举例：

构造下面的文法的项目集：

G = (V_N ,V_T ,P, S)

V_N = { S, E, A,B}

V_T = { a, b, c, d}

P={

0: S ->E

1: E -> a A

2: E -> b B

3: A -> c A

4: A ->d

5: B-> c B

6: B->d

}

LR(0)预测分析表构造

1) 若项目（A→ α •a β ）∈ I_k，且 GO(I_k,a) = I_j，a 为终结符，则置 ACTION[I_k,a]=Sj，即移进 a，并转向 I_j状态；
2) 若项目（A → α • ）∈ I_k，则对任何终结符 a(包括语句结束符#),置 ACTION[I_k,a]=rj; 其中，j 为产生式 A→ α 的编号，即根据 j 号产生式进行归约；
3) 若项目（S`→ S• ）属于 I_k，则置 ACTION[I_k , #] = accept，简写为 acc；
4) 若 GO(I_k，A)= Ij，A 是非终结符，则置 GOTO[I_k,A] = j；
5) 分析表中凡不能用步骤 1 至 4 填入信息的空白项，均置上“出错标志”。

	Action					Goto			
	a	b	c	d	#	S	E	A	B
I_0	S1	S2					3		
I_1			S5	S6				4	
I_2			S8	S9					7
I_3					acc				
I_4	r1	r1	r1	r1	r1				
I_5			S5	S6				10	
I_6	r4	r4	r4	r4	r4				
I_7	r2	r2	r2	r2	r2				
I_8			S8	S9					11
I_9	r6	r6	r6	r6	r6				
I_{10}	r3	r3	r3	r3	r3				
I_{11}	r5	r5	r5	r5	r5				

	Action					Goto			
	a	b	c	d	#	S	E	A	B
I_0	S1	S2					3		
I_1			S5	S6				4	
I_2			S8	S9					7
I_3					acc				
I_4	r1	r1	r1	r1	r1				
I_5			S5	S6				10	
I_6	r4	r4	r4	r4	r4				
I_7	r2	r2	r2	r2	r2				
I_8			S8	S9					11
I_9	r6	r6	r6	r6	r6				
I_{10}	r3	r3	r3	r3	r3				
I_{11}	r5	r5	r5	r5	r5				

LR(0)文法

若文法 G 按上面算法构造出来的预测分析表，不包含多重定义项，则该文法 G 是 LR(0)文法。

说明：

1. LR(0)文法的每个项目集中不包含有任何冲突项目；

   设文法 G 的 LR(0)项目集规范族中含有如下一个 项目集(状态)I:

   {

   $X \rightarrow \delta \bullet b \beta$, /*移进项目*/

   $A \rightarrow \alpha \bullet$, /*归约项目*/

   $B \rightarrow \beta \bullet$. /*归约项目*/

}

这三个项目告诉我们应做的动作各不相同，出现了移进-归约冲突 和 归约-归约冲突，因此这个文法 G 一定不是 LR(0)文法。

2. LR(0)文法的能力很弱，甚至连表达式文法也不属于 LR(0)文法，所以没有实用价值，但可以利用它的构造算法来构造其他 LR 分析表。

iii. SLR 分析法

SLR（Simple LR）是 LR(0)的一种改进，它在归约时除了考虑历史情况之外还考虑了一点现实（读头下的符号）。

消除冲突

一般而言，对于任何形如 I={ X→ δ •bβ , A→ α •, B→ β •} 的 LR(0)项目集，若 Follow(A) ∩ Follow(B) = Φ 且 b ∉ Follow(A), b ∉ Follow(B)，则可以根据当前读头下符号 a 来消除冲突。即在构造 LR 分析表的算法中做一些改变：

1)若当前输入符 a = b，做移进；
2)若当前输入符 a ∈ Follow(A)，按 A→ α 产生式归约；
3)若当前输入符 a ∈ Follow(B)，按 B→ β 产生式归约；
4)其他，报错。

