第四章 语法分析
Syntax Analysis

Part II

4、LL(1)预测分析法

- 常用终结符号集计算
- LL(1)文法
- LL(1)预测分析表
- LL(1)预测分析方法

1) 常用终结符号集

- First(**α**)
- Follow(A)
- Select(**A→β**)

FIRST集 和 FOLLOW集

- 对于$\forall\alpha\in(V_T\cup V_N)^*$，定义α的首符号集
  - $FIRST(\alpha)=\{a|\alpha\Rightarrow^*a\ldots，\ a\in V_T\}$
  - 若$\alpha\Rightarrow^*\varepsilon$，则规定 $\varepsilon\in FIRST(\alpha)$

- 对于$\forall A\in V_N$，定义A的后续符号集
  - $FOLLOW(A)=\{a\mid S\Rightarrow^*\ldots Aa\ldots，\ a\in V_T\}$

SELECT集

- 如果 β *=> ε
  - **Select(A→ β) = First(β)\{ ε } ∪ Follow(A)**
- 否则
  - **Select(A→ β) = First(β)**

例子

- S->pA
- S->qB
- A->cAd
- A->a

w=pccadd

例子

- S->Ap
- S->Bq
- A->a
- A->cA
- B->b
- B->dB

w_1=ccap

w_2=ddbq

例子

- S->aA
- S->d
- A->bAS
- A-> ε

w=abd

求FIRST(X) 的算法

1) 对$\forall X \in V_T$，FIRST(X)={X}

2) 对$\forall X \in V_N$，设置FIRST(X)的初值：

$$FIRST(X)=\begin{cases}\{a|X\rightarrow a\ldots\in P\} & X\rightarrow\varepsilon\notin P\\ \{a|X\rightarrow a\ldots\in P\}\cup\{\varepsilon\} & X\rightarrow\varepsilon\in P\end{cases}$$

3) 对$\forall X \in V_N$，重复如下过程，直到所有FIRST集不变

若 $X \rightarrow Y\ldots \in P$，且$Y \in V_N$，

则 $FIRST(X) = FIRST(X) \cup (FIRST(Y) - \{\varepsilon\})$

若 $X \rightarrow Y_1 \ldots Y_k \in P$，且$Y_1 \ldots Y_{i-1} \Rightarrow^* \varepsilon$，

则 对n=1到i-1:

$FIRST(X) = FIRST(X) \cup (FIRST(Y_n) - \{\varepsilon\})$

若 $Y_1 \ldots Y_k \Rightarrow^* \varepsilon$，

则 $FIRST(X) = FIRST(X) \cup \{\varepsilon\}$

例：表达式文法语法符号的FIRST集

FIRST(F)={ (, id }

FIRST(T)=FIRST(F)={ (, id }

FIRST(E)=FIRST(T)={ (, id }

FIRST(E')={ +, ε }

FIRST(T')={ *, ε }

FIRST(+)={+}, FIRST(*)={*}

FIRST(( )={ (}

FIRST( ))={) }

FIRST(id)={id}

E→TE'

E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→(E)|id

求FOLLOW(B)的算法

1) 先将 # 加入到 FOLLOW(S)

– #为句子结束符

对于所有非终结符，重复进行以下计算：

2) 若 $A \rightarrow \alpha B \beta$，

– 则 $FOLLOW(B)=FOLLOW(B) \cup FIRST(\beta)-\{\varepsilon\}$

3) 若$A \rightarrow \alpha B$或$A \rightarrow \alpha B \beta$，且$\beta \Rightarrow^{*} \varepsilon$，$A \neq B$，

– 则 $FOLLOW(B)=FOLLOW(B) \cup FOLLOW(A)$

例：表达式文法语法符号的 FOLLOW集

FIRST(F)= {(, id }

FIRST(T)= {(, id }

FIRST(E)= {(, id }

FIRST(E')={+, ε }

FIRST(T')={*, ε }

E→TE'

E'→+TE'|ε

T→FT'

