構文解析

LexとＹＡＣＣ

構文解析の一般的なアルゴリズム（ボトムアップ法）

①入力：記号列$w \in \Sigma^*$，文法$G=(V,\Sigma,P,S)$

②w中のβをAに置き換える．

ただし，$A \rightarrow \beta \in P$

③不可能？

最後の置き換えをもとにもどす．

（後戻り）

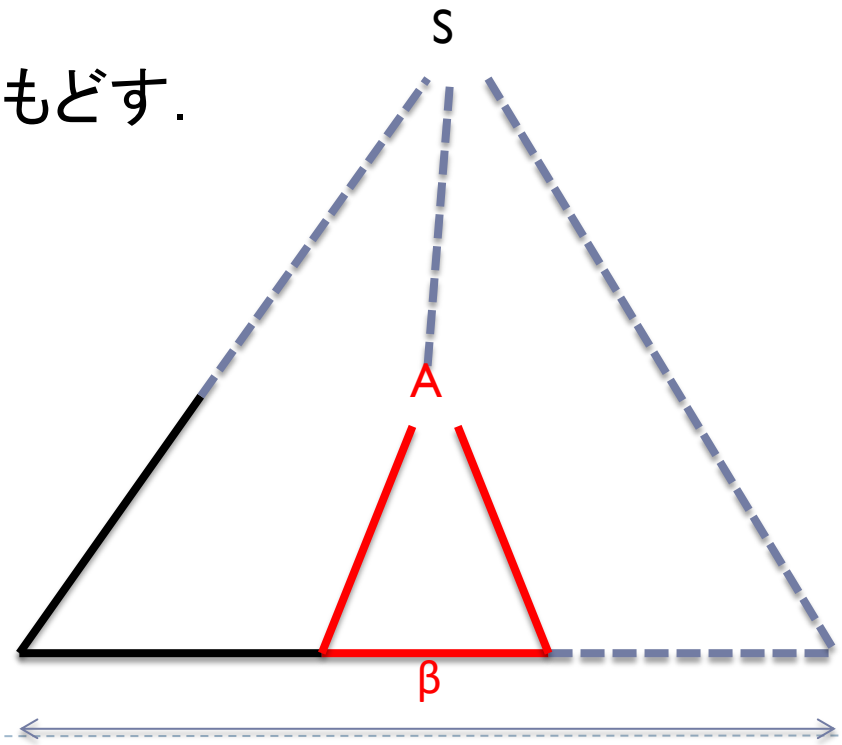
④行き詰まり？

$$w \notin L(G), \text{stop}$$

最終的にSが得られた？

$$w \in L(G), \text{stop}$$

⑤otherwise goto ②



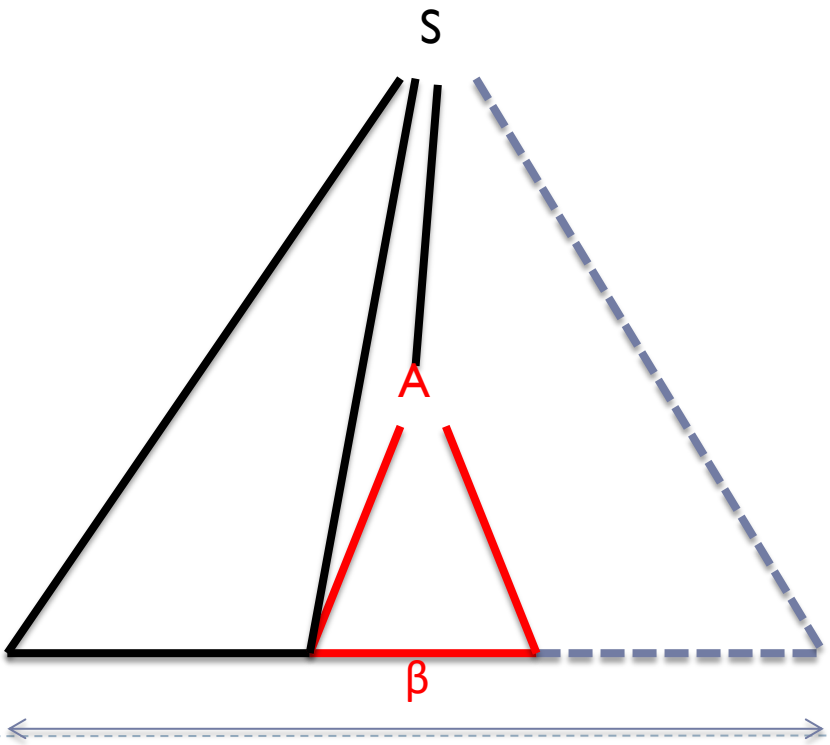
構文解析の一般的なアルゴリズム（トップダウン法）

① 入力：記号列w∈Σ*，文法G=(V,Σ,P,S)
　開始記号Sから始める
② 導出の一番左の非終端記号にルールを適用する
③ 終端記号が一致する？
　②に戻って繰り返す。
④ 終端記号は一致しない。
　②で別のルールを探す。
⑤ 行き詰まり？

$$w \notin L(G), \text{stop}$$

⑥ 全ての終端記号が一致。

$$w \in L(G), \text{stop}$$



係り受け解析（CaboChaの使用例）

```
bigMac:trunk tam$ cabocha
全国に700人いる従業員のほぼ全員がリモートワークという企業がある。
              全国に---D
                 700人-D
                    いる-D
                  従業員の---D
                        ほぼ-D
                        全員が-D
            リモートワークという-D
                           企業が-D
                             ある。
EOS
2014年の創業時からオンラインを駆使する中川さんは「リモートワークを当たり前にする」ことを使命とする。
          2014年の-D
          創業時から---D
          オンラインを-D
                駆使する-D
                中川さんは----------D
