第4章 LL(1)文法及其分析程序

4.1 预测分析程序

4.2 LL(1)文法

- FIRST和FOLLOW集定义和计算LL(1)
- 文法定义LL(1)分析程序的生成

4.3 非LL(1)文法的改造

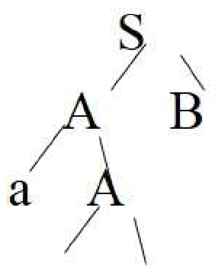
自上而下分析算法

要点：

.由根向下构造语法树

.构造最左推导

.推导出的终结符是否与当前输入符匹配



aaab

S –> AB

A –> aA | ε

B –> b | bB

aaab.

S

⇒AB	S –> AB
⇒aAB	A –> aA
⇒aaAB	A –> aA
⇒aaaAB	A –> aA
⇒aaa ε B	A –> ε
⇒aaab	B –> b

带回溯的自上而下分析

S –> AB
A –> aA | ε
B –> b | bB
a *a a b* **b**.
S

(1) ⇒A...	S –> AB
(2) ⇒*a*A...	A –> aA
(3) ⇒*aa*A...	A –> aA
(4) ⇒*aaa*A...	A –> aA
(5) ⇒*aaa* ε B...	A –> ε
(6) ⇒*aaab*	B –> b

aaabb.
S

(1) ⇒A...	S –> AB
(2) ⇒aA...	A –> aA
(3) ⇒aaA...	A –> aA
(4) ⇒aaaA...	A –> aA
(5) ⇒aaa ε B	A –> ε
(6') ⇒aaa b B	B –> bB
(7) ⇒aaabb	B –> b

预测分析程序Predictive parser无回溯的自顶向下分析程序

特征——根据下一个输入符号为当前要处理的非终结符选择产生式

要求——文法是**LL(1)**的

第一个**L**　从左到右扫描输入串

第二个**L**　生成的是最左推导

1　向前看一个输入符号（**lookahead)**

预测分析程序的实现技术

1 递归下降子程序

2 表驱动分析程序

PL/0语言的EBNF

〈程序〉::=〈分程序〉.

〈分程序〉::=[〈常量说明部分〉][〈变量说明部分〉][〈过程说明部分〉]〈语句〉

〈常量说明部分〉::=CONST〈常量定义部分〉{,〈常量定义〉};

〈变量说明部分〉::=VAR〈标识符〉{,〈标识符〉};

〈过程说明部分〉::= PROCEDURE〈标识符〉〈分程序〉{;〈过程说明部分〉};

〈语句〉::=〈标识符〉:=〈表达式〉|IF〈条件〉then〈语句〉|CALL...|READ...|BEGIN〈语句〉{;〈语句〉} END|WHILE...|...

```
begin(*statement*)
 if sym=ident
 then (*parsing ass.st.*)
   begin
   getsym;
   if sym=becomes
   then getsym
   else error(13);
   expression(fsys);
  end
else
 if sym=readsym
 then
(* parsing read st.*)
```

```
begin
 getsym;
 if sym<>lparen
 then error(34)
 else
 repeat
  getsym;
  if sym <> ident
  then  error(35)
  else getsym
 until sym<>comma;
 if sym<>rparen
 then error(33);
end
```

递归下降子程序

program –> function_list

function_list –> function   function_list | ε

function –> FUNC identifier (parameter_list) statement

```
void ParseFunction()
{
MatchToken(T_FUNC);
ParseIdentifier();
MatchToken(T_LPAREN);
ParseParameterList();
MatchToken(T_RPAREN);
ParseStatement();
}
```

```
void MatchToken(int expected)
{
if (lookahead != expected) {
printf("syntax error \n");
exit(0);
} else // if match, consume
  token and move on
lookahead = yylex();
}
```