T'→*FT'|ε

F→(E)|id

FOLLOW(E) = {), # }

FOLLOW(E') = FOLLOW(E) = {), #}

FOLLOW(T) = { +,), # }

FOLLOW(T') = FOLLOW(T) = { +,), # }

FOLLOW(F) = { *, +,), # }

例：表达式文法产生式的 SELECT集

FIRST(F)= {(, id }　　FOLLOW(E) = {), # }

FIRST(T)= {(, id }　　FOLLOW(E') = {), # }

FIRST(E)= {(, id }　　FOLLOW(T) = { +,), # }

FIRST(E')={+, ε }　　FOLLOW(T') = { +,), # }

FIRST(T')={*, ε }　　FOLLOW(F) = { *, +,), # }

SELECT(E→TE')={(, id }　　E→TE'

SELECT(E'→+TE’)={ + }　　E'→+TE'|ε

SELECT(E'→ε)={), # }　　T→FT'

SELECT(T→FT')={(, id }　　T'→*FT'|ε

SELECT(T'→ε)={ +,) , # }　　F→(E)|id

Review

- First(A)
- Follow(A)

The *first set* of a sequence of symbols u, written as First(u) is the set of terminals which start all the sequences of symbols derivable from u. A bit more formally, consider all strings derivable from u by a leftmost derivation. If u =>* v , where v begins with some terminal, that terminal is in First(u). If u =>* ε , then ε is in First(u).

The *follow set* of a nonterminal A is the set of terminal symbols that can appear immediately to the right of A in a valid sentence. A bit more formally, for every valid sentence S =>*uAv , where v begins with some terminal, that terminal is in Follow(A).

To calculate First(u) where u has the form X1X2...Xn, do the following:

1. If X1 is a terminal, then add X1 to First(u), otherwise add First(X1) - ε to First(u) .
2. If X1 is a nullable nonterminal, i.e., X1 =>* ε , add First(X2) - ε to First(u). Furthermore,

   if X2 can also go to ε , then add First(X3) - ε and so on, through all Xn until the first nonnullable one.
3. If X1X2...Xn =>* ε , add ε to the First set.

Follow()

1. Place # in Follow(S) where S is the start symbol and # is the input's right endmarker. The endmarker might be end of file, it might be newline, it might be a special symbol, whatever is the expected end of input indication for this grammar. We will typically use # as the endmarker.
2. For every production A –> uBv where u and v are any string of grammar symbols and B is a nonterminal, everything in First(v) except ε is placed in Follow(B).
3. For every production A –> uB, or a production A –> uBv where First(v) contains ε (i.e. v is nullable), then everything in Follow(A) is added to Follow(B).

例:

- S->aAd
- A->BC
- B->b | ε
- C->c | ε

求First(A), Follow(B)

例：

- **S → A B**
- **A → B a | ε**
- **B → D b | D**
- **D → d | ε**

Select(S → AB)=?

2) LL(1)文法

Top-Down Parsing

- 从左到右扫描输入串——寻找它的一个最左推导
- 文法G是 LL(1)文法的充要条件
  - 对于G 的每个非终结符 A 的任何两个不同的候选式 $A\rightarrow\alpha|\beta$，满足：

    (1) $FIRST(\alpha)\cap FIRST(\beta)=\phi$

    (2) 如果$\beta\Rightarrow^{*}\varepsilon$，则 $FISRT(\alpha)\cap FOLLOW(A)=\phi$

- LL(1)文法表示自顶向下方法能够处理的一类文法
  - 第一个 L 表示从左向右扫描输入符号串
  - 第二个 L 表示生成最左推导
  - 1 表示读入一个符号可确定下一步推导

表达式文法是 LL(1) 文法

- **E → T E'**
- **E'→ + T E' | ε**
- **T → F T'**
- **T'→ * F T' | ε**
- **F → (E) | id**

判断如下:

- **E': + 不在 FOLLOW(E') = {), # }**
- **T': * 不在 FOLLOW(T') = { +,), # }**
- **F : (和 id 不同**

文法是否为LL(1)？

G[S]:

- S->AB | bC
- A->b | ε
- B->aD | ε
- C->AD | b
- D->aS | c

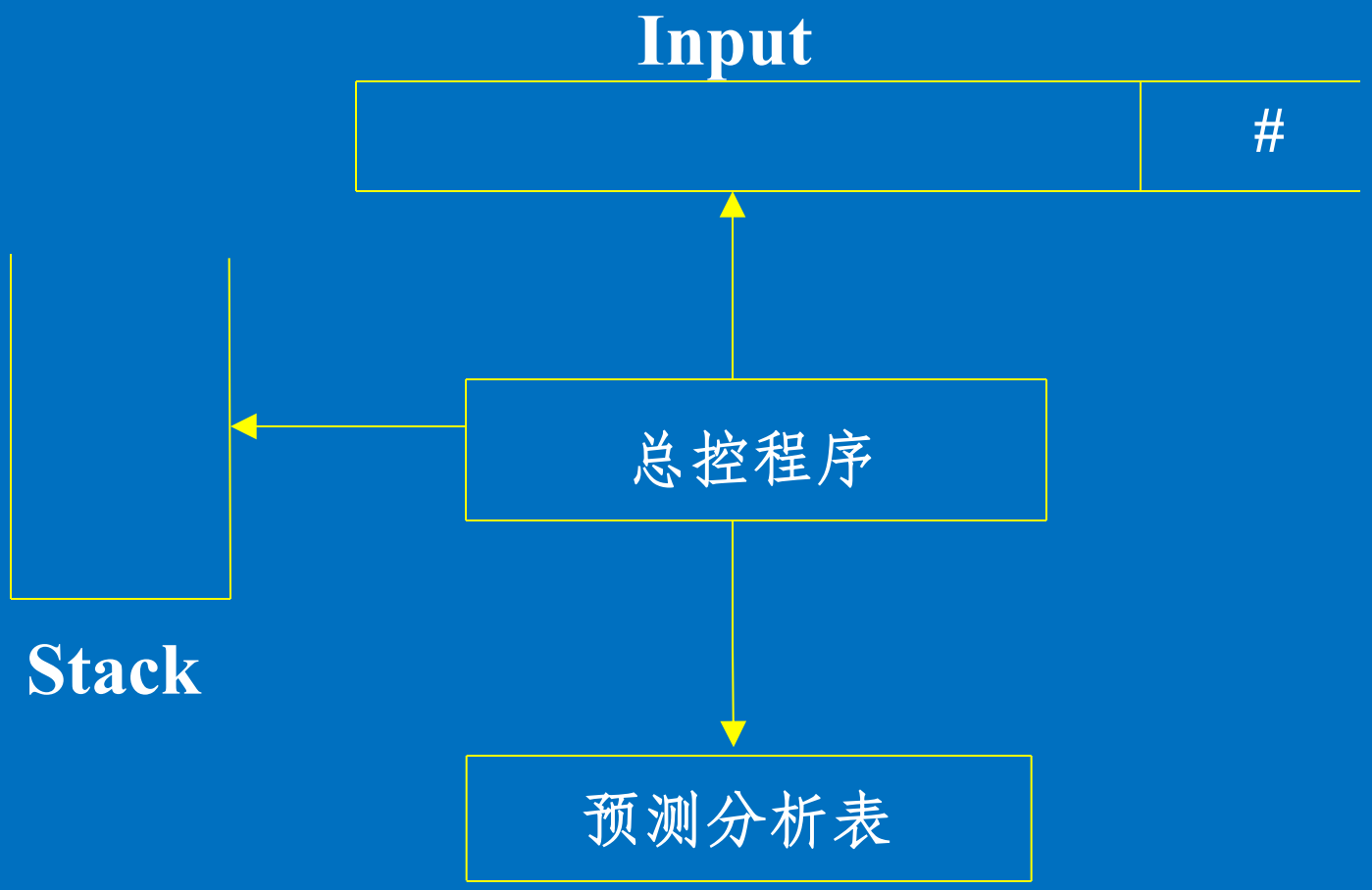
针对LL(1)文法的统一的控制算法

- 通过一个统一的算法实现语法分析
- 根据以下两个因素决定候选产生式
  - 栈顶符号（当前的语法变量）
  - 输入符号

预测分析方法

- 系统维持一个分析表和一个分析栈，根据当前扫描到的符号，选择当前语法变量（处于栈顶）的候选式进行推导——找到相应的输入符号串的最左推导
- 一个通用的控制算法
- 一个统一形式的分析表M
  - 不同语言使用内容不同的分析表
- 一个分析栈
  - #为栈底符号
- 一个输入缓冲区
  - #为输入串结束符