           「リモートワークを-D      |
              当たり前にする」-D    |
                          ことを---D
                            使命と-D
                              する。
EOS
```

CaboCha/南瓜(Yet Another Japanese Dependency Structure Analyzer)
http://taku910.github.io/cabocha/

構文解析器の例

- 再帰的下向き構文解析
  - 文法規則を再帰的な手続きに書き換える。
  - 扱える文法のクラスに制限あり
    - LL(1)という文法クラス（授業では扱わない）
- LR(1)構文解析（ボトムアップ構文解析）
  - LALR(1)パーサの原理
  - YACC(Yet Another Compiler Compiler)（計算機による実習）
    - 算術式のスキャナ（構文チェック）

再帰的下向き構文解析

$S \rightarrow aBd$ （$S, B \in V_N$, a, $d \in \Sigma$）

に対して、次の手続きを生成する

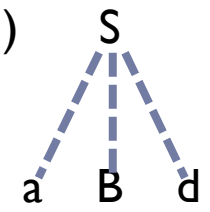
```
void S( );
{
   aを読む;
   B( );
   dを読む;
}
```

$B \rightarrow b|bc$ （$B \in V_N$, b, $c \in \Sigma$）

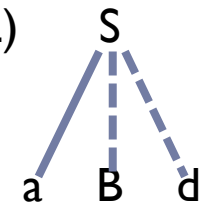
```
void B( );
{
      bを読む;
   または
      bを読む;
      cを読む;
}
```

入力：abcd

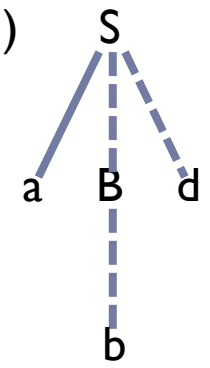
1)



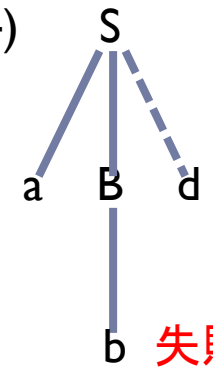
2)



3)



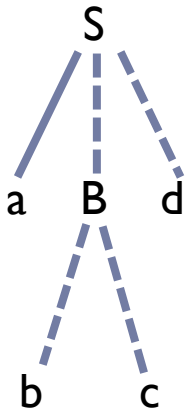
4)



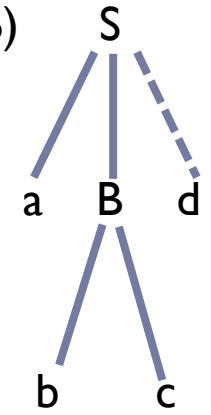
失敗

バックトラック

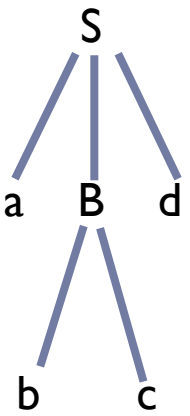
5)



6)