例：递归子程序实现表达式的语法分析

表达式的EBNF

〈表达式〉::=[+|-]〈项〉{(+|-)〈项〉}

〈项〉::=〈因子〉{(*|/)〈因子〉}

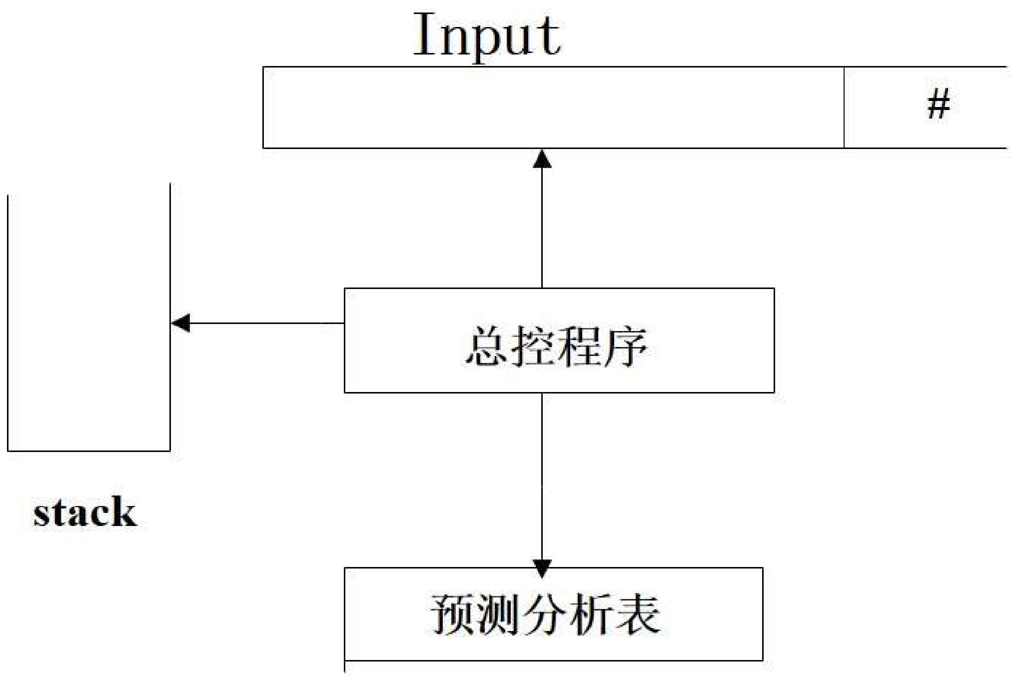
〈因子〉::=<u>ident</u>|<u>number</u>|‘(’〈表达式〉‘)’

```
procedure expr;
  begin
    if sym in [ plus, minus ] then
    begin
      getsym; term;
    end
  else term;
    while sym in [plus, minus] do
    begin
      getsym; term;
    end
  end;
```

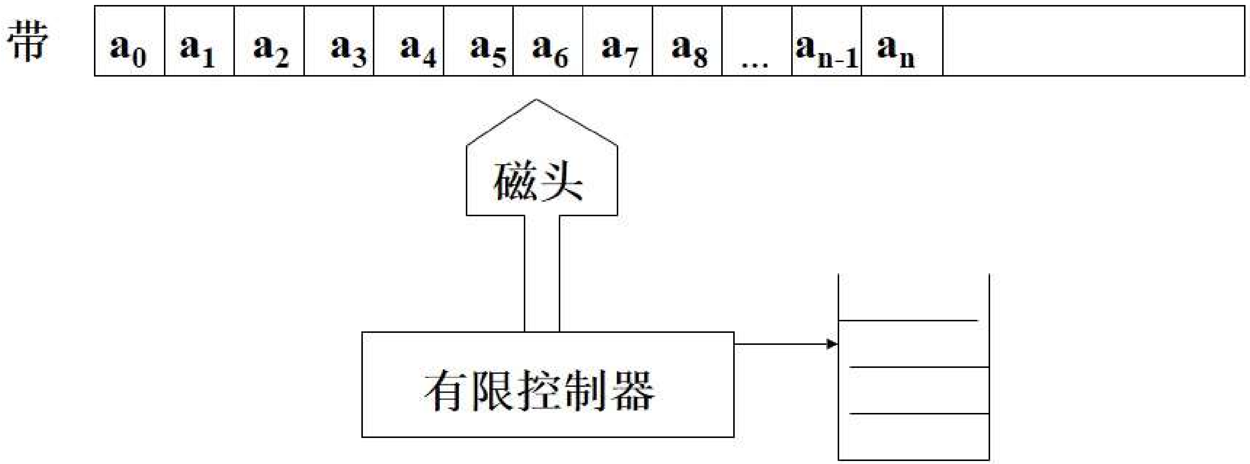
```
Procedure term;
  begin factor;
        while sym in [times,slash] do
        begin getsym;factor end
end;
```

```
Procedure factor;
      begin    if  sym=ident
               then getsym
               else
                  if sym=number
                  then getsym
                else
                if sym=‘(‘
                then begin
                     getsym;
                     expr;
                     if sym=‘)’
                     then getsym
                     else error
                     end
                else error
      end;
```

