编译技术课程

试验指导书(2021 版)

不自己经历一次 TINY 语言的编译器构造过程，就等于没学这门课。经历过了，而且走通了，你就从现在的 1/100,飞跃到了 1/3000，甚至 1/10000。毕业后，你就拿着这个名片去华为，去阿里，去腾讯，去百度找工作，或者去北大，清华，上海交大，北航这些学校读研，绝对没问题。这个名片要比现在所谓的竞赛获奖的含金量大得多，别人相信得多。为什么？因为招聘官或者老师都是过来人，都知道编译技术是最难学的知识，没几个人真正学懂。现在你真正学懂了，别人刮目相看。

尽管有 LEX，FLEX，BISON 等开源工具。这些工具与现代工具相比，只有芝麻大。但对于一个人，在 2 个月时间内，要搞懂它又太难了，难于上青天。一个是量太大，另一个是看不懂。有这两个拦路虎，很难逾越。还不如自己来，而且远远地超越这些工具。

这就是老师要说的前言。

TINY 是一门高级程序语言。与真实的高级程序语言相比，例如 C 语言，TINY 当然是很简单的高级程序语言。TINY 有它的词法规则，和语法规则。TINY 既然是一门高级程序语言，就要有 TINY 语言的编译器。TINY 语言的编译器叫 TINY 编译器，它的源代码，用 C 语言写的，已经有了。因此，我们要用一个 C 语言开发工具，创建一个项目，给它取名为 TINY，把 TINY 编译器的源代码和头文件导入到该项目中，然后 build 它，生成了一个 tiny.exe 文件，这就是 TINY 编译器。既然是 exe 文件，就是一个应用程序。然后运行 tiny.exe 文件，以一个用 TINY 语言写的源程序文件（例如 sample.tny）作为其输入参数。于是，sample.tny 源程序文件就被 tiny.exe 编译成了一个目标文件 sample.tm。sample.tm 文件相当于一个 exe 文件。这个目标文件 sample.tm 要用一个虚拟机 tm.exe 来解释执行得到运行结果。类似的，Java 语言源程序编译成字节码文件，然后用 JAVA 虚拟机解释执行。

对课程试验，大家要有全局观念，四次试验的总目标：按照编译器构造方法说，自己实现编译器构造工具，以 TINY 语言为案例，得出它的编译器。然后用这个编译器，去编译一个用 TINY 语言写的程序，得到它的可执行文件（tm 文件）。然后用 tiny 虚拟机运行这个 tm 文件，得到正确的计算结果。如果再高攀一层的话，就是自己再写出 tiny 虚拟机的源代码。

第一项工作就是，自己写一个词法分析器构造工具，要做的事情自然是生成正则表达式的 NFA，再把 NFA 转化为 DFA。首先要拿定注意：是用 Java 语言，还是 C++语言，或者 C 语言来写。最好用 Java 语言，或者 C++语言来做，因为有更多的东东可用，写代码更加简单。接下来就是确定存储：是用文本文件来存储输入/输出的数据，还是用数据库来存储输入/输出的数据。如果

是用文件来存储，文件是字节流，离我们定义的数据结构，彼此之间有距离，彼此之间的转化代码，可以从 FLEX 那里摘取。要么就用数据库来存储，这样数据结构和存储就一致了，处理就要简单很多。甚至可利用开发工具，基于数据库的表来生成类的定义，这样数据结构的定义代码都不用手工来写了，只须定义数据库表了。 确定了存储之后，就是确定概念及其数据结构。其中包括词法分析的输入字符集。输出的词法单元由类型和值两个部分构成，例如【keyword,if】，【id,j】,【integer, 99】，【operator，+】等等。

对于正则表达式中的运算符，我们设定联接运算用&表示，或运算用|表示， 闭包运算用@表示，正闭包运算用#表示， 0 个或者 1 个运算用？表示。我们假定这些符号在要编译的语言（即目标语言）中不会出现。例如，在 TINY 语言中，这些字符就不会出现。如果出现，就要考虑转意符的问题。实验中，假设这几个符号在目标语言中不会出现，不考虑转意符。