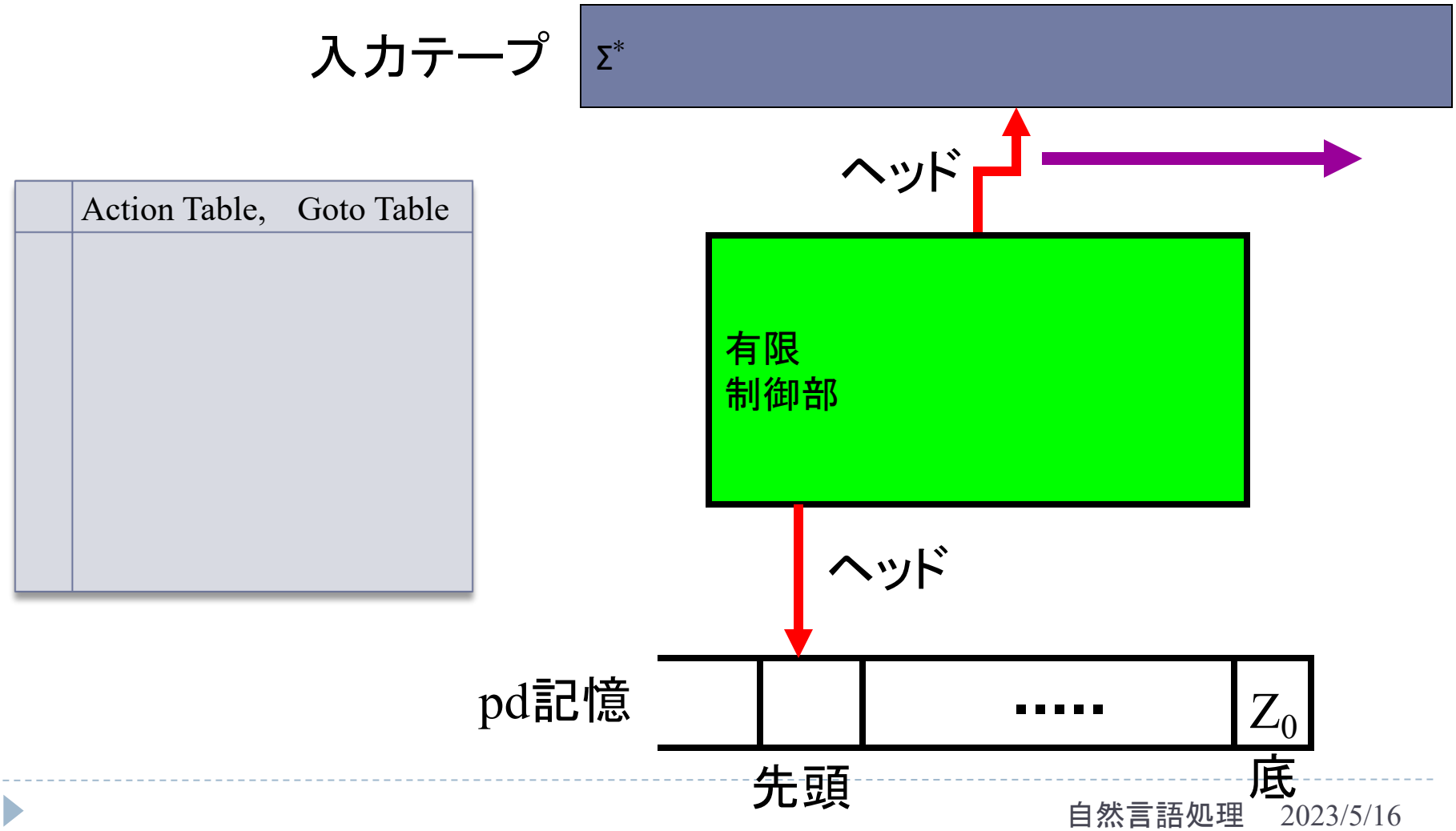


7)