说明：

当我们选择 A→ α 这个产生式进行规约的时候，而如果读头下的符号 a∉ Follow(A)，后面一定式出错标志。

SLR 预测分析表构造

设 C={I_0, I_1,... I_n},以各项目集 I_k(k=0，...,n)的 k 作为状态 序号，并以包含 S`→•S 的项目集作为初始状态，同时将产生式进行编号。然后按下列步骤填写 ACTION 表和 GOTO 表:

1)若项目 A→ α •a β ∈ I_k 状态且 GO(I_k,a) = I_j ， a 为终结符，则置 ACTION[I_k,a] = Sj，即:移进 a，并转向 I_j 状态；

2)若项目 A→ α •∈ I_k ，则对任何终结符 a ∈ Follow(A)，置 ACTION[k,a]=rj;其中 j 为产生式 A→ α 的编号，即根据 j 号产生式 A→ α 进行归约；

3)若项目 S`→S•∈ I_k，则置 ACTION[k,#]=acc;

4)若 GO(I_k， A)= I_j，A 是非终结符，则置 GOTO[k,A]=j;

5)分析表中凡不能用步骤 1 至 4 填入信息的空白项，均置上“出错标志”。

Follow(S) = [#]
Follow(E) = [#]
Follow(A) = [#]
Follow(B) = [#]

	Action					Goto			
	a	b	c	d	#	S	E	A	B
I_0	S1	S2					3		
I_1			S5	S6				4	
I_2			S8	S9					7
I_3					acc				
I_4					r1				
I_5			S5	S6				10	
I_6					r4				
I_7					r2				
I_8			S8	S9					11
I_9					r6				
I_{10}					r3				
I_{11}					r5				

说明：
1)若文法 G 按上面算法构造出来的分析表不包含多重定义项，则该文法 G 是 SLR 文法；
2)二义文法一定不是 SLR 文法；
3)SLR 分析法包含的展望信息是体现在利用了 Follow(A)信息，可以解决“归约-归约”冲突
4)SLR 分析法没有包含足够的展望信息，不能解决“移进-归约”冲突，需要改进。

SLR 举例

文法产生式：
$G = (V_N , V_T ,P, S)$
V_N = { E, S, L,R}
V_T = { =, *, i, #}
S = E
P={
0. E → S
1. S → L = R
2. S → R
3. L → * R
4. L → i
5. R → L
}

Follow(E) = [#]
Follow(S) = [#]
Follow(L) = [#, =]
Follow(R) = [#, =]

	Action	Goto

	=	*	i	#	E	S	L	R
I_0		S4	S3			2	5	1
I_1				r2				
I_2				acc				
I_3	r4			r4				
I_4		S4	S3				7	6
I_5	r5/S8			r5				
I_6	r3			r3				
I_7	r5			r5				
I_8		S4	S3				7	9
I_9				r1				

SLR 分析总结:

在构造 SLR 分析表的方法中，若项目集 I_k 中含有 $A\rightarrow\alpha\bullet$，那么在状态 I_k 时，只要面临 输入符号 a $\in$ Follow(A)，就确定采用 $A\rightarrow\alpha$ 产生式进行归约。

但是，在某种情况下，当状态 I_k 呈现于栈顶时，栈里的符号串所构成的活前缀 $\beta\ \alpha$ 未必允许把 α 归约为 A。因为可能没有一个规范句型含有前缀 β Aa。因此此时用 $A\rightarrow\alpha$ 产生式进行归约未必有效。

识别 i = i 过程

状态栏	符号栏	识别串	动作
I_0	#	i=i#	初始化
I_0, I_3	#,i	=i#	ACTION（I_0，i）= S3 移进操作 I_3 入状态栏 i 入符号栏 读头后移
I_0, I_5	#,L	=i#	ACTION（I_3，=）=r4 4 号产生式：L → i 规约：产生 式右部长度为 1 状态栏 I_3 出栈， 符号栏啊 i 出栈 栈顶式 I_0，产生式右部为 L 查表 Goto（I_0，L）= I_5 I_5 符状态栏

