判断题

1️⃣ ε是字母表$\{\mathrm{a},\mathrm{d}\}$上的符号串 ✅ 对

- 空串ε是任何字母表上的有效符号串，并且该符号串长度为0

2️⃣ 当一个DFA运行过程中消耗掉输入串x后，所能到达的状态是$\hat{\delta}(q_0,x)$，其中q_0是开始状态 ✅ 对

- $\delta(q,a)$是标准转移函数，表示q在读入单个输入符号a后所到达的状态
- $\hat{\delta}(q,x)$是扩展转移函数，归纳地定义了从状态q开始，读取完整个字符串x后所到达的状态

3️⃣ 如果n状态DFA定义的语言是无穷的，那么这个语言中某元素长度大于n ✅ 对

- n状态DFA，意思是一共有n个不同状态的有穷自动机
- 语言是无穷的，即DFA所接收的字符串的长度可以是无限的，即DFA中必定有环路
- 语言中某元素长度大于n，即DFA所接收的语言中，至少一个字符长度大于n
  - 直观来讲，都说了语言是无穷的(字符串可以无限长)，那肯定有长度大于n的串
  - 反过来说，如果语言中存在长度大于n的串，那必定是因为DFA中有环路

4️⃣ 由0和1组成的串且其中0和1的个数相等，该语言是正则语言 ❌ 错

- 直觉来讲，构造不出接收一个语言的FA，这个语言就不是正则语言
- 假设某串中有n个0和n个1，设计一个flag表示状态，读入一个1则$\mathrm{flag}++$，读入一个0则$\mathrm{flag}--$，
- 则flag一共需要$2n$个状态，当$n\to\infty$时就需要无穷个状态
- FA的状态有限，所以不能用FA描述，所以不是正则语言

5️⃣ 在上下文无关文法中，变元集合可以为空 ❌ 错

- CFG的含义，对所有产生式$\forall\alpha\to\beta\in P$，$\alpha$必须是非终结符，亦可写作$A\to\beta$
- 变元即非终结符，都说了产生式左部必须为非终结符，非终结符的集合就不可能为空

6️⃣ 句子的句柄也是该句了的直接短语 ✅ 对

- 句柄就是特殊的直接短语
- 句型：从开始符，推导过程中任意的字符串
- 短语：推导过程中的一个句型中，可以逆着推导过程多步规约回某个非终结符的子串
- 直接短语：一个句型中，可以一步规约回某个非终结符的子串，因此其一定是某个产生式的右部
- 句柄：自底向上分析中执行规约的对象，规定为一个句型中最左边的那个直接短语

7️⃣ 自上而下语法分析过程中，M为预测分析表，元素$M[N,c]$中为产生式$N\to\alpha$，那么$c\in\mathrm{FIRST}(\alpha)$ ❌ 错

- 首先$\mathrm{LL}(0)$预测分析表大致长这样

非终结符	输入符号					
	id	+	*	(	)	$
E	$E\to TE'$			$E\to TE'$		
E'		$E'\to +TE'$			$E'\to\varepsilon$	$E'\to\varepsilon$
T	$T\to FT'$			$T\to FT'$		
T'		$T'\to\varepsilon$	$T'\to *FT'$		$T'\to\varepsilon$	$T'\to\varepsilon$
F	$F\to$id			$F\to(E)$		

- 元素$M[N,c]$中为产生式$N\to\alpha$，则说明了$c\in\mathrm{SELECT}(N\to\alpha)$
- 但是关于$\mathrm{FIRST}(\alpha)$则要分情况讨论
  - 当$\mathrm{FIRST}(\alpha)$中有空串时，$\mathrm{SELECT}(N\to\alpha)=\{\mathrm{FIRST}(\alpha)\backslash\{\varepsilon\}\}\cup\mathrm{FOLLOW(N)}$
    - 如果α就是空串的话，$\mathrm{SELECT}(N\to\alpha)=\mathrm{FOLLOW(N)}$
  - 当$\mathrm{FIRST}(\alpha)$中无空串时，$\mathrm{SELECT}(N\to\alpha)=\mathrm{FIRST}(\alpha)$，这是才一定有$c\in\mathrm{FIRST}(\alpha)$