成功

LR(1)構文解析（ボトムアップ構文解析）



LR(1)構文解析器の動作

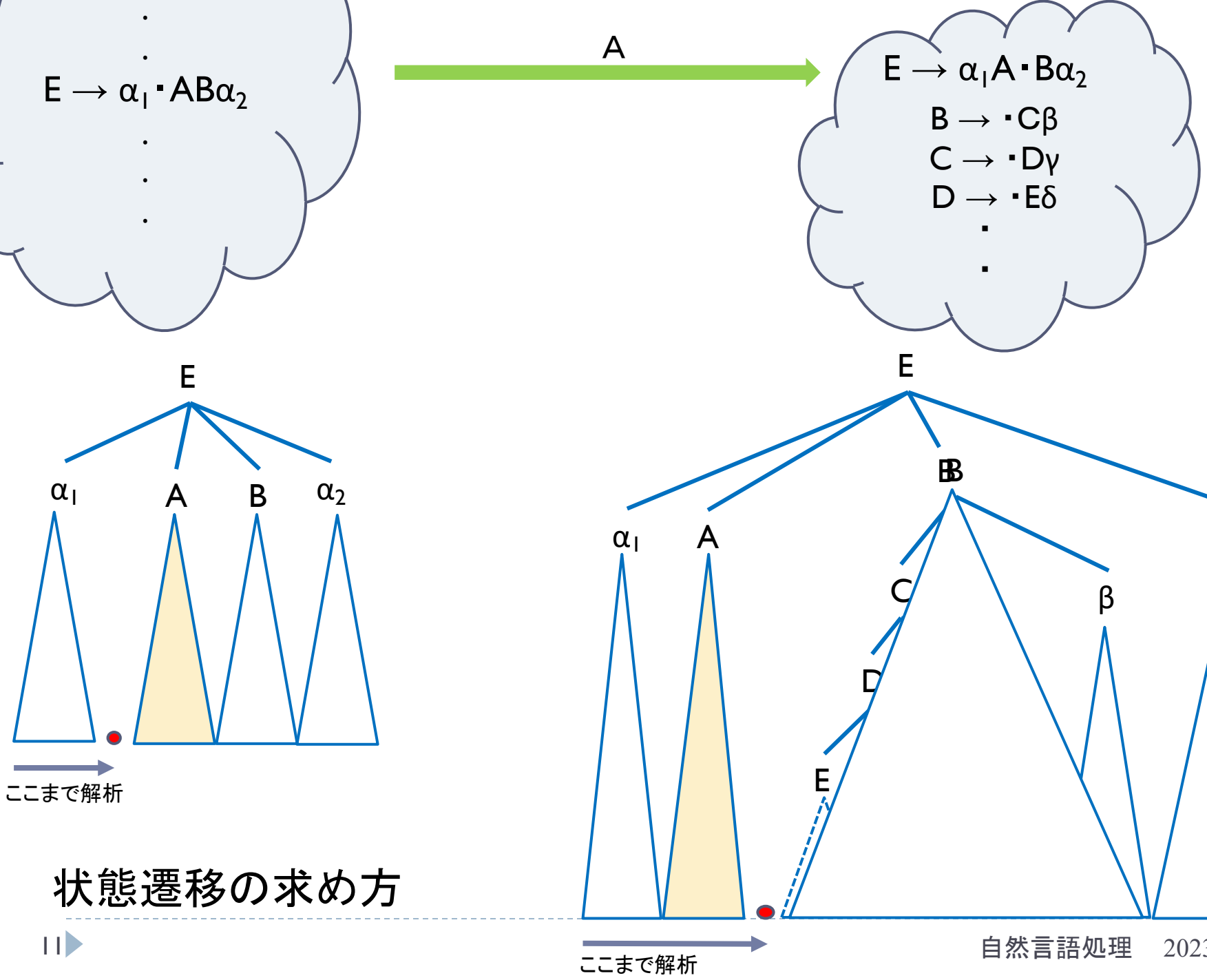
- action(*p*,a) = shift *q*のとき、
  - 入力文字aをスタックにプッシュする
  - 次の状態は*q*

- action(p,a) = reduce $A \rightarrow \beta$のとき、
  - スタック上のβをポップしてAをプッシュする
  - 次の状態は*goto(r,A)=q*により*q*

- Accept
- Error

Action Table, Goto Tableの例

- 文法
  0. E’→E$　　(拡張。元々の開始記号はE)
  1. E→E＋T
  2. E→T
  3. T→T＊F
  4. T→F
  5. F→(E)
  6. F→id


E → α1・ABα2
A
E → α1A・Bα2
B → ・Cβ
C → ・Dγ
D → ・Eδ
E
α1
A
B
α2
ここまで解析
E
α1
A
B
C
D
E
β
α2
ここまで解析