表驱动予测分析程序模型



识别程序的数学模型下推自动机



上下文无关语言句型分析（识别）程序的数学模型

下推自动机Pda=(K, Σ, f, H, h_0, S, Z)

H:下推栈符号的有穷字母表

h_0 :H中的初始符号

f: $K \times (\Sigma \cup \{\varepsilon\}) \times H \rightarrow K \times H^*$

Pda的一个组态是$K \times \Sigma^* \times H$ 中的一个（k, w, α） k:当前状态，w:余留输入串， α: 栈中符号，最左边的符号在栈顶。

Pda的一次移动用组态表示

终止和接受的条件:

1. 到达输入串结尾时，处在Z中的一个状态

或 2. 某个动作序列导致栈空时

例：Pda P=({A, B, C), {a, b, c), f, {h, i}, i, A, { })

f(A, a, i) = (B, h)　　f(B, a, h) = (B, hh)

f(C, b, h) = (C, ε)　　f(A, c, i) = (A, ε)

f(B, c, h) = (C, h)

接受输入串aacbb的过程

(A,aacbb,i) 读a, pop i, push h, goto B

(B,acbb,h) 读a, pop h, push hh, goto B

(B,cbb,hh) 读c, pop h, push h , goto C

(C,bb,hh) 读b, pop h, push ε , goto C

(C,b,h) 读b ,pop h, push ε , goto C

(C, ε , 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 –> a

G[E]: (1) $E \to \overset{\alpha}{TE'}$ (2) $E' \to +TE'$ (3) $E' \to \varepsilon$

(4) $T \to FT'$ (5) $T' \to *FT'$ (6) $T' \to \varepsilon$

(7) $F \to (E)$ (8) $F \to a$

	a	+	*	(	)	#
E	(1)			(1)		
E′		(2)			(3)	(3)
T	(4)			(4)		
T′		(6)	(5)		(6)	(6)
F	(8)			(7)		

分析算法

```
BEGIN
   首先把'#'然后把文法开始符号推入栈；把第一个输入符
  号读进b；    FLAG：=TRUE；
WHILE FLAG  DO
         BEGIN
     把栈顶符号上托出去并放在X中；
    IF X ∈ Vt   THEN   IF X=b   THEN
                                  把下一个输入符号读进a
                                 ELSE ERROR
       ELSE   IF X='#' THEN
                            IF X=b   THEN   FLAG:=FALSE
                            ELSE ERROR
     ELSE IF M[X,b]={X –> X1X2..XK}
        THEN    把XK, X K-1,..,X1一一推进栈
        ELSE   ERROR
   END OF WHILE;
STOP/*分析成功，过程完毕*/
END
```

分析输入串#a+a#

栈内容	栈顶符号	当前输入	余留串	M[X,b]
1 #E	E	a	+a#	E –> TE’
2 #E’T	T	a	+a#	T –> FT’
3 #E’T’F	F	a	+a#	F –> a
4 #E’T’a	a	a	+a#	
5 # E’T’	T’	+	a#	T’ –> ε
6 #E’	E’	+	a#	E’ –> +TE’
7 #E’T+	+	+	a#	
8 # E’T	T	a	#	T –> FT’
9 #E’T’F	F	a	#	F –> a
10 #E’T’a	a	a	#	
11 #E’T’	T’	#		T’ –> ε
12 #E’	E’	#		E ’ –> ε
13 #	#	#		