8️⃣ 如果语言不允许过程递归调用，那么同一个过程的活动的生命期都不会相交✅对

- 先来看基本概念
  - 首先所谓的过程即函数，每对过程(函数)调用一次就产生一个活动
  - 活动利用调用栈管理，一个活动对应一个栈帧(参数/变量/返回地址)，函数被调用后栈帧就入栈
  - 生命周期，即一个函数对应的栈帧从入栈到出栈的过程
- 再来看这段表述
  - 允许同一函数递归，则调用栈可能会出现如下情形

```
-
- func(4)
- func(4) -> func(2)
- func(4) -> func(2) -> func(1) 递归终结
- func(4) -> func(2)
- func(4)
-
```

    - 某些过程的生命周期必定依附于另一个过程的生命周期，故一定有相交
  - 不允许递归的话则好理解了，必定是串行的，无相交

```
-
- func(7)
-
- func(4)
-
```

9️⃣ 编译器的源语言与它的目标语言可以相同 ✅对

- 编译器就是要把源语言翻译成目标语言，这俩一样不吃屎吗，NO这只是直觉
- 其实二者可以一样的，因为其实还有重编译器，用来重构优化代码

0️⃣ 在设计词法分析器时，实数这个词法单位采用全体一种表示比较合理✅对

- 词法分析器关注的是——这玩意是啥
- 而所有的实数本质上都可以记为形如`3.14e0`形式，正是这一统一形式，可以将实数统一为一种表示
- 反例，如实数和整数就不能识别成有一个东西，大多编程语言都将二者分开

填空题

1️⃣ 符号串s是语言S中的句子，则$s{\cdot}s$是语言___中的句子(答案：SS)

- 假设语言$S=\{s,t,r\}$
- 则$SS=\{s,t,r\}\{s,t,r\}=\{ss,st,sr,ts,tt,tr,rs,rt,rr\}$
- 可见$ss\in SS$

2️⃣ NFA-M的开始状态不是结束状态，那么M不接受符号串___(答案：空串)

- 这题似乎缺了一个限制条件，就是排除带空边的NFA
- 加上这个先之后很好理解，从开始状态到达结束状态必定存在状态转移，就不可能是空串

3️⃣ 从声明语句`int a[2]`获得的有用信息有；维数是`___`；维长是`___`；元素类型是`___`(答案：`1/2/int`)

- 类型是`int`不必多说
- 数组有几个括号就是几维的，`a[2]`的维数是`1`
- `a[2]`中的`2`定义了该维度的长度，所以维长为`2`

4️⃣ 对照语法树，结点N的综合承雇性值只依赖___的属性值(答案：孩子结点)

- 基本概念了
  - 综合属性：只依赖于其孩子结点
  - 继承属性：依赖于其左边的兄弟(兄终弟即)，或者其父亲结点(父传子位)

5️⃣ 有过程声明`void f(int x, float y) {...}`，现要访问`f`的活动记录中`x`单元，那么基址是`___`偏移取是`___`。注不含参数个数单元(答案：`fp/+2`)

```
形参单元
访问链
控制链     <-- fp
返回地址
局部变量
临时变量   <-- sp
```

- 基地址：也就是栈指针`fp`，一般指向栈帧(活动记录)中的控制链
- 函数传入两个形参，`y`先入栈`x`随后，所以大致格局如下，偏移量为`+2`

```
高地址：形参单元y
      ：形参单元x     <-- fp + 2
      ：访问链        <-- fp + 1
低地址：控制链        <-- fp
(略)
```

6️⃣ 表达式 `x-(b+c)*a` 的逆波兰表示为`___`，三地址码示为`___`(答案：逆波兰为`xbc+a*-`，三地址见下)