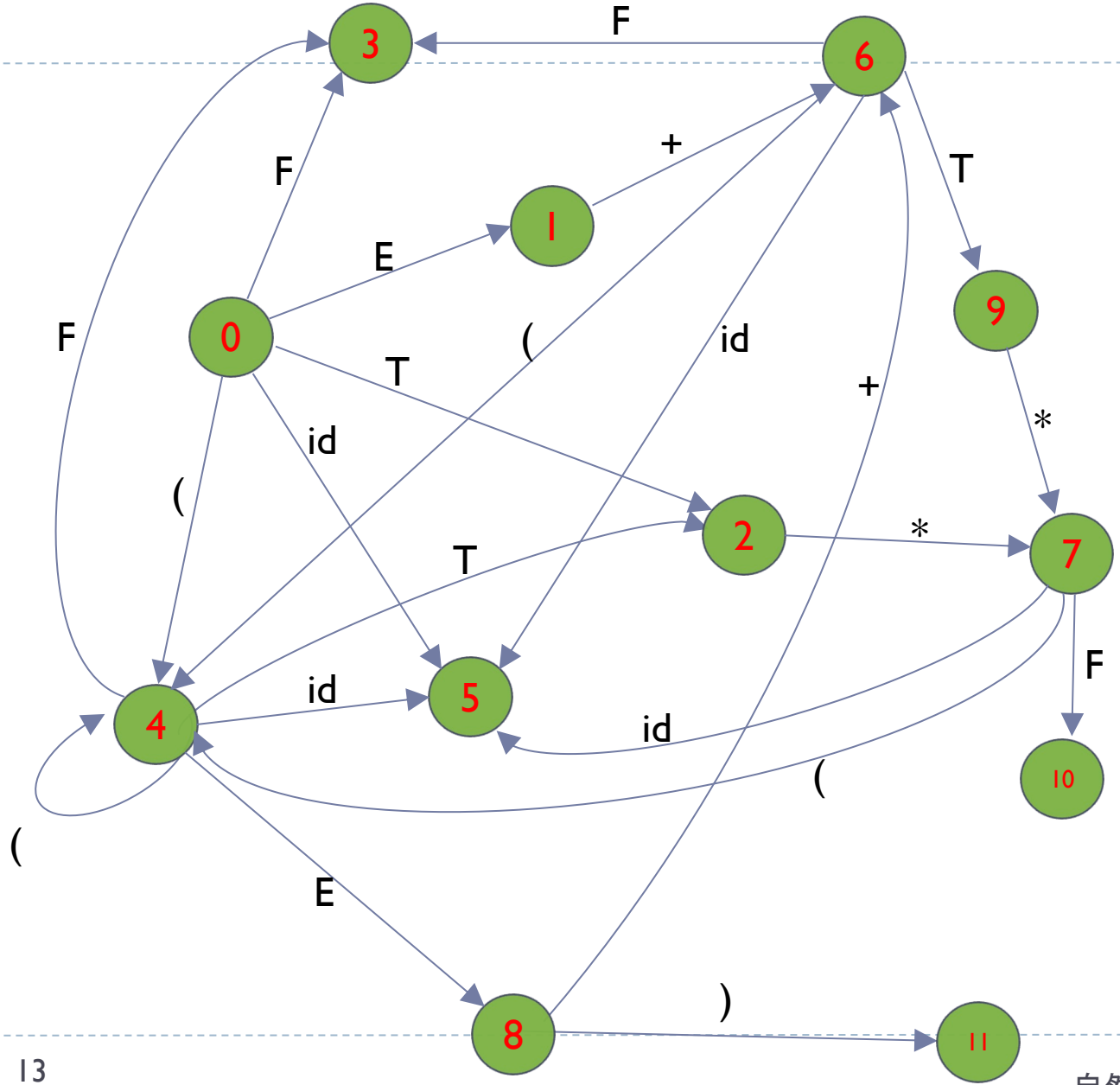
状態遷移の求め方

状態遷移

State	Item	Goto
0	E'→.E	1
	E→.E+T	1
	E→.T	2
	T→.T*F	2
	T→.F	3
	F→.(E)	4
	F→.id	5
1	E'→E.	
	E→E.+T	6
2	E→T.	
	T→T.*F	7
3	T→F.	
4	F→(.E)	8
	E→.E+T	8
	E→.T	2
	T→.T*F	2
	T→.F	3
	F→.(E)	4
	F→.id	5

State	Item	Goto
5	F→id.	
6	E→E+.T	9
	T→.T*F	9
	T→.F	3
	F→.(E)	4
	F→.id	5
7	T→T*.F	10
	F→.(E)	4
	F→.id	5
8	F→(E.)	11
	E→E.+T	6
9	E→E+T.	
	T→T.*F	7
10	T→T*F.	
11	F→(E).	

