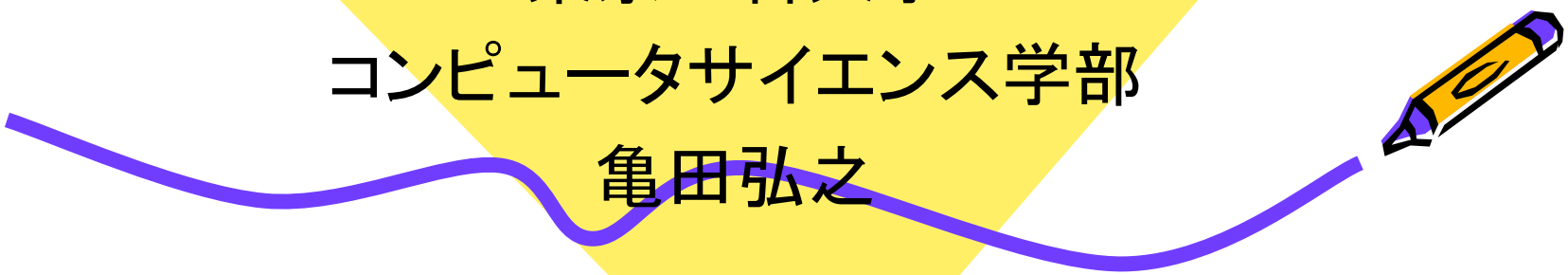
言語プロセッサ2014
-No.5-

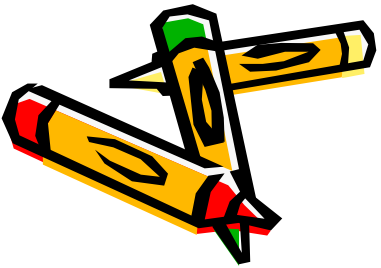
東京工科大学
コンピュータサイエンス学部
亀田弘之



お知らせ（確認）

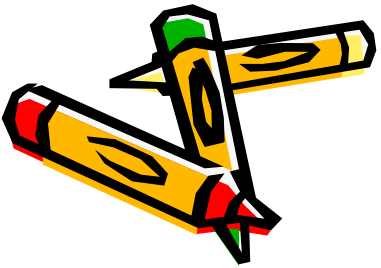
- 平成26年11月17日（月）は休講。
- 平成26年12月20日（土）に補講の予定。

（注）平成26年1月21日（水）も台風補講の予定。



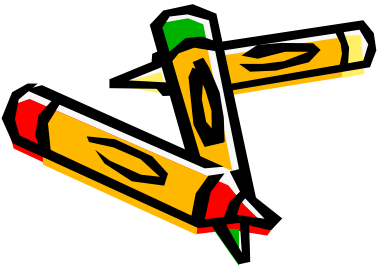
これからの内容

1. 字句解析プログラムの作成方法
   - 手書きの方法
   - Flexを利用する方法
2. 構文解析
   - 解析手法の種類
   - 左再帰とその除去
   - 括りだし
   - FirstとFollow



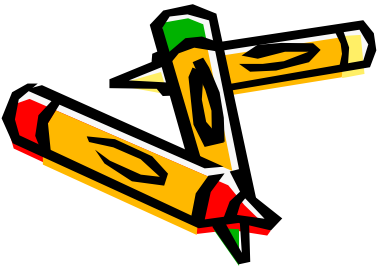
参考資料(発展)

- Intermediate Representations in Imperative Compilers: A Survey, James Stanier and Des Watson, ACM Computing Surveys, Vol.45, No.3, Article 26(27 pages), 2013.