- 三地址表示

```
x1 = b + c
x2 = x1 * a
x3 = x - x2
```

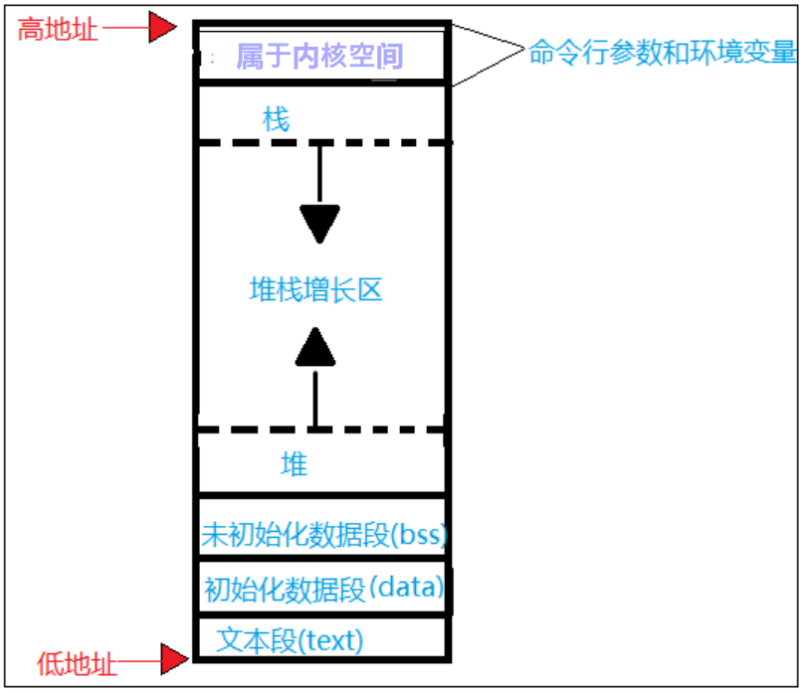
- 逆波兰表示：将操作数挪到后面去，如`a b +`

当前字符	操作符栈	输出 (Output)	说明
x		`x`	操作数可以直接输出
-	`-`	`x`	空栈时`-`直接入栈
(	`-(`	`x`	`(`可以直接入栈
b	`-(`	`x b`	操作数可以直接输出
+	`-(+`	`x b`	`+`遇到栈顶`(`直接入栈
c	`-(+`	`x b c`	操作数可以直接输出
)	`-`	`x b c +`	遇到`)`，输出栈顶直到遇见`(`，丢弃`()`
*****	`-*`	`x b c +`	`*`的优先级高于栈顶的`-`，所以`*`入栈
a	`-*`	`x b c + a`	操作数可以直接输出
		`x b c + a * -`	将栈中剩余的`*`和`-`依次弹出到输出

  - 遇到操作数直接输出
  - 遇到`(`直接入栈
  - 遇到`)`则符号栈持续输出栈顶，直符号栈中遇到`(`为止，但是这一对括号`()`则不输出直接丢弃
  - 遇到运算符`^+-*/`：其中`^`最高，其次`*/`的优先级高于`+-`
    - 直接入栈的情形：符号栈为空，符号栈顶为`(`，当前算符优先级高于栈顶
    - 其他情形：当前输入优先级小于等于栈顶时，持续输出栈顶直到大于栈顶为止，再将其入栈
    - 特殊情况：`^`遇到栈顶也为`^`时，反而"破例"入栈

7️⃣ 程序代码存放在运行时存储空间的`___`区(答案：代码`text`)

- 基本概念了见下，在文本段(代码区)



1. 代码区(文本段)：存放程序编译好的二进制机器码，只读，存储方向自底向上
2. 静态区：存储全局变量/静态变量，可读写，，存储方向自底向上
    - 数据段：存放已显式初始化的全局变量和静态变量
    - BSS段：存放未被初始化/初始化为0的全局变量和静态变量，不用显式地记录变量的值
3. 堆：用于内存动态分配，如`malloc`和`new`时所需空间就从这里割一块，并且是自底向上扩展
4. 栈：用于函数调用(局部环境)，每调用一函数就会在栈顶创建一个栈帧(存放参数/局部变量/返回地址)
    - 栈增长方向为地址减小方向

简答题

1️⃣ 范围`-127 ~ +127`的小整数，用十六进制表示时最多有两位，用正则表达式定义十六进制表示的小整数。举例十六进`9`/`-B`/`-1F`/`+7F`依次为十进制`9`/`-11`/`-31`/`+127`

- 符号部分`[+-]?`：表示可以是`+`或`-`，并出现`0`或`1`次
- 数组部分，分情况讨论
    - 一位十六进制数，范围是`0-F`则正则表达式为`[0-9A-F]`
    - 二位十六进制数，范围是`10-7F`，第一位为`[1-7]`第二位为`[0-9A-F]`，合一起为`[1-7][0-9A-F]`
    - 以上情况是二或一的情形，具体可以体现为`[1-7]`出现(二位)或者不出现(一位)，即`[1-7]?[0-9A-F]`
- 综上`[+-]?[1-7]?[0-9A-F]`，当然也可以笨一点`[+-]?([1-7][0-9A-F]|[0-9A-F])`
- 或者写成`(+|-|ϵ)(1|2|3|4|5|6|7|ϵ)(0|1|2|3|4|5|6|7|8|9|A|B|C|D|E|F)`