状態遷移(等価なグラフ表現)



Followセット

- Follow(S): 非終端記号Sから生成される文字列に続く可能性がある終端記号の集合

  - Follow(E) = {$,),+}
  - Follow(T) = {$,),+,*}
  - Follow(F) = {$,),+,*}

Action Table, Goto Tableの作り方

- Action(I,a) = shift s

  …. 状態I、終端記号aの時、状態sへ遷移なら。

- Goto(I,A) = s

  ….状態I、非終端記号Aの時、状態sへ遷移なら。

- Action(I,a) = reduce r

  ….全てのa∈Follow(A)について、状態Iのitemに、規則r:A→β.があるとき。

- Action(I,$) = accept

  ….特に、状態Iのitemに規則S'→S. があるとき。（ただし、Sは開始記号）

state	action						goto		
	id	(	)	+	*	$	E	T	F
0	s5	s4					1	2	3
1				s6		acc			
2			r2	r2	s7	r2			
3			r4	r4	r4	r4			
4	s5	s4					8	2	3
5			r6	r6	r6	r6			
6	s5	s4						9	3
7	s5	s4							10
8			s11	s6					
9			r1	r1	s7	r1			
10			r3	r3	r3	r3			
11			r5	r5	r5	r5			

trace: 入力はid + id

($0Z_0$,id + id $)
↓action(0,id) = s5
($5id0Z_0$,+ id $)
↓action(5,+) = r6
($F0Z_0$,+id $)
↓goto(0,F) = 3
($3F0Z_0$,+id$)
↓action(3,+)=r4
($T0Z_0$,+id$)
↓goto(0,T)=2
($2T0Z_0$,+id$)
↓action(2,+)=r2
($E0Z_0$,+id$)
↓goto(0,E)=1
($1E0Z_0$,+id$)
↓action(1,+)=s6
($6+1E0Z_0$,id$)
↓action(6,id)=s5
($5id6+1E0Z_0$,$)
↓action(5,$)=r6
($F6+1E0Z_0$,$)
↓goto(F,6)=3
($3F6+E0Z_0$,$)
↓action(3,$)=r4
($T6+E0Z_0$,$)
↓goto(6,T)=9
($9T6+E0Z_0$,$)
↓action(9,$)=r1
($E0Z_0$,$)
↓goto(E,0)=1
($1E0Z_0$,$)
↓action(1,$)=acc
Accept