LL (1) 文法

FIRST集和FOLLOW集的定义

设G=(V_T, V_N, P, S)是上下文无关文法

FIRST（α）=$\{a \mid \alpha \Rightarrow^* a\beta, a \in V_T, \alpha, \beta \in V^*\}$

若$\alpha \Rightarrow^* \varepsilon$ 则规定 $\varepsilon \in$ FRIST（α）

FOLLOW（A）=$\{a \mid S \Rightarrow^* \mu A \beta$ 且 $a \in$ FRIST（β），$\mu \in V^*$，$\beta \in V^+\}$

若$S \Rightarrow^* uA\beta$，且$\beta \Rightarrow^* \varepsilon$，则 $\# \in$ FOLLOW(A)

计算FIRST集

1. 若$X \in V_T$, 则FIRST(X)={X}

2. 若$X \in V_N$, 且有产生式$X \rightarrow a\ldots$， 则把a加入到FIRST(X)中；若$X \rightarrow \varepsilon$也是一条产生式，则把ε也加到FIRST(X)中.

3. 若$X \rightarrow Y\ldots$是一个产生式且$Y \in V_N$, 则把FIRST(Y)中的所有非ε元素都加到FIRST(X)中；若$X \rightarrow Y_1Y_2\ldots Y_K$ 是一个产生式，$Y_1, Y_2, \ldots, Y_{(i-1)}$都是非终结符，而且，对于任何j, $1 \leqslant j \leqslant i-1$, FIRST($Y_j$)都含有ε（即$Y_1..Y_{(i-1)} \Rightarrow^* \varepsilon$），则把FIRST($Y_j$)中的所有非ε元素都加到FIRST(X)中；特别是，若所有的FIRST(Y_j，$j=1, 2, \ldots, K$)均含有ε，则把ε加到FRIST(X)中.

计算FOLLOW集

1. 对于文法的开始符号S, 置#于FOLLOW(S) 中;
2. 若A→ α B β是一个产生式, 则把 FIRST(β)\{ε}加至FOLLOW(B)中;
3. 若A→ α B是一个产生式, 或A→ αBβ是一个产生式而β =>* ε(即ε∈FIRST(β)), 则把FOLLOW（A）加至FOLLOW（B）中.

一个文法G是LL（1）的，当且仅当对于G的每一个非终结符A的任何两个不同产生式$A \rightarrow \alpha \mid \beta$，下面的条件成立：

1．FIRST（α）∩FIRST(β)=ϕ, 也就是α 和β推导不出以同一个终结符a为首的符号串；它们不应该都能推出空字ε.

2．假若$\beta \Rightarrow^* \varepsilon$，那么，

FIRST（α）∩FOLLOW（A）＝ϕ. 也就是，

若$\beta \Rightarrow^* \varepsilon$. 则$\alpha$所能推出的串的首符号不应在FOLLOW(A）中.

.

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 –> i

·各非终结符的FIRST集合如下：

FIRST(E)= { (, i }

FIRST(E′)= {+, ε }

FIRST(T)= { (, i }

FIRST(T′)= {*, ε }

FIRST(F)= { (, i }

·各非终结符的FOLLOW集合为：

FOLLOW(E)= {), # }

FOLLOW(E′)= {), # }

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

FOLLOW(T′)= {+,), #}

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

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 –> a

E' –> +TE' | ε FIRST(+TE')= {+}

FOLLOW(E′)= {), #}

T' –> *FT' | ε FIRST(*FT')= {*}

FOLLOW(T′)= {+,), #}

F –> (E) | a FIRST((E))= {(}

FIRST(a)= {a}

所以G[E]是LL（1）的

予测分析表构造算法

1. 对文法G的每个产生式$A \rightarrow \alpha$执行第二步和第三步；

2. 对每个终结符$a \in FIRST(\alpha)$，把$A \rightarrow \alpha$加至$M[A, a]$中，

3. 若$\varepsilon \in FIRST(\alpha)$，则对任何$b \in FOLLOW(A)$把$A \rightarrow \alpha$加至$M[A, b]$中，