			L 入符号栏
I_0, I_1	#,R	=i#	ACTION（I_5，=）= r5/S8 有两种操作： r5 进行规约 5 号产生式：R → L 规约：产生式右部长度为 1 状态栏 I_5 出栈， 符号栏啊 L 出栈 栈顶式 I_0，产生式右部为 R 查表 Goto（I_0，R）= I_1 I_1 入状态栈 R 入符号栈
I_0, I_1	#,R	=i#	ACTION（I_1，=）= NULL，出错 原因： 上一步中， 下推栈里的符号栏所构成的活前缀#,L，不允许把 L 归约为 R，因为没有一个规范句型含有前缀#，R 所以 ACTION（I_5，=）= r5 是一个错误的选择。

从上面的例子，

I_5

{

4 : S → L•= R

15 : R → L•

}

结论:

由此看出，并非随符都出现在规范句型中。

对策:

给每个 LR(0)项目添加展望信息，即:添加句柄之后可能跟的终结符，因为这些终结符确

实是规范句型中跟在句柄之后的。

这就是 LR(1)的分析法。

可能引起的问题

LR(1)项目是对 LR(0)项目的分裂，若文法中终结符的数目为 n，则每个 LR(0)项目都可以分裂成 n 个 LR(1)项目。这可能会引起分析表的膨胀。

iv. LR(1)分析法

LR(1)项目

形如(A→α•β,a)的二元式称为 LR(1) 项目。其中， A→α β 是文法的一个产生式，a 是终结符， 称为搜索符。

说明：

LR(1)项目是对 LR(0)项目的分裂，若文法中终结符的数目为 n，则每个 LR(0)项目可以分裂成 n 个 LR(1)项目；

(A→α•β,a)的含义:预期当栈顶句柄 α β 形成后， 在读头下读到 a。此时 α 在栈内， β 还未入栈，即它展望了句柄后的一个符号。

LR(1)有效项目:

若存在规范推导 S`→δAω → δαβω ，其中δα称规范句型δαβω的活前缀(记作 γ)，a ∈ First(ω)，则 LR(1)项目(A→α•β,a)对于活前缀 γ 是有效的。如果 a∉First(ω)，即使 a∈ Follow(A)，项目(A→α•β,a) 也是无效的。

说明:

1)规范 LR 分析法仅考虑有效的 LR(1)项目。在 LR(1)项目中有效的项目并不多。

2)对于多数程序设计语言，向前展望一个符号就足以决定归约与否，所以只研究 LR(1)。

LR(1)预测分析表构造

LR(1)项目集的闭包 CLOSURE(I) – I

(1) I 的任何项目都属于 CLOSURE(I)；

(2) 若项目（ A→α•Bβ,a）属于 CLOSURE(I)， B →γ 是一个产生式，那么对于 FIRST(βa)中的每个终结符 b，如果（B→•γ,b）原来不在 CLOSURE(I)中，则把它加进去；

(3) 重复步骤（2）直到 CLOSURE(I)不再扩大为止。

举例：

G = (V_N , V_T ,P, S)

V_N = { E, S, L,R}

V_T = { =, *, i, #}

S = E

P={

0. E → S
1. S → L = R
2. S → R
3. L → * R
4. L → i
5. R → L

}

求 CLOSURE(I)

LR(0)项目

1 : E → ● S
2 : E → S ●

3 : S → ● L = R
4 : S → L ● = R
5 : S → L = ● R
6 : S → L = R ●

7 : S → ● R
8 : S → R ●

9 : L → ● * R
10 : L → * ● R
11 : L → * R ●

12 : L → ● i
13 : L → i ●

14 : R → ● L
15 : R → L ●

I = {1 : E → ● S [#]}，求 CLOSURE(I)

1 : E → ● S [#]	根据 1 : E → ● S [#]，找以 S 开头的产生式 3 : S → ● L = R 7 : S → ● R 求搜索符 First(β #) = First(#) ={#} 所以 CLOSURE(I)中添加 2 个 LR(1)的项目 3 : S → ● L = R[#] 7 : S → ● R [#]
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#]	根据 3 : S → ● L = R[#]，寻找以 L 开头的产生式 9 : L → ● * R 12 : L → ● i