表驱动预测分析程序模型



表达式文法的预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
E	→TE'			→TE'		
E'		→+TE'			→ε	→ε
T	→FT'			→FT'		
T'		→ε	→*FT'		→ε	→ε
F	→id			→(E)		

3）LL(1)分析表的构造

LL(1)分析表的构造算法I

- 1) 对于每一产生式 A→α，执行2)
- 2) 对于 Select(A→α)中的每一终结符a, 将 A→α 填入 M[A, a]
- 3) 将所有无定义的 M[A, b] 标上错误标志

G[E]: (1) E –> TE' (2) E' –> +TE' (3) E' –> ε
(4) T –> FT' (5) T' –> *FT' (6) T' –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE′			–> TE′		
E′						
T						
T′						
F						

Select(E→TE')={(, id}

G[E]: (1) E –> TE' (2) E' –> +TE' (3) E' –> ε
(4) T –> FT' (5) T' –> *FT' (6) T' –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE′			–> TE′		
E′		–> +TE′			–> ε	–> ε
T						
T′						
F						

Select(E'→+TE')={+}

Select(E'→ε)={),#}

G[E]: (1) E –> TE' (2) E' –> +TE' (3) E' –> ε
(4) T –> FT' (5) T' –> *FT' (6) T' –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE′			–> TE′		
E′		–> +TE′			–> ε	–> ε
T	–> FT′			–> FT′		
T′						
F						

Select(T→FT‘)={ (, id }

G[E]: (1) E –> TE' (2) E' –> +TE' (3) E' –> ε
(4) T –> FT' (5) T' –> *FT' (6) T' –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE′			–> TE′		
E′		–> +TE′			–> ε	–> ε
T	–> FT′			–> FT′		
T′		–> ε	–>*FT′		–> ε	–> ε
F	–> id			–>(E)		

Select(T'→ε)={+,),#}

G[E]: (1) E –> TE' (2) E' –> +TE' (3) E' –> ε
(4) T –> FT' (5) T' –> *FT' (6) T' –> ε
(7) F –> (E) (8) F –> id

	id	+	*	(	)	#
E	–> TE′			–> TE′		
E′		–>+TE′			–> ε	–> ε
T	–> FT′			–> FT′		
T′		–> ε	–>*FT′		–> ε	–> ε
F	–> id			–>(E)		

表达式文法的预测分析表

非终结符	输入符号					
	id	+	*	(	)	#
E	→TE'			→TE'		
E'		→+TE'			→ε	→ε
T	→FT'			→FT'		
T'		→ε	→*FT'		→ε	→ε
F	→id			→(E)		