4. 把所有无定义的$M[A, a]$标上"出错标志"。

可以证明，一个文法G的予测分析表不含多重入口，当且仅当该文法是LL(1)的

LL (1) 文法的性质：

LL (1) 文法是无二义的

LL (1) 文法不含左递归

非LL (1) 文法的改造

消除左递归

提左公因子

将产生式 $A \to \alpha\beta \mid \alpha\gamma$ 变换为：

$A \to \alpha B$

$B \to \beta \mid \gamma$

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to i \mid (E)$

FIRST(E) = { (, i }

FIRST(T) = { (, i }

FIRST(F) = { (, i }

消左递归

E –> TE'

E' –> +TE'

E' –> ε

$S \to \text{if } C\ t\ S \mid$

$\quad \text{if } C\ t\ S\ e\ S$

$C \to b$

提左因子

$S \to \text{if } C\ t\ S\ A$

$A \to e\ S \mid \varepsilon$

	First集	Follow集
S	if	#, e
A	e, ε	#, e
C	**b**	**t**

M[A,e]={$A \to e\ S$

$A \to \varepsilon$ }

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

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

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 –> a

	a	+	*	(	)	#
E	(1)			(1)	syn	
E′		(2)			(3)	(3)
T	(4)	syn		(4)		
T′		(6)	(5)		(6)	(6)
F	(8)			(7)		

review---parsing

The syntax analysis phase of a compiler verifies that the sequence of tokens returned from the scanner represent valid sentences in the grammar of the programming language.

There are two major parsing approaches: *top-down* and *bottom-up*.

In top-down parsing, you start with the start symbol and apply the productions until you arrive at the desired string.

In bottom-up parsing, you start with the string and reduce it to the start symbol by applying the productions backwards.

In the top-down parsing,we begin with the start symbol and at each step, expand one of the remaining nonterminals by replacing it with the right side of one its productions.

We repeat until only terminals remain. The top-down parse prints a leftmost derivation of the sentence.

A bottom-up parse works in reverse. We begin with the sentence of terminals and each step applies a production in reverse, replacing a substring that matches the right side with the nonterminal on the left.

We continue until we have substituted our way back to the start symbol. If you read from the bottom to top, the bottom-up parse prints out a rightmost derivation of the sentence.

lookahead symbol. The *lookahead symbol* is the next symbol coming up in the input.

backtracking. Based on the information the parser currently has about the input, a decision is made to go with one particular production. If this choice leads to a dead end, the parser would have to backtrack to that decision point, moving *backwards* through the input, and start again making a different choice and so on until it either found the production that was the appropriate one or ran out of choices.

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

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 $S \Rightarrow^{*} uAv$, where v begins with some terminal, that terminal is in Follow(A).

Computing first

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.

Calculating follow sets. For each nonterminal in the grammar, do the following:

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 –> u Bv where First(v) contains ε (i.e. v is nullable), then everything in Follow(A) is added to Follow(B).

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.

Panic-mode error recovery

Panic-mode error recovery is a simple technique that just bails out of the current construct, looking for a safe symbol at which to restart parsing. The parser just discards input tokens until it finds what is called a *synchronizing* token. The set of synchronizing tokens are those that we believe confirm the end of the invalid statement and allow us to pick up at the next piece of code.

P99 练习4

1. 对文法G［S］

S→a|∧|(T)

T→T,S|S

对文法G进行改写，改写后的文法G’是否LL(1)?

若是，给出它的递归下降分析程序和预测分析表，

并描述对输入串(a,a)#的分析过程。

2. 已知文法G[S]：

S→MH|a

H→LSo|ε

K→dML|ε

L→eHf

M→K|bLM 判断G是否是LL(1)文法，如果是，构造LL(1)分析表。

3. 对于一个文法若消除了左递归，提取了左公共因子后是否一定为LL(1)文法?试对下面文法进行改写，并对改写后的文法进行判断。

(1) A→aABe|a

B→Bb|d

(3) S→Aa|b

A→SB

B→ab