正则表达式由头部和表达式两个部分组成。表达式中的元素可以是目标语言中的字符，也可以是一个正则表达式的名称。例如：

Keyword → **i** & **f** | **r** & **e** & **p** & **e** & **a** & **t** | **e** & **l** & **s** & **e**

d → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;

Integer → d#

注意：这里连接运算符是用&表识，正闭包运算是用#表示。第一个正则表达式中的每个元素都是目标语言中出现的字符，而第三个正则表达式中的元素 d 就是一个正则表达式的名称。

由此可知，正则表达式中的每个元素包括两个部分，一个是类型，一个是名称。于是正则表达式元素的数据结构应该为：

```
CREATE TABLE    element (
      element_id    SMALLINT,
      class_id    CHAR(1) CKECK VALUE IN('c', 'v')
      name      CHAR(1),
      remark      VARCHAR（32）    //注释说明
      PRIMARY KEY (element_id )
);
```

class_id 只有两个取值：
'c'： 表示是目标语言的一个输入字符(charactor)，name 的值就为该输入字符；
'v'： 表示是一个正则表达式名称(variant)，这个字符集的名称为 name 的值。

例如：

element 表

element_id	class_id	name	remark
0	c	'0'	目标语言字符

1	c	‘1’	目标语言字符
2	c	‘2’	目标语言字符
3	c	‘3’	目标语言字符
4	c	‘4’	目标语言字符
5	c	‘5’	目标语言字符
6	c	‘6’	目标语言字符
7	c	‘7’	目标语言字符
8	c	‘8’	目标语言字符
9	c	‘9’	目标语言字符
10	v	d	名称：数字
11	v	i	名称：整数常量
12	v	k	名称：保留字

以这个表中的数据为例，，element_id 为 0 到 9 的这 10 行，这是个元素是目标语言中出现的字符，而 element_id 为 10,11,12 的这三行，表达的元素 d, i, k 都是正则表达式的名称。这样来设计，就使得正则表达式的头部也是元素。于是，正则表达式由元素，运算符，以及→，这三者构成。基于上述元素表 element， 正则表达式：

10 → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9；

注意：这个正则表达式中的 10,0,1,2,3,4,5,6,7,8,9 都是 element_id。头部的 10 表示头部是 element 表中 element_id 为 10 的元素。9 表示是 element 表中 element_id 为 9 的元素。其它的 0 到 8 也是如此。

再看一个正则表达式例子：

11→ 10 #

其含义是：该正则表达式的头部为 element 表中 element_id 为 11 的元素，10 表示是 element 表中 element_id 为 0 的元素。#为正闭包运算。

于是，我们可以创建一个表来存储正则表达式：

```
CREATE TABLE    regular_expression     (
    head_element_id    SMALLINT，
    expression VARCHAR(64)，
    lexcial_class    CHAR(1) CHECK VALUE IN (‘i’, ‘n’, ‘r’)
    remark    VARCHAR(32)，          //注释说明
    PRIMARY KEY (head_element_id)
```

);

其中的 lexcial_class 字段，记录该表达式的词法类型，以便在语法分析中标识一个词素的类型。其取值，我们暂时只搞三种，以满足 TINY 语言的须要：‘i’表示是变量，identifier 的缩写， ‘n’表示是整数常量，numeric 的缩写， ‘r’表示是保留字，reserved 的缩写。对类型，以后可以进一步扩展。

按照上述方案来及定义正则表达式，带来了简单性。于是，正则表达式中出现的字符只有数字和运算符|，&，#，@，？，(，)，共计 17 个字符了。正则表达式中的概念也就只有 element_id，和运算符这两个了。对正则表达式，经上述处理后，仅只 17 个字符，和两个概念，变得很简单了，有利于对正则表达式做词法分析，和语法分析。还可加上一个空格字符，遇到它就把它忽略掉。