预测分析表构造算法II

1.对文法G的每个产生式A→α执行第二步和第三步；

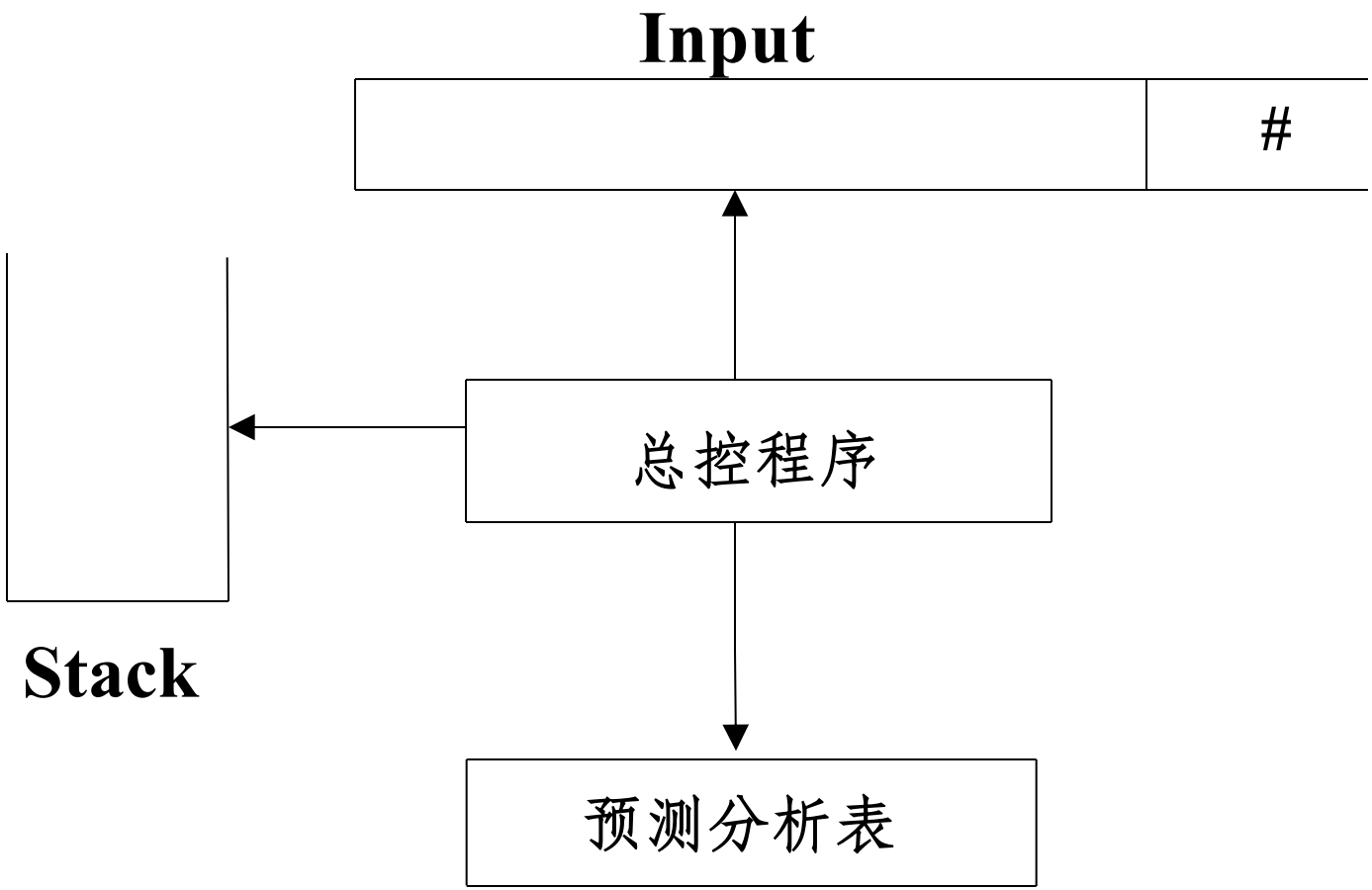
2.对每个终结符a∈FIRST(α)，把A→α加至M[A,a]中；

3.若ε∈ FIRST(α)，则对任何b∈FOLLOW(A)把A→α加至M[A,b]中；

4.把所有无定义的M[A,a]标上“出错标志”。

4) LL(1)分析法

表驱动预测分析程序模型



系统执行及特点

- 系统启动时，输入指针指向输入串的第一个字符，分析栈中存放着栈底符号#和文法的开始符号
- 根据栈顶符号A和读入的符号a，查看分析表M，以决定相应的动作
- 优点：
  - 1）效率高
  - 2）便于维护、自动生成
- 关键：
  - 分析表M的构造

LL(1)通用控制算法

repeat

X=当前栈顶符号;a=当前输入符号;

if $X \in V_T \cup \{\#\}$ *then*

if X=a *then*

if X≠# *then* {将X弹出,且前移输入指针}

else error

else

if $M[X,a]=Y_1Y_2\ldots Y_k$ *then*

{将X弹出;依次将$Y_k\ldots Y_2Y_1$压入栈;

输出产生式$X \rightarrow Y_1Y_2\ldots Y_k$}

else error

until X=#

执行例：分析 id+id*id

栈	输入缓冲区	输出
#E	id+id*id#	
#E'T	id+id*id#	E→TE'
#E'T'F	id+id*id#	T→FT'
#E'T'id	id+id*id#	F→id
#E'T'	+id*id#	
#E'	+id*id#	T'→ε
#E'T+	+id*id#	E'→+TE'
#E'T	id*id#	

#E'T	id*id#	
#E'T'F	id*id#	T→FT'
#E'T'id	id*id#	F→id
#E'T'	*id#	
#E'T'F*	*id#	T'→*FT'
#E'T'F	id#	
#E'T'id	id#	F→id
#E'T'	#	
#E'	#	T'→ε
#	#	E'→ε