2️⃣ 消除文法中的无用符号：`S->Aa|ϵ, A->Aa, B->Bc|d`

- 首先看看哪些非终结符推导不出终结符串，
    - `B->Bc|d`推导出`dcccc....`(左递归实在看不顺眼，可以转化为`B->dX, X->cX|ϵ`)
    - `A->Aa`推导出`Aaaa....`永远无法消掉`A`，所以直接干掉删了`S->Aa`和`A->Aa`

    - 现在只剩`S->ϵ，B->Bc|d`
- 再消除不可达的符号，即不能由`S`推导而来的符号
    - 只有`S->ϵ`，所以其它全删，最后就只剩`S->ϵ`

3️⃣ 消除文法中的`ϵ`产生式：`S->ABC|ϵ，A->Bb|a，B->Cb|ϵ，C->ϵ`

- 首先找出所有可空变量，即能推导出空串的非终结符
    - 有`S->ϵ，B->ϵ，C->ϵ`只剩下`A`了
    - 而`A->a`或者`A->Bb->b`或者`A->Bb->Cbb->bb`故不是可空变量
- 先强行"删掉"所有(除开始符的)空产生式`S->ABC|ϵ，A->Bb|a，B->Cb`
    - 注意如果开始符有空产生式，一定是不能删的
- 改造上面的产生式，右部的可空变量可以删除或者不删除
    - `S->ABC`变为`S->ABC|AB|AC|A|ϵ`
    - `A->Bb|a`变为`A->Bb|b|a`
    - `B->Cb`变为`B->Cb|b`
- 记得还要化简文法，和第题类似
    - 找出推导不出终结符串的非终结符，可见`A->a`和`B->b`而没一个`C`在左部，故删掉`C`
        - `S->AB|A|ϵ`
        - `A->Bb|b|a`
        - `B->b`
    - 再删除从`S`推导不可达的符号
        - `S->AB|A`故`AB`都可达
        - `S->AB->BbB|bB|aB`故`ab`也可达，不用删
    - 最终的文法(标准化一点)
        - `S'->S|ϵ`
        - `S->AB|A`
        - `A->Bb|b|a`
        - `B->b`

4️⃣ 消除文法中的单位产生式：`E->E+T|T，T->F|T*F，F->i|(E)`

- 单位产生式，即产生式的右部只包含一个非终结符，如此处的`E->T，T->F`
    - 这一过程只是简单的非终结符替换，卵用没有所以有必要消除
- 第一步：找出每个非终结符一步或者多步推导可能得到的非终结符
    - `F`推导集：只能导出其自己，故`{F}`
    - `T`推导集：有`T->F`和他自己，故`{F,T}`
    - `E`推导集：有`E->T`和`E->T->F`和他自己，故`{E,T,F}`
- 第二步：构建新产生式，即如果`A`的推集中有`B`，且`B->α`(非单位产生式)，则`A->α`
    - 对`F`及其推导集`{F}`：有`F->i|(E)`，故`F->i|(E)`

    - 对`T`及其推导集`{F,T}`：
        - `T`和`F`：有`F->i|(E)`，故`T->i|(E)`
        - `T`和`T`：有`T->T*F`，故`T->T*F`
    - 对`E`及其推导集`{E,F,T}`：
        - `E`和`T`：有`T->T*F`，故`E->T*F`
        - `E`和`F`：有`F->i|(E)`，故`E->i|(E)`
        - `E`和`E`：有`E->E+T`，故`E->E+T`
- 综上
    - `E->E+T|T*F|i|(E)`
    - `T->T*F|i|(E)`
    - `F->i|(E)`