解决了正则表达式的数据结构后，接下来就是要定义 NFA/DFA 的数据结构。对于正则表达式中的五种运算，其中联接运算以及或运算，它俩都是两元运算，即输入为两个 NFA，输出为一个 NFA。

NFA/DFA 都有一个开始状态，和一个结束状态，结束状态还有类型的概念。类型的标识号就是 element 表中的元素的标识号 element_id。每个 NFA/DFA 都有一个边集，边集中包含至少一条边。NFA/DFA 和它包含的边的数据结构大致为：

```
CREATE TABLE    state_transition_graph     (
        graph_id    SMALLINT
        start_state    SMALLINT，
        end_state    SMALLIN，
        end_state_type CHAR(1) CHECK VALUE IN ( 'E', 'I');
        end_state_class CHAR(1) CHECK VALUE IN ( 'i', 'n', ‘r’);
        PRIMARY KEY (graph_id)
);

  CREATE TABLE    graph_edge     (
        graph_id    SMALLINT
        state    SMALLINT，
       driver_char_element_id    SMALLINT，
        next_state    SMALLINT，
        PRIMARY KEY (graph_id,state ,driver_char, next_state )
        FOREIGN KEY (graph_id) REFERENCES state_transition_graph(graph_id)
);
```

在 state_transition_graph 表中，end_state_type 字段的取值含义：
'E'：匹配状态，但是不包含(exclude)当前字符;
'I'：匹配状态，包含(include)当前字符；

end_state_class 字段的取值含义： 'i'表示变量， 'n'表示整数常量,‘r’表示保留字。这个在上表正则表达式中已讲。

我们说了，正则表达式也是一门语言,不过是一门非常简单的语言。既然是一门语言，它就五脏俱全，有词法分析和语法分析问题。我们首先要对正则表达式进行词法分析。然后进行语法分析，看是否是一个正则表达式。是的话，再看首先要执行哪个运算，再依次执行哪些运算。

对上面所述作总结。我们首先来看正则表达式这门语言中，有哪些字符？

根据我们上面描述的数据结构，正则表达式这门语言中的字符只有 17 个，分别是：

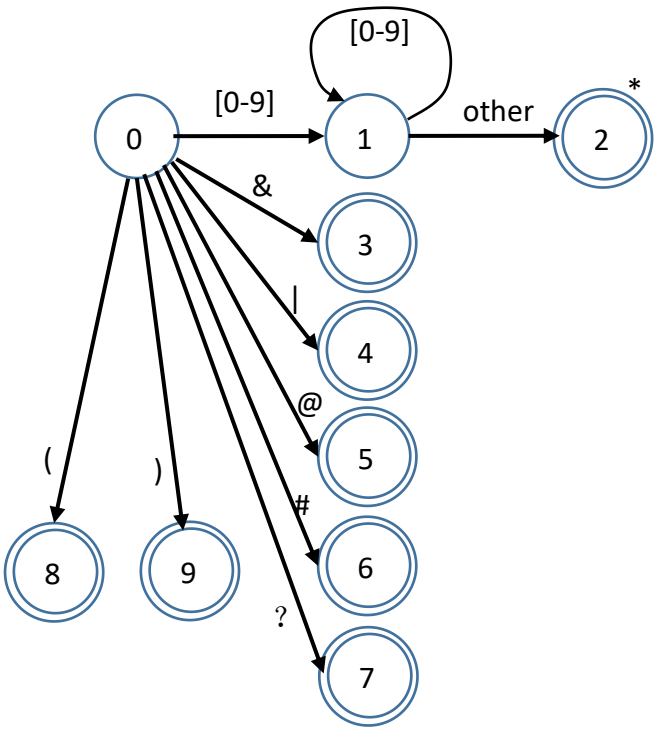
- 数字 0-9；
- 以及运算符：|， &， @， #，？，以及(，)；

再来看正则表达式这门语言的词法单元类型。它只有两种词法单元：element_id，和运算符。

正则表达式这门语言的词法分析中，词素构成的正则表达式是：

Lexeme → element_id 或者 | 或者 & 或者 @ 或者 # 或者(或者)