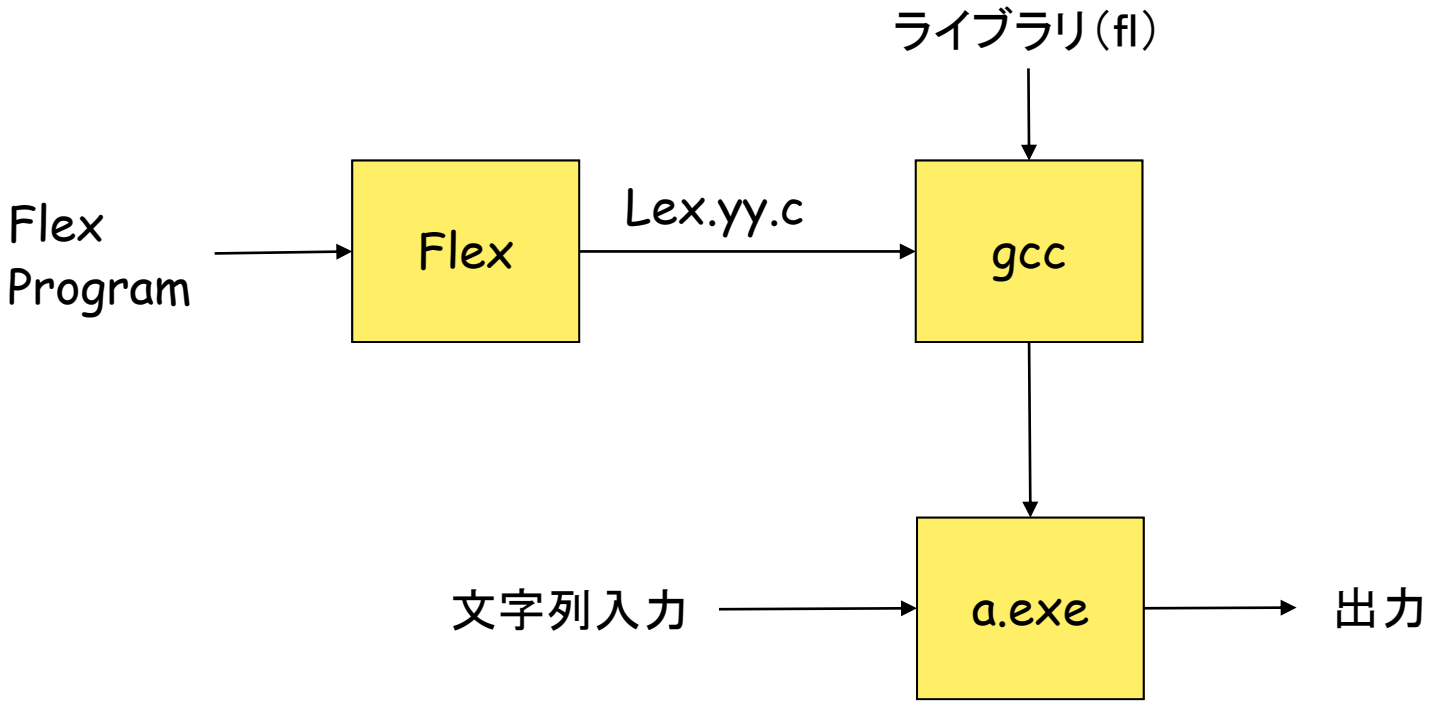


復習課題：
Flexを使ってみよう！

- 自分で過去問に取り組む。
- 自分で新しい課題を見つける。
- その他（自由に）

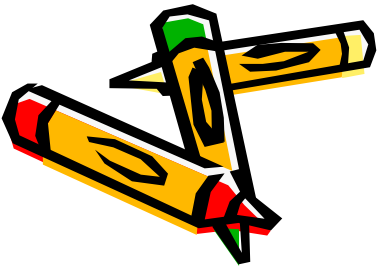


手順



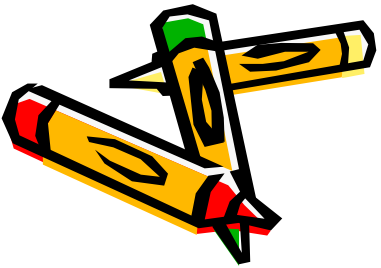
Flexプログラムの記述(1)

```
delim     [ ¥t¥n]
ws        {delim}+
letter    [a-zA-Z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(¥.{digit}+)?(E[+¥-]?{digit}+)?
%%
```



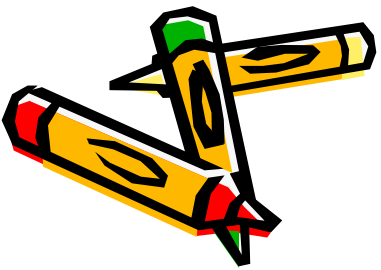
Flexプログラムの記述(2)

```
{ws}      { /* do nothing */ }
If        {return(IF);}
Then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = install_id( ); return(ID);}
{number}  {yylval = install_num();
            return(NUMBER);}
```



Flexプログラムの記述(3)

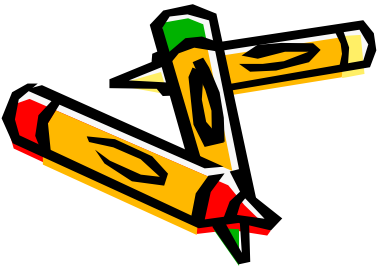
```
"<"     {yylval = LT; return(RELOP);}
"<="    {yylval = LE; return(RELOP);}
"="     {yylval = EQ; return(RELOP);}
"<>"    {yylval = NE; return(RELOP);}
">"     {yylval = GT; return(RELOP);}
">="    {yylval = GE; return(RELOP);}
%%
```



Flexプログラムの記述(4)

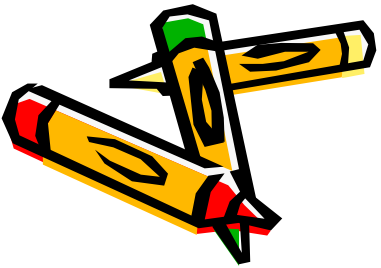
```
install_id( ){
   static int id_ptr=0;
   return(id_ptr); }

install_num( ){
   static int num_ptr=0;
   return(num_ptr); }
```



Flexの復習

- (いくつか実例をやります。)

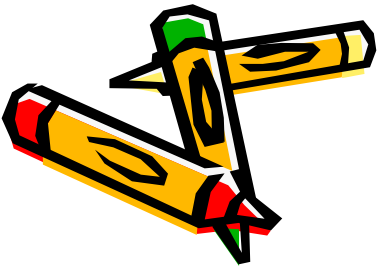


数式をトークンに分解する！