5️⃣ 消除文法中的左递归：`S->AB|a, A->Ab|Ba, B->Ac|d`

- 首先确定处理非终结符的顺序`S,A,B`其中`S`自然没有左递归
- 对于`A`：直接左递归的是`A->Ab|Ba`(左边候选式)，其推导出来的是`A->Babbb.....`，可以转化为
    - `A->BaX`以及`X->bX|ϵ`
    - 文法变成了`S->AB|a, A->BaX, X->bX|ϵ, B->Ac|d`
- 对于`B`：不要忘了间接的左递归
    - 利用`B->Ac|d`和`A->BaX`，得到左递归`B->Ac|d->BaXc|d`
    - 此时的文法变为了`S->AB|a, A->BaX, X->bX|ϵ, B->BaXc|d`
    - 其推导出的是`B->d aXc aXc aXc.....`，可转化为`B->dY`以及`Y->aXcY|ϵ`
    - 文法变为了`S->AB|a, A->BaX, X->bX|ϵ, B->dY, Y->aXcY|ϵ`

6️⃣ 对文法`S->P|o, P->i(B)SF, F->eS|ϵ, B->0|1`，写出`FIRST(S,F)`和`FOLLOW(F,B)`

- `FIRST(A)`即从`A`出发能推导出的第一个终结符的集合
    - 对于`FIRST(S)`
        - `S->o`所以`FIRST(S)={o}`
        - `S->P->i(B)SF`所以`FIRST(S)={o,i}`
    - 对于`FIRST(F)`
        - `F->ϵ`所以`FIRST(F)={ϵ}`
        - `F->eS`所以`FIRST(F)={e,ϵ}`
- `FOLLOW(A)`即跟在非终结符`A`后的终结符集合
- 第一步：首先确定依赖关系
    - `S->P`所以`..S<FOLLOW(S)>..->..P<FOLLOW(S)>..`，`FOLLOW(P)`含`FOLLOW(S)`
    - `P->i(B)SF`所以`..P<FOLLOW(P)>..->..i(B)SF<FOLLOW(P)>..`，`FOLLOW(F)`含`FOLLOW(P)`
    - 考虑`F->ϵ`则`P->i(B)S`，即`..P<FOLLOW(P)>..->..i(B)S<FOLLOW(P)>..`，`FOLLOW(S)`含`FOLLOW(P)`

    - `F->eS|ϵ`所以`F->eS`所以`..F<FOLLOW(F)>..->..eS<FOLLOW(F)>..`，`FOLLOW(S)`含`FOLLOW(F)`
- 第二步：整合依赖关系
    - `FOLLOW(P)`含`FOLLOW(S)`，以及`FOLLOW(S)`含`FOLLOW(P)`，所以`FOLLOW(S) = FOLLOW(P)`
    - `FOLLOW(P)`含`FOLLOW(S)`，`FOLLOW(S)`含`FOLLOW(F)`所以`FOLLOW(P)`含`FOLLOW(F)`
        - 以及`FOLLOW(F)`含`FOLLOW(P)`，所以`FOLLOW(F) = FOLLOW(P)`
    - 综上`FOLLOW(F) = FOLLOW(P) = FOLLOW(S)`
- 第三步：开始迭代
    - 初始状态：将`$`放入开始符的`FOLLOW`集

```
FOLLOW(S) = {$}
FOLLOW(P) = {}
FOLLOW(F) = {}
FOLLOW(B) = {}
```

    - 分析`S->P|o`：除了已经分析出的依赖关系，没什么新的信息
    - 分析`P->i(B)SF`：除了已经分析出的依赖关系，还有——

```
FOLLOW(S) = {$,e}
FOLLOW(P) = {}
FOLLOW(F) = {}
FOLLOW(B) = {)}
```

        - 由于`(B)`的存在，所以毫无疑问`FOLLOW(B)`包含`{)}`
        - 由于`SF`的结构，所以`FOLLOW(S)`包含`FIRST(F)\{ϵ}`，所以`FOLLOW(S)`包含`{$,e}`
        - 排除`ϵ`是因为`F->ϵ`会让`F`消失，`SF`中`F`消失了`S`后边还跟个寂寞
    - 分析`F->eS|ϵ`：除了已经分析出的依赖关系，没什么新的信息，于是代入依赖关系

```
FOLLOW(S) = {$,e}
FOLLOW(P) = {$,e}
FOLLOW(F) = {$,e}
FOLLOW(B) = {)}
```