输出的产生式序列形成了最左推导对应的分析树

- **#i*(i+i)#**

栈	输入
#E	i*(i+i)#

LL(1)分析中的一种错误处理办法

发现错误：

1. 栈顶的终结符与当前输入符不匹配
2. 非终结符A于栈顶，面临的输入符为a，但分析表M的M[A,a]为空

“应急”恢复策略：

策略：跳过输入串中的一些符号直至遇到“同步符号”为止

同步符号的选择：

1. 把FOLLOW(A)中的所有符号作为A的同步符号，跳过输入串中的一些符号直至遇到这些“同步符号”，把A从栈中弹出，可使分析继续
2. 把FIRST(A)中的符号加到A的同步符号集，当FIRST(A)中的符号在输入中出现时，可根据A恢复分析

LL(1)预测分析法(小结)

- 消除左递归、提取左因子
- 求每个候选式的FIRST集和非终结符的FOLLOW集
- 检查是否为 LL(1) 文法
  - 若不是 LL(1),说明文法复杂性超过LL(1) 的分析能力
- 构造预测分析表
- 实现预测分析器

LL(1)预测分析法(小结)

- 存在问题
  - 对于某些语言，难以用 **LL(1)** 文法来描述
  - 消除左递归和提取左因子影响文法的可读性，造成语义处理的困难

Review

- **LL(1) grammar**
- **Recursive-descent**
- **Table-driven LL(1) parsing**

Predictive parser and LL(1) grammar

Predictive parser is a non-backtracking top-down parser. A predictive parser is characterized by its ability to choose the production to apply solely on the basis of the next input symbol and the current nonterminal being processed.

To enable this, the grammar must take a particular form. We call such a grammar *LL(1)*. The first "L" means we scan the input from left to right; the second "L" means we create a leftmost derivation; and the 1 means one input symbol of lookahead.

Recursive-descent

The first technique for implementing a predictive parser is called *recursive-descent*.

A recursive descent parser consists of several small functions(procedures), one for each nonterminal in the grammar. As we parse a sentence, we call the functions (procedures) that correspond to the left side nonterminal of the productions we are applying. If these productions are recursive, we end up calling the functions recursively.

Table-driven LL(1) parsing

In a recursive-descent parser, the production information is embedded in the individual parse functions for each nonterminal and the run-time execution stack is keeping track of our progress through the parse. There is another method for implementing a predictive parser that uses a table to store that production along with an explicit stack to keep track of where we are in the parse.

How a table-driven predictive parser works

We push the start symbol on the stack and read the first input token. As the parser works through the input, there are the following possibilities for the top stack symbol X and the input token nonterminal a:

1. If X = a and a = end of input (#): parser halts and parse completed successfully
2. If X = a and a != #: successful match, pop X and advance to next input token. This is called a *match* action.
3. If X != a and X is a nonterminal, pop X and consult table at [X,a] to see which production applies, push right side of production on stack. This is called a *predict* action.
4. If none of the preceding cases applies or the table entry from step 3 is blank, there has been a parse error

Constructing the parse table

1. For each production A –> u of the grammar, do steps 2 and 3
2. For each terminal a in First(u), add A –> u to M[A,a]
3. If ε in First(u), (i.e. A is nullable) add A –> u to M[A,b] for each terminal b in Follow(A),

   If ε in First(u), and # is in Follow(A), add A –> u to M[A,#]
4. All undefined entries are errors

LL(1) grammar

A grammar G is LL(1) iff whenever A –> u | v are two distinct productions of G, the following conditions hold:

- for no terminal a do both u and v derive strings beginning with a (i.e. first sets are disjoint)
- at most one of u and v can derive the empty string
- if v =>* ε then v does not derive any string beginning with a terminal in Follow(A)

Error-reporting and recovery

An error is detected in predictive parsing when the terminal on top of the stack does not match the next input symbol or

when nonterminal A is on top of the stack, a is the next input symbol and the parsing table entry M[A,a] is empty.

习题

1、P.99　第1题 3),4)
2、P.100 第3题
3、P.101 第7题 1),3)