因此，对于正则表达式这门语言，其词法分析的 DFA 为:



对于给出的正则表达式，要做语法分析，确定首先要执行哪个运算，然后再依次执行哪些运算。每执行一次运算，要做的事情就是 NFA 的一次构造。一个正则表达式的语法分析过程，也就是该正则表达式的 NFA 构造过程。

对于这门正则表达式语言，用于其语法分析的上下文无关文法如下：

E→ E & E ｜E｜E｜E? ｜E@｜E#｜(E)｜id

这里的 id 是指 element_id，而且是指 element 表中 class_id 字段值为'c'的那些行，即目标语言中包含的字符。

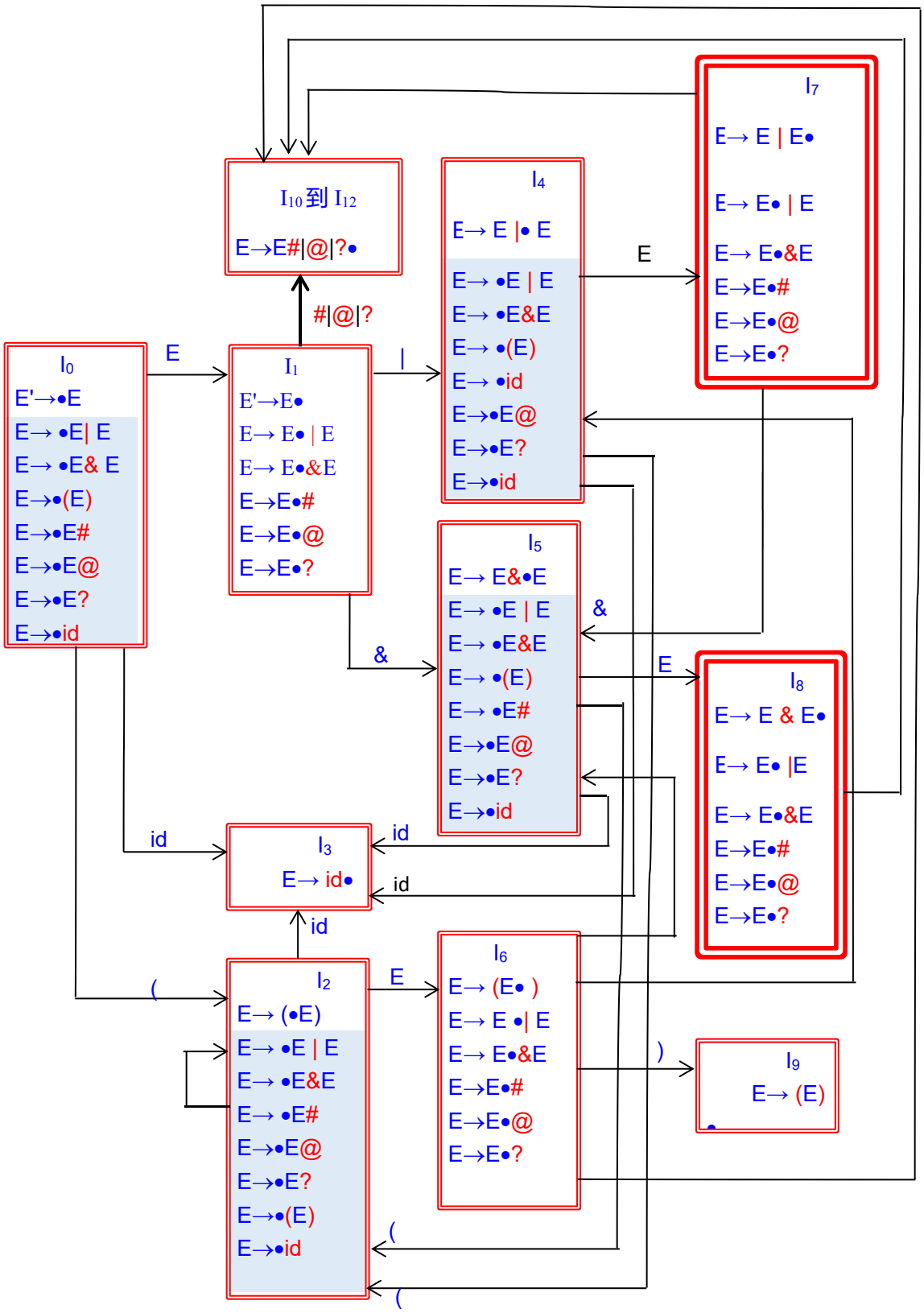
注意：这里，目标语言不是正则表达式这门语言。而是我们要考虑的语言，例如 TINY 语言。