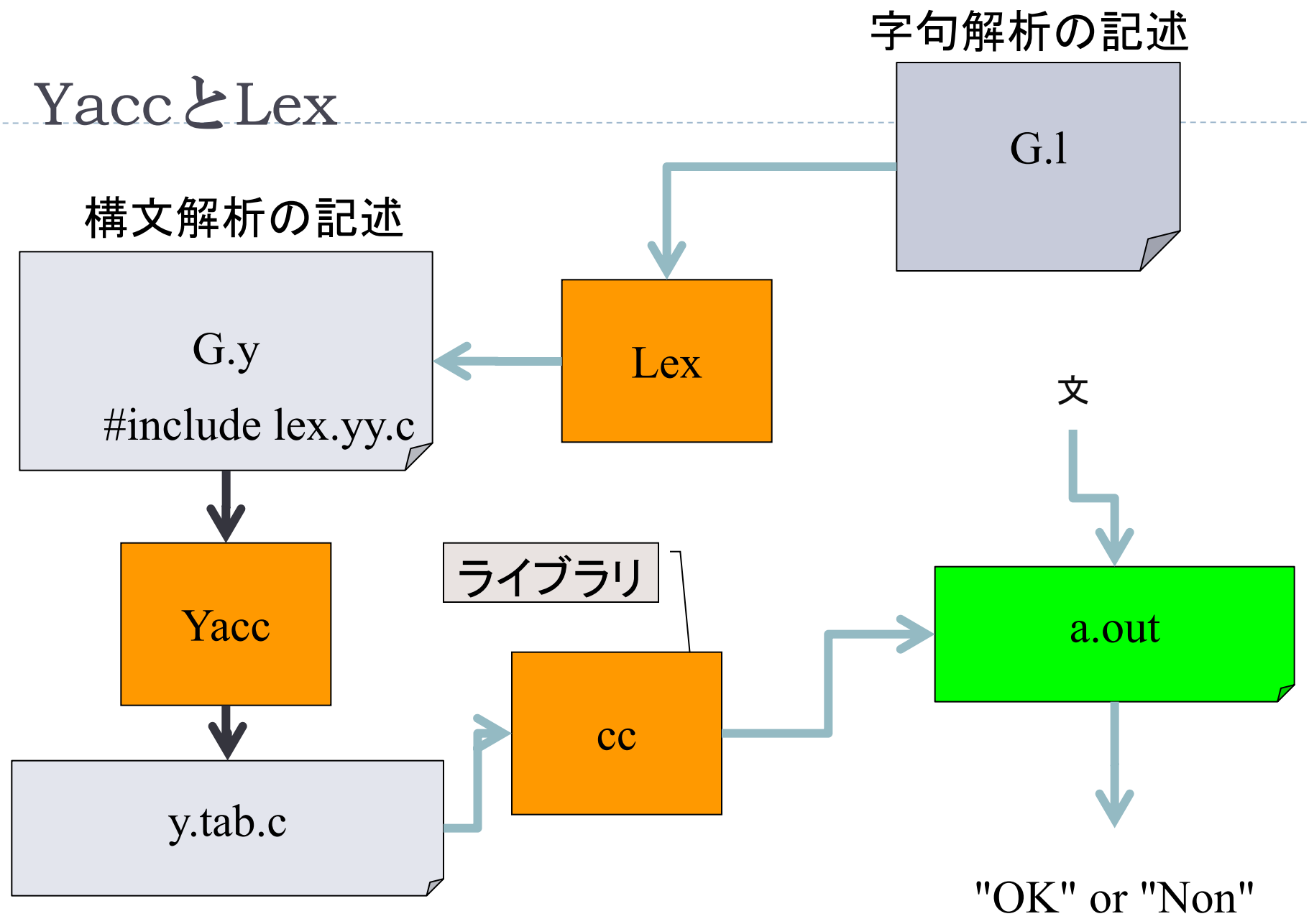
課題

- 入力：id + (id) について、受業で使用したLR（1）パーサの動作をトレースせよ。

コンピュータによる演習

- YACC(Yet Another Compiler Compiler)の原理
  - LALR(1)パーサの原理
- 計算機による実習
  - 算術式のスキャナ

YaccとLex


字句解析の記述
G.l
構文解析の記述
G.y
#include lex.yy.c
Lex
文
Yacc
ライブラリ
a.out
cc
y.tab.c
"OK" or "Non"

YACCの演習

＜演習0：同数のa, bからなる語＞

Ge2 = ({S, A, B, a, b}, {a, b}, Pe2, S)

Pe2 = { S --> a B S, S --> a B, B --> a B B, B --> b,
S --> b A S, S --> b A, A --> b A A, A --> a }

のパーサをつくる。

ちなみに、L(Ge2) = { w | wは、同数のa,bからなる語 }

なお、以下で「y.tab.c」等の特定の名前以外は、ファイルの名前は任意です。対応関係のみ注意してください。

＜語彙解析部の作り方＞

入力から語「a」、「b」を読み、それを終端記号として「tA」「tB」に対応づけるプログラムをlex により生成します。

Ge2.lの中身：

```
----------ここから----------
%%
a               { return tA ;}
b               { return tB ;}
%%
----------ここまで----------
```

command:

```
lex Ge2.l
```

これにより、語彙解析部「lex.yy.c」が生成されます。

＜パーサの作り方＞

文法に対応して、yaccのソースプログラムを作る。

Ge2.yの中身：

```
-----------ここから-----------
%start      S
%token      tA tB
%%
S       : tA B S
        | tA B
        | tB A S
        | tB A
        ;
A       : tB A A
        | tA
        ;
B       : tA B B
        | tB
        ;
%%
#include "lex.yy.c"
-----------ここまで-----------
```

```
A       : tB A A
        | tA
        ;
```