	求搜索符 First(β #) = First(= R #) = {=} 所以 CLOSURE(I)中添加 2 个 LR(1)的项目 9 : L → ● * R [=] 12 : L → ● i [=]
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#] 9 : L → ● * R [=] 12 : L → ● i [=]	根据 7 : S → ● R [#]，寻找以 R 开头的产生式 14 : R → ● L 求搜索符 First(β #) = First(#) = {#} 所以 CLOSURE(I)中添加 1 个 LR(1)的项目 14 : R → ● L [#]
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#] 9 : L → ● * R [=] 12 : L → ● i [=] 14 : R → ● L [#]	9 : L → ● * R [=]，●后面是终结符 *，不能扩展
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#] 9 : L → ● * R [=] 12 : L → ● i [=] 14 : R → ● L [#]	12 : L → ● i [=]，●后面是终结符 I，不能扩展
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#] 9 : L → ● * R [=] 12 : L → ● i [=] 14 : R → ● L [#] 9 : L → ● * R [#] 12 : L → ● i [#]	根据 14 : R → ● L [#]，寻找以 L 开头的产生式 9 : L → ● * R 12 : L → ● i 求搜索符 First(β #) = First(#) = {#} 所以 CLOSURE(I)中添加 2 个 LR(1)的项目 9 : L → ● * R [#] 12 : L → ● i [#]
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#] 9 : L → ● * R [=] 12 : L → ● i [=] 14 : R → ● L [#] 9 : L → ● * R [#] 12 : L → ● i [#]	9 : L → ● * R [#]，●后面是终结符 *，不能扩展
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#] 9 : L → ● * R [=] 12 : L → ● i [=]	12 : L → ● i [=]，●后面是终结符 I，不能扩展

14 : R → ● L [#] 9 : L → ● * R [#] 12 : L → ● i [#]	
1 : E → ● S [#] 3 : S → ● L = R[#] 7 : S → ● R [#] 9 : L → ● * R [=,#] 12 : L → ● i [=,#] 14 : R → ● L [#]	合并搜索符

构造 LR(1)项目集规范族（同 LR0）

注意搜索符的处理。

举例：画出下面文法的状态转化图

$G = (V_N , V_T ,P, S)$

V_N = { E, S, L,R}

V_T = { =, *, i, #}

S = E

P={

0. E → S
1. S → L = R
2. S → R
3. L → * R
4. L → i
5. R → L

}

构造 LR(1)预测分析表

设 $C=\{I_0, I_1,... I_n\}$,以各项目集 I_k(k=0，...,n)的下标 k 为分析表中的状态，并以包含(S` → •S，#)的项目的状态为分析表初态。按下列步骤填写 ACTION 表和 GOTO 表:

1)若项目(A→ α •a β ,b)属于 I_k，且 $GO(I_k,a)= I_j$，a 为终结符，则置 ACTION[k,a]=Sj；即:移进 a，并转向 I_j 状态；

2)若项目(A→ α •,a) ∈ I_k，则置 ACTION[k,a]=rj ;其中 j 为产生式 A→ α 的编号，即根据 j 号产生式 A→ α 进行归约；

3)若项目(S` → S•,#)属于 I_k，则置 ACTION[k,#]=acc;

4)若 $GO(I_k$，A)= I_j,A 是非终结符，则置 GOTO[k,A]=j;

5)分析表中凡不能用步骤 1 至 4 填入信息的空白项，均置 上“出错标志”。

举例：

$G = (V_N , V_T ,P, S)$

V_N = { E, S, L,R}

V_T = { =, *, i, #}

S = E

P={

0. E → S
1. S → L = R
2. S → R
3. L → * R
4. L → i
5. R → L

}

填写 LR(1)的预测分析表:

	Action				Goto			
	=	*	i	#	E	S	L	R
I_0		S4	S3			2	5	1
I_1				r2				
I_2				acc				
I_3	r4			r4				
I_4							7	6
I_5	S8			r5				
I_6	r3			r3				
I_7	r5			r5				
I_8		S11	S10				12	9
I_9				r1				
I_{10}				r4				
I_{11}		S11	S10				12	13
I_{12}				r5				
I_{13}				r3				