上述正则表达式这门语言的上下文无关文法中，只含一个非终结符 E。对目标语言，例如 TINY 语言，其词法规则通常包含有多个正则表达式，例如整数常量的正则表达式，小数常数的正则表达式，保留字的正则表达式等等。正则表达式中的运算符有优先级，单元运算#，@，？优先于双元运算&，|。双元运算中，&优先于|。括号运算的优先级最高。

正则表达式这门语言的上下文无关文法，共有 7 个产生式：

产生式序号	产生式β	产生式体中符号的数量|β|
1	E→ E ｜ E	3
2	E→ E & E	3
3	E→ (E)	3
4	E→ E#	2
5	E→ E@	2
6	E→ E?	2
7	E→ id	1

上述文法的 DFA 为：


I0
E'→•E
E→ •E| E
E→ •E& E
E→•(E)
E→•E#
E→•E@
E→•E?
E→•id
E
I1
E'→E•
E→ E• | E
E→ E•&E
E→E•#
E→E•@
E→E•?
|
I4
E→ E |• E
E→ •E | E
E→ •E&E
E→ •(E)
E→ •id
E→•E@
E→•E?
E→•id
E
I7
E→ E | E•
E→ E• | E
E→ E•&E
E→E•#
E→E•@
E→E•?
#|@|?
I10到 I12
E→E#|@|?•
&
I5
E→ E&•E
E→ •E | E
E→ •E&E
E→ •(E)
E→ •E#
E→•E@
E→•E?
E→•id
&
E
I8
E→ E & E•
E→ E• |E
E→ E•&E
E→E•#
E→E•@
E→E•?
id
I3
E→ id•
id
id
id
(
I2
E→ (•E)
E→ •E | E
E→ •E&E
E→ •E#
E→•E@
E→•E?
E→•(E)
E→•id
E
I6
E→ (E•)
E→ E •| E
E→ E•&E
E→E•#
E→E•@
E→E•?
)
I9
E→ (E)
(
(

由上述 DFA 得出的 SLR(1)语法分析表如下：

状态	ACTION									GOTO
	id	|	&	(	)	$	#	@	?	E
0	s3			s2						1
1		s4	s5			acc	s10	s11	s12	
2	s3			s2						6
3		r7	r7		r7		r7	r7	r7	
4	s3			s2						7
5	s3			s2						8
6		s4	s5		s9		s10	s11	s12	
7		r1	s5				s10	s11	s12	
8		r2	r2				s10	s11	s12	
9		r3	r3		r3					
10		r4	r4		r4					
11		r5	r5		r5					
12		r6	r6		r6					

对于正则表达式这门语言，我们用它来描述目标语言的词法构成规则，以便得到目标语言的词法分析的 NFA。目标语言的词法构成规则，是用正则表达式这门语言来编写的。因此，对正则表达式这门语言，不仅有词法分析，还有语法分析。对给定的目标语言，例如 TINY 语言，先要手工写出其词法构成规则的正则表达式，再使用正则表达式这门语言的语法分析器，来对给出的正则表达式进行语法分析。语法分析的目的是生成每个正则表达式的 NFA。