は、A --> tB A A,　A --> tA
をまとめたものです。「|」は、orですね。

command:

```
yacc Ge2.y
```

これにより、構文解析プログラム（パーサ）「y.tab.c」が生成されます。

なお、
yacc -v Ge2.y
とすると、action/goto table「y.output」が生成されますので、覗いてみてください。

＜コンパイル＞

ライブラリを指定して、gccでコンパイルします。

command:

gcc y.tab.c -ly -ll -o Ge2 -Wno-implicit-function-declaration

パーサ「Ge2」が完成します。

・最後「Ge2」はすきな名前でいいです。

・「-ly」、「-ll」はgccのライブラリ指定のためのパラメータで、
それぞれ「liby.so」（yaccのライブラリ）、「libl.so」（lexの
ライブラリ）の使用を意味します。(UNIXのライブラリ指定)

・ -Wno-implicit-function-declaration は、あるオプションです。

＜実行＞

次の2通りがあります。

echo aaabbb | ./Ge2

または、

./Ge2

とすると、入力待ちになるので、キーボードから入力します。

入力の終わりは、Cont-D（コントロール＋D）です。

結果：エラーがなければ何も出力されません。

「aaabbb」をいろいろ変えて試してください。

＜演習1＞

Gh = ({S,T,F,a,+,-,*,/,(,)},{a,+,-,*,/,(,)},Ph,S)

Ph = { S --> S + T, S --> S - T, S --> T, S --> + T, S --> - T,
T --> T * F, T --> F, F --> a, F --> (S) }

を扱います。もちろん言語は、

L(Gh) = { e | eは算術式 }

これを受理するパーサを作成します。

まず、次の文法による簡略版を作成してみましょう。

Gh1 = ({S,T,F,a,+},{a,+},Ph1,S)

Ph1 = { S --> S + T, S --> T, T --> F, F --> a }

＜語彙解析部の作り方＞

Gh1.lの中身：

(1)(2)...は、プログラムには書かないこと。

```
----------ここから----------
numb          [0-9]+                    (1)
%%
{numb}        { return NUMBER; }        (2)
¥+            { return PLUS; }          (3)
([ ]|¥t|¥n)*      { /* nop */ }         (4)
%%
----------ここまで----------
```

(1)は、lexの内部で使うnumbを定義しています。「[0-9]+」は、数字が1個以上続くことを意味します。

(2)終端記号は、NUMBERです。

(3)記号「+」は、終端記号PLUSとします。「¥」はくり返しの「+」と区別するためのエスケープ記号です。

(4) 空白「[ ]」、タブ「¥t」、改行「¥n」は無視「 /* nop */ 」されます。

「｜」は、orです。

command:

lex Gh1.l

＜パーサの作り方＞

Gh1.yの中身：

```
----------ここから----------
%start     S
%token    NUMBER PLUS
%%
S      : S PLUS T
       | T
       ;
T      : F
       ;
F      : NUMBER
       ;
%%
#include "lex.yy.c"
----------ここまで----------
```

「F」は、文法では「F-->a」となっていましたが、数字を扱うために変更されています。

command:

```
    yacc Gh1.y
```

これにより、構文解析プログラム（パーサ）「y.tab.c」が生成されます。

＜コンパイル＞

ライブラリを指定して、gccでコンパイルします。

command:

```
gcc y.tab.c -ly -ll -o Gh1 -Wno-implicit-function-declaration
```

＜実行＞

いろいろありますが、

```
echo "1+2+3+4" | ./Gh1
echo "1+2+a+4" | ./Gh1
...
```

括弧を使う時は、""で式を囲みます。

結果：エラーがなければ何も出力されません。

＜拡張版の注意点＞

（1）「(」、「)」等の記号を終端記号（例えば、それぞれLP,RP）に置き換える

lexの部分。

```
¥(    { return LP ;}
```

（2）同様に、「-」、「*」、「/」の扱い。

課題（やらなくていいです。出来る人だけ）

- 目的：文法Ghのパーサを作る
  - Gh1のパーサを拡張し、文法Ghのパーサを作って下さい。
- 動作の確認
  - ＜実行＞
  - echo "1+2+3+4" | ./Gh
  - echo "1+2+a+4" | ./Gh
  - echo ”(1+2)*4" | ./Gh
  - echo ”((1-2)+4)*7" | ./Gh
  - ...

課題（続き）

- 実行例をスクリーン・キャプチャして提出して下さい。

```
bigMac:大連理工大学 tam$ ./Gh1
6+2-1
syntax error
-bigMac:大連理工大学 tam$ ./Gh1
6+2+1
bigMac:大連理工大学 tam$ echo "((1-2)+4*7"|./Gh1
syntax error
((-bigMac:大連理工大学 tam$ echo "1+2+4+7"|./Gh1
bigMac:大連理工大学 tam$
```