7️⃣ 对于文法`E->E/E|E&E|i`，写出串`i/i&i`的所有最左推导，并判断该文法是不是歧义的

- 再明显不过的突破口就是`i/i&i = (i/i)&i = i/(i&i)`
- `(i/i)&i`情形
    - `E -> E&E -> E/E&E -> i/E&E -> i/i&E -> i/i&i`
- `i/(i&i)`情形
    - `E -> E/E -> i/E -> i/E&E -> i/i&E -> i/i&i`
- 文法有歧义的意思就是，该文法接收的串有超过一种分析树(推导方式)，所以这个文法一定有歧义

解答题

第五题

1️⃣ 下表是一个NFA迁移表，填写补充最右边一列`ϵ-`闭包列

	a	b	ε	ε—闭包
$\rightarrow 1$	$\{2,3\}$		$\{3\}$	(1)
2		$\{3\}$	$\{3,4\}$	(2)
3	$\{4\}$			(3)
4		$\{5\}$		(4)
^{*5}			$\{1\}$	(5)

- `ϵ-`闭包的含义：从`1`状态出发，不消耗任何字符串(仅靠`ϵ`)，所能到达的状态的集合，记作`ϵ-CLOSURE(1)`
- 对于状态`1`：除了其自身，通过`ϵ`有`1 -> 3`，故`ϵ-CLOSURE(1)={1,3}`
- 对于状态`2`：除了其自身，通过`ϵ`有`2 -> 3,4`，故`ϵ-CLOSURE(1)={2,3,4}`
- 对于状态`3`：除了其自身，没了，故`ϵ-CLOSURE(3)={3}`
- 对于状态`4`：除了其自身，没了，故`ϵ-CLOSURE(4)={4}`
- 对于状态`5`：除了其自身，通过`ϵ`有`5 -> 1 -> 3`，故`ϵ-CLOSURE(4)={5,1,3}`

2️⃣ 将这个NFA转化为DFA(写出迁移表)，其中DFA状态用NFA状态集合表示，此处解答采用子集转换法

	a	b	ε	ε—闭包
$\rightarrow 1$	$\{2,3\}$		$\{3\}$	$\{1,3\}$
2		$\{3\}$	$\{3,4\}$	$\{2,3,4\}$
3	$\{4\}$			$\{3\}$
4		$\{5\}$		$\{4\}$
^{*5}			$\{1\}$	$\{1,3,5\}$

- DFA的开始状态，是NFA开始状态的`ϵ-`闭包，即`{1,3}`
- 从`{1,3}`出发接收输入，试图得到其它状态`δ_DFA(S,a) = ϵ-CLOSURE(move(S,a))`

```
{1,3} --a-> {2,3,4}      求出ϵ-CLOSURE({2,3,4})={2,3,4}✅新状态
{1,3} --b-> {}
{1,3} --ϵ-> {3}          //记住这里不应该出现ϵ的状态转移

{2,3,4} --a-> {4}        求出ϵ-CLOSURE({4})={4}✅新状态
{2,3,4} --b-> {3,5}      求出ϵ-CLOSURE({3,5})={1,3,5}✅新状态

{4} --a-> {}
{4} --b-> {5}            求出ϵ-CLOSURE({5})={1,3,5}

{1,3,5} --a-> {2,3,4}    求出ϵ-CLOSURE({2,3,4})={2,3,4}
{1,3,5} --b-> {}
```

  - 注意`ϵ-CLOSURE({a,b}) = ϵ-CLOSURE({a}) + ϵ-CLOSURE({b})`

- 一共有状态 `{1,3} {2,3,4} {4} {1,3,5}`
- 画出状态转化表：`ε` 列直接无

	a	b
$\rightarrow \{1,3\}$	$\{2,3,4\}$	
$\{2,3,4\}$	$\{4\}$	$\{1,3,5\}$
$\{4\}$		$\{1,3,5\}$
$^*\{1,3,5\}$	$\{2,3,4\}$	

第六题

1️⃣ 题目：

1. 给定上下文无关文法

```
1. S -> E-n
2. S -> +
3. E -> n
4. E -> n+
```

2. 构造DFA，来识别该文法中的每个活前缀(即$\mathrm{LR}(0)$项目集规范族)
   - 所谓活前缀，其实就是符号栈中可能出现的内容
3. 判断是否有冲突(如规约规约冲突/规约移进冲突)，如果有冲突则解决

2️⃣ 构造DFA(即所谓$\mathrm{ItemDFA}$)