正则表达式这门语言中的每种运算，都对应有相应 NFA 的运算。对每种运算，要执行的 NFA 操作，如下表所示。

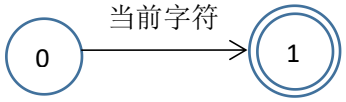
产生式序号	产生式β	产生式体中符号的数量|β|	关系	要调用的函数
1	E→ E | E	3	$E_{.NFA}$ = $E_{1.NFA}$ | $E_{2.NFA}$	NFA_Or_combine($E_{1.NFA}$, $E_{2.NFA}$)
2	E→ E & E	3	$E_{.NFA}$= $E_{1.NFA}$ & $E_{2.NFA}$	NFA_connect_combine($E_{1.NFA}$, $E_{2.NFA}$)
3	E→ (E)	3	$E_{.NFA}$= $E_{1.NFA}$	$E_{.NFA}$= $E_{1.NFA}$
4	E→ E#	2	$E_{.NFA}$= $E_{1.NFA}$ #	NFA_Plus_Closure($E_{1.NFA}$)
5	E→ E@	2	$E_{.NFA}$= $E_{1.NFA}$ @	NFA_Closure($E_{1.NFA}$)

6	E→ E?	2	$E_{.NFA}$= $E_{1.NFA}$?	NFA_Zero_one($E_{1.NFA}$)
7	E→ id	1	$E_{.NFA}$= $id_{.NFA}$	$E_{.NFA}$= $id_{.NFA}$

在上述正则表达式这门语言的上下文无关文法的 DFA 中：

规约的状态共有 7 个，分别是：

I_3：产生最基本的 NFA：



I_7：如果当前输入符是|，或者结束符，那么就要规约，调用函数 NFA_Or_combine($E_{1.NFA}$，$E_{2.NFA}$)得到一个新的 NFA，然后将其赋给规约得到的 $E_{.NFA}$；

I_8：如果当前输入符是|，或者&，或者结束符，那么就要规约，调用函数 NFA_connect_combine($E_{1.NFA}$，$E_{2.NFA}$)得到一个新的 NFA，然后将其赋给规约得到的 $E_{.NFA}$；

I_9：无条件规约，把 $E_{1.NFA}$ 的值，将其赋给规约得到的 $E_{.NFA}$；

I_{10}：无条件规约，调用函数 NFA_Plus_Closure($E_{1.NFA}$)得到一个新的 NFA，然后将其赋给规约得到的 $E_{.NFA}$；

I_{11}：无条件规约，调用函数 NFA_Closure($E_{1.NFA}$)得到一个新的 NFA，然后将其赋给规约得到的 $E_{.NFA}$；

I_{12}:无条件规约，调用函数 NFA_Zero_one($E_{1.NFA}$)得到一个新的 NFA，然后将其赋给规约得到的 $E_{.NFA}$；

正则表达式语言中语法分析及翻译的通用代码

有两个栈：state_stack 和 object_stack;

其中 state_stack 栈中的元素的类型都为整型。其作用就是和 LR 语法分析表配合使用，完成语法分析。而 object_stack 栈和 state_stack 栈是一一对应的，其作用就是为了完成翻译。

上下文无关文法中有终结符和非终结符。因此，我没定义如下数据结构：

```
class Base {
    char type;   //两种取值：’t’表示终结符；’n’表示非终结符
    string name;
}
```

对于上述正则表达式这门语言，其上下文无关文法中的非终结符 E，我们定义其数据结构为：

```
class E: class Base {
    int NFA_graph_id;
}
```

对某门语言，例如 TINY，它的词法构成规则用正则表达式这门语言来表述。表述的结果就是正则表达式。

对正则表达式进行语法分析，其目的是做翻译。翻译的目的就是生产其 NFA。因此，语法分析的过程中，每规约出一个 E，翻译的目的就是得出这个 E 的 NFA。所以要定义的 E 的综合属性就是 NFA_graph_id。每个 NFA 是用其主键 graph_id 标识，因此只须存储主键值 graph_id 即可。

注意：整个处理过程是：对给定的目标语言的正则表达式，执行语法分析，在语法分析的过程中完成正则表达式的 NFA 的构建。例如，对于 LR 语法分析，就是在每次规约时，都执行一次 NFA 操作。语法分析的过程中，会得到一个规约序列。这个规约序列，也就是一个 NFA 操作的序列。这个操作序列的结果，就是我们所要的正则表达式的 NFA。