	Action				Goto			
	=	*	i	#	E	S	L	R
I_0		S4	S3			2	5	1
I_1				r2				
I_2				acc				
I_3	r4			r4				
I_4		S4	S3				7	6
I_5	S8			r5				
I_6	r3			r3				
I_7	r5			r5				
I_8		S11	S10				12	9
I_9				r1				

I_{10}				r4				
I_{11}		S11	S10				12	13
I_{12}				r5				
I_{13}				r3				

使用预测表进行预测分析。

LR(1)文法

定义：

若文法 G 按构造 LR(1)分析表算法构造出来的分析表不包含多重定义项，则该文法 G 是 LR(1)文法

说明：

1. 每个 SLR 文法都是 LR(1)文法，反之不一定成立。
2. 一个文法 G 的 LR(1)分析表要比其 SLR 分析表，含有更多的状态。在严重的情况下，状态数可能成几倍增长。因此需要简化，简化的算法是 LALR 分析法。

v. LALR 分析法

6. 语义分析生成中间代码

中间代码

中间代码不是机器语言，中间代码的存在，是为了便于生成机器语言，以及便于进行代码优化。

中间代码存在的形式：

1. 逆波兰式
2. 树形表示法
3. 三元式
4. 四元式:最常用的形式(Operator,Operand1,Operand2,Result)

A. Operand1,Operand2,Result 可能是用户自定义的变量，也可能是编译时引进的变量。这里 Operator 可能是双目运算符，Operand1 和 Operand2 都有效；Operator 也可能是单目运算符，则只有一个运算量 Operand1 或者 Operand2 有效；
B. 四元式中变量采用符号表入口的地址，而不用变量的地址，因为语义分析不仅需要变量的地址，还需要从符号表查到的变量的属性、类型和地址等；
C. 四元式的优点是容易转换为目标代码和容易进行代码优化。

中间代码生成方法

语法制导翻译

在语法分析的基础上进行边分析边翻译。

1)语法制导翻译时会根据文法产生式右部符号串的含义，进行翻译，翻译的结果是生成相应中间代码；

2)语法制导翻译的依据是语义子程序；

3)具体做法：为每个产生式配置一个语义子程序，当语法分析进行归约或推导时，调用语义子程序，完成一部分翻译任务。

4)语法分析完成，翻译工作也告结束。

主要讨论 LR 语法制导翻译过程，特别是 LR(1) 语法制导翻译过程。

属性文法制导翻译

语义子程序

语意子程序的作用

用来描述一个产生式所对应的翻译工作，如：

1. 改变某些变量的值；
2. 查填各种符号表；
3. 发现并报告源程序错误；
4. 产生中间代码

语义子程序写法

通常写法：

X → α { 语义子程序 }

示例代码中的写法：

Package -> package PackageName ; @Package

语义值

为了描述语义动作，需要为每个文法符号赋予不同的语义值 ：类型、地址、代码值等；

语义栏

• 各个符号的语义值放在下推栈中，当产生式进行归约时，需对产生式右部符号的语义值进行综合，其结果作为左部符号的语义值保存到下推栈的语义栏中；
• 下推栈包含 3 部分：状态栏、符号栏和语义栏，并且他们是同步变化的。

a) 语义栈

当产生式进行归约时，需对产生式右部符号的语义值进行综合，其结果作为左部符号的语义值保存到语义栈中。

b) 简单算术表达式翻译

c) 赋值语句的翻译

d) 控制语句翻译

7. 中间代码优化

三、 JVM 理论

1. Class 文件构成（参加虚拟机规范 1.8 版本第四章 The class File Format）

通过编写一个 javap 程序，来加深对本章的理解。

2. Class 的加载，连接和初始化过程，重点讲初始化（参虚拟机规范 1.8 版本第五章 Loading, Linking, and Initializing）

通过讲解初始化的过程，来理解编译程序如何处理字段，静态字段，代码块和静态代码块。

3. JVM 的指令集，暂时只讲几个常用的（参加虚拟机规范 1.8 版本第六章 The Java Virtual Machine Instruction Set）

四、 中间代码生成目标 class 文件

把编译原理生成的中间代码，按照 JVM 的要求，生成相应的 Class 文件。