1. 首先将文法增广，并确定项目

产生式	项目
`S'-> S`	`S'-> •S`，`S'-> S•`
`S -> E-n`	`S -> •E-n`，`S -> E•-n`，`S -> E-•n`，`S -> E-n•`
`S -> +`	`S -> •+`，`S -> +•`
`E -> n`	`E -> •n`，`E -> n•`
`E -> n+`	`E -> •n+`，`E -> n•+`，`E -> n+•`

2. 找出等价的项目，如果一项集包含`A -> α•Bβ`，则一定也要包含`B -> •γ`
   - 首先以`S'-> •S`为核心构建出初始`I0`状态

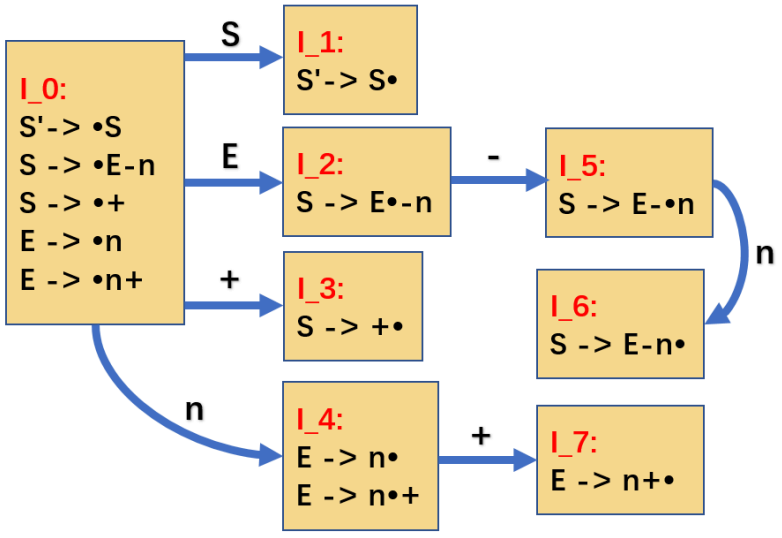
状态	内核项目	等价项	等价项(二阶)
`I0`	`S'-> •S`	`S -> •E-n`，`S -> •+`	`E -> •n`，`E -> •n+`

   - 让`I0`分别接收`S/E/+/n`以进入`I1/I2/I3/I4`状态

状态	项目
I0	S'-> •S, S -> •E-n, S -> •+, E -> •n, E -> •n+
I1	S'-> S•
I2	S -> E•-n
I3	S -> +•
I4	E -> n•, E -> n•+

- I1/I2/I3/I4中I2和I4中还有移进项目，所以让二者各自在接收一些串，达到新的状态

状态	项目
I0	S'-> •S, S -> •E-n, S -> •+, E -> •n, E -> •n+
I1	S'-> S•
I2	S -> E•-n
I3	S -> +•
I4	E -> n•, E -> n•+
I5	S -> E-•n
I6	S -> E-n•
I7	E -> n+•



3️⃣ 冲突与解决

1. 构建出分析表
   - 对移进项目而言，若I_a跨过一终结符x到达I_b，则ACTION[a,x]=sb
   - 对移进项目而言，若I_a跨过一非终结符X到达I_b，则ACTION[a,X]=b

- 对规约项目而言，$\mathrm{LR}(0)$分析直接将其`ACTION`暴力设为`r<产生式编号>`
- 对接收项目而言，设定`ACTION`的`$`单元为`acc`

状态	ACTION表+	ACTION表-	ACTION表n	ACTION表$	GOTO表S	GOTO表E
I0	s3		s4		1	2
I1				acc		
I2		s5				
I3	r2	r2	r2	r2		
I4	s7/r3	r3	r3	r3		
I5			s6			
I6	r1	r1	r1	r1		
I7	r4	r4	r4	r4		

2. 很显然是移入规约冲突，冲突的解决就是采用$\mathrm{SLR}(1)$分析
   - $\mathrm{LR}(0)$文法中，对所有的非终结符都暴力规约
   - 而$\mathrm{SLR}(1)$文法中，只对产生式左部的的`FOLLOW`集中的终结符进行规约

```
1. S -> E-n
2. S -> +
3. E -> n
4. E -> n+
```

     - `r2`规约产生式是`S -> +`，`FOLLOW(S) = {$}`
     - `r3`规约产生式为`E -> n`，`FOLLOW(E) = {-}`
     - `r1`规约产生式为`S -> E-n`，`FOLLOW(S) = {$}`
     - `r4`规约产生式为`E -> n+`，`FOLLOW(E) = {-}`
   - 最后更新符号表为

状态	ACTION表+	ACTION表-	ACTION表n	ACTION表$	GOTO表S	GOTO表E
I0	s3		s4		1	2
I1				acc		
I2		s5				
I3				r2		
I4	s7	r3				

状态	ACTION表 +	ACTION表 -	ACTION表 n	ACTION表 $	GOTO表 S	GOTO表 E
I5			s6			
I6				r1		
I7		r4				

第七题

1️⃣ 题目

1. 源程序

```
program test;
  procedure foo(var y:integer);
  begin
    writeln(y);
  end;

  procedure bar(procedure t; var x:integer);
  begin
    t(x);
  end;

  procedure hool;
    var x:integer;
  begin
    x := 3;
    bar(foo, x);
  end;

begin // 主程序入口
  hool;
end.
```

2. 补充信息

   - 栈帧结构

```
形参单元
访问链
控制链     <-- fp
返回地址
局部变量
临时变量   <-- sp
```

   - 栈增长方向：向地址减小的方向生长
   - 数据长度：一律为1

3. 核心任务：程序执行到 `foo` 过程体内的状态是什么

- `test` 帧：`100，099，098`
- `hool` 帧：`097，096，095，094`
- `bar` 帧：`093，092，091，090，089，088，087`
- `foo` 帧：`086，085，084，083`

2️⃣ 解答：过程执行顺序：`test -> hool -> bar -> foo`

1. `test` 的执行：
   - `100` 访问链：指向 `test` 函数的外层函数，`test` 为主程序再也没有外层函数，所以为 `NIL`
   - `099` 控制链：指向调用 `test` 的函数，`test` 为主程序不可能被调用，所以设为 `0`
   - `098` 返回地址：执行后只能返回给操作系统，所以记为 `0`
2. `hool` 的执行：
   - `097` 访问链：指向 `hool` 函数的外层函数即 `test` 的栈帧指针(`test` 控制链)，即 `99`
   - `096` 控制链：指向调用 `hool` 的函数即 `test` 的栈帧指针(`test` 控制链)，即 `99`
   - `095` 返回地址：返回 `test` 主程序中某个地址，无法确定故只能记作 `<返址>`
   - `094` 局部变量：这里的局部变量是 `x=3`，所以直接值为 `x:3`
3. `bar` 的执行：注意靠外的参数 `x` 先入栈
   - `093` 形参 `x` 的地址：注意不是 `x` 的值而是对 `x` 的引用，故应该为其地址 `94`
   - `092` 形参 `foo` 的访问链：指向 `foo` 的外层函数即 `test` 的栈帧指针(`test` 控制链)，即 `99`
   - `091` 形参 `foo` 的地址：也就是 `foo` 的入口地址，记作 `foo@label`
   - `090` 访问链：指向 `bar` 的外层函数 `test` 的栈帧指针(`test` 控制链)，即 `99`
   - `089` 控制链：指向调用 `bar` 的函数 `hool` 的栈帧指针(控制链)，即 `096`
   - `088` 返回地址：`bar` 执行完后要返回给 `hool` 中某地址，记为 `<返址>`
   - `087`：无内容
4. `foo` 的执行：
   - `086` 形参单元：最终传入 `foo` 的形参本质上还是 `x`，所以此处为 `x` 的地址 `094`
   - `085` 访问链：指向 `foo` 的外层函数即 `test` 的栈帧地址(`test` 控制链)，即 `99`
   - `084` 控制链：指向调用 `foo` 的函数即 `bar` 的栈帧指针(控制链)，即 `089`
   - `083` 返回地址：`foo` 执行完返回给调用 `foo` 的函数 `bar` 的某个位置，即 `bar_addr`

3️⃣ 附加：程序执行到 `foo` 过程时，`fp/sp` 是多少

1. `fp` 指向控 `foo` 制链 `084`
2. `sp` 指向整个的栈顶 `083`

第八题

干，题目咋那么长，妈的放弃了，大四下忙的要死不说，还来考这B考试，课又讲的真尼玛抽象。真心奉劝各位学弟学妹出国交换能换的课尽量换，别把核心专业课拖到大四下