LR 语法分析中，状态栈中的初始状态是 0 状态，此时的栈顶状态(top)为 0 状态。然后一轮一轮地执行如下操作。在每一轮中，执行的动作包括：1）调用词法分析函数，得到当前词素，把它赋给语法分析的当前符号（a）；2）查 LR 语法分析表中第 top 行，第 a 列的单元中的取值。这个取值有两种情形：不为空，空。如果为空，那就说明输入存在语法错误。只要发现语法错误，语法分析便结束。

当取值不为空时，又有三种情形，那就是移入，规约，完成。如果是移入，取值还会给出要移入的状态，要执行的操作就是把要移入的状态压入状态栈 state_stack 中，把移入的词素压入 object_stack 中。如果是规约，则执行规约操作。取值还会给出按哪个产生式规约，即取值为产生式的序号。如果是完成，那么表明整个语法分析已经完成。

为了代码的简单性，我们设 state_stack 栈的栈顶元素为 top，当前输入符为 a。然后查 LR 语法分析表 LRTable，得到的结果的数据类型是：

```
class Action {
    char type;   //两种取值：’s’表示移入； ’r’表示规约
    int  id;    //如果为移入，表示要移入的状态号；如果为规约，则表示产生式的序号，
}
```

调用函数 LRTable.get_ACTION_type(top, a)，返回一个 Action 类型的对象；

对于词法分析器，定义其输出的数据类型：

```
class Lexeme {
    char * type;   //两种取值：’id’表示自定义符； ’reserved’表示预定义符；
```

```
    char *  value;    //取值
}
```

对于一个正则表达式，其语法分析的代码如下：

```
state_stack.Clear( );
object_stack.Clear( );

state_stack.push( 0 );  //初始化：把 0 状态入栈
Action action;   //
int top;  //状态栈的栈顶元素的状态号
while (not state_stack.isEmpty( )  {
    Lexeme *a = Lexeme_parse( );   //调用词法分析函数得到当前输入符 a
    int top = state_stack.get_top_element();
    action = LRTable.get_ACTION_type(top, a)
    if (action->type ==  ‘s’ ) {     //移入
        next_state = action->id;

        //执行移入操作：
        object_stack.push((void *)a);
        state_stack.push(next_state);

    }
    else if (Action_type ==  ‘r’ )   {   //规约
          production_id = LRTable.get_ACTION_id(top, a)
          reduce(action->id,a);

          //接下来执行 GOTO 操作：
          char nonTerminal = Production.getHead(action->id); //由产生式 id 得出其
                                                              头部的非终结符
          top = state_stack.get_top_element();
          next_state = LRTable.get_GOTO_id(top, nonTerminal);
          state_stack.push(next_state);

     }
    else if (Action_type ==  ‘a’ )   {   //接受
            break;
    }
    else   {                            //语法错误
           正则表达式有语法错误;
           break;

}
```

注意：上述语法分析代码是通用的，固定的。

对于正则表达式这门语言，其上下文无关文法，基于语法制导的翻译方案 SDT 如下：

产生式序号	SDT
1	$E \rightarrow E_1 \mid E_2$ { NFA_Or_combine(E, E_1, E_2); }
2	$E \rightarrow E_1$ & E_2 { NFA_connect_combine(E, E_1, E_2); }
3	$E \rightarrow (E)$ { NFA_Set_Equal(E, E_1); }
4	$E \rightarrow E_1$# { NFA_Plus_Closure(E, E_1, E_2); }
5	$E \rightarrow E_1$@ { NFA_Closure(E, E_1); }
6	$E \rightarrow E_1$? { NFA_Zero_one(E, E_1); }
7	E→ id { generate_new_basic_NFA(E, id); }

```
//基于文法和 SDT 得到的规约函数：
void reduce( int production_id, Lexeme * a)   {
    switch(production_id)   {
       case 1:       //按 E→E | E 规约
          E *E  = new E( );
          E *E2  = (E *)object_stack.pop( );
          object_stack.pop( );
          E *E1  = (E *)object_stack.pop( );
          NFA_or_combine(E, E1, E2);
          object_stack.push((void *)E);
          state_stack.pop();  //该产生式的左部有三个符号，弹出三个
          state_stack.pop();
          state_stack.pop();
      break;

       case 2:       //按 E→E & E 规约
          E *E  = new E( );
          E *E2  = (E *)object_stack.pop( );
          object_stack.pop( );
          E *E1  = (E *)object_stack.pop( );
          NFA_connect_combine(E, E1, E2);
          object_stack.push((void *)E);
          state_stack.pop();  //该产生式的左部有三个符号，弹出三个
```

```
        state_stack.pop();
        state_stack.pop();
    break;

     case 3:          //按 E→(E)规约
        E *E  = new E( );
        object_stack.pop( );
        E *E1  = (E *)object_stack.pop( );
        object_stack.pop( );
        NFA_Set_Equal( E, E1);
        object_stack.push((void *)E);
        state_stack.pop();  //该产生式的左部有三个符号，弹出三个
        state_stack.pop();
        state_stack.pop();
    break;

     case 4:          //按 E→E#规约
        E *E  = new E( );
        object_stack.pop( );
        E *E1  = (E *)object_stack.pop( );
        NFA_plus_closure(E, E1);
        object_stack.push((void *)E);
        state_stack.pop();  //该产生式的左部有二个符号，弹出二个
        state_stack.pop();
    break;

  case 5:          //按 E→E@规约
        E *E  = new E( );
        object_stack.pop( );
        E *E1  = (E *)object_stack.pop( );
        NFA_closure(E, E1);
        object_stack.push((void *)E);
        state_stack.pop();  //该产生式的左部有二个符号，弹出二个
        state_stack.pop();
    break;

  case 6:          //按 E→E? 规约
        E *E  = new E( );
        object_stack.pop( );
        E *E1  = (E *)object_stack.pop( );
        NFA_zero_one(E, E1);
        object_stack.push((void *)E);
        state_stack.pop();  //该产生式的左部有二个符号，弹出二个
        state_stack.pop();
```

```
            break;

        case 7:          //按 E→id 规约
            E *E  = new E( );
            Lexeme *id = (Lexeme *)object_stack.pop( );
            generate_new_basic_NFA(E, id);
            object_stack.push((void *)E );
            state_stack.pop();  //该产生式的左部有一个符号，弹出一个
            break;

            default:
            break;
    }
}
```

对规约函数，我们可以看到，完全可以由工具基于给定的上下文无关文法和 SDT 自动生成。

因此**语法分析及翻译代码，完全可以由语法分析器自动生成工具来自动生成。**

对于 NFA 的操作函数，即 generate_new_basic_NFA，NFA_or_combine，NFA_connect_combine，NFA_plus_closure，NFA_plus_closure，NFA_zero_one 这些函数的实现，在有了上述 NFA/DFA 的数据结构后，就很容易实现了。这些函数的实现要人工来完成。请学生自己写出。

得出了目标语言词法构成规则的正则表达式的 NFA 后，再把它转化成 DFA，就完成了目标语言词法构成规则的正则表达式的 DFA 构建。把目标语言的每个正则表达式的 DFA 合成一个 DFA，也就得到了目标语言的词法分析器。

为了简单起见，对 TINY 语言，我们进一步假定变量的起始部分不允许是保留字。这样保留字和变量就不相关了。于是，保留字和变量，它们两者的 DFA 合成就简单了，直接合到一起，只要修改其中一个的状态号即可。做成功之后，再来扩展，再来考虑变量和保留字的相关问题。

语法分析工具做出来，80%的编译知识就掌握了，那个上甘岭战役就挺过来了。

试验课可多个人组团奋战，分工合作。真正做出来的，经老师检查合格的，最终成绩可额外加分，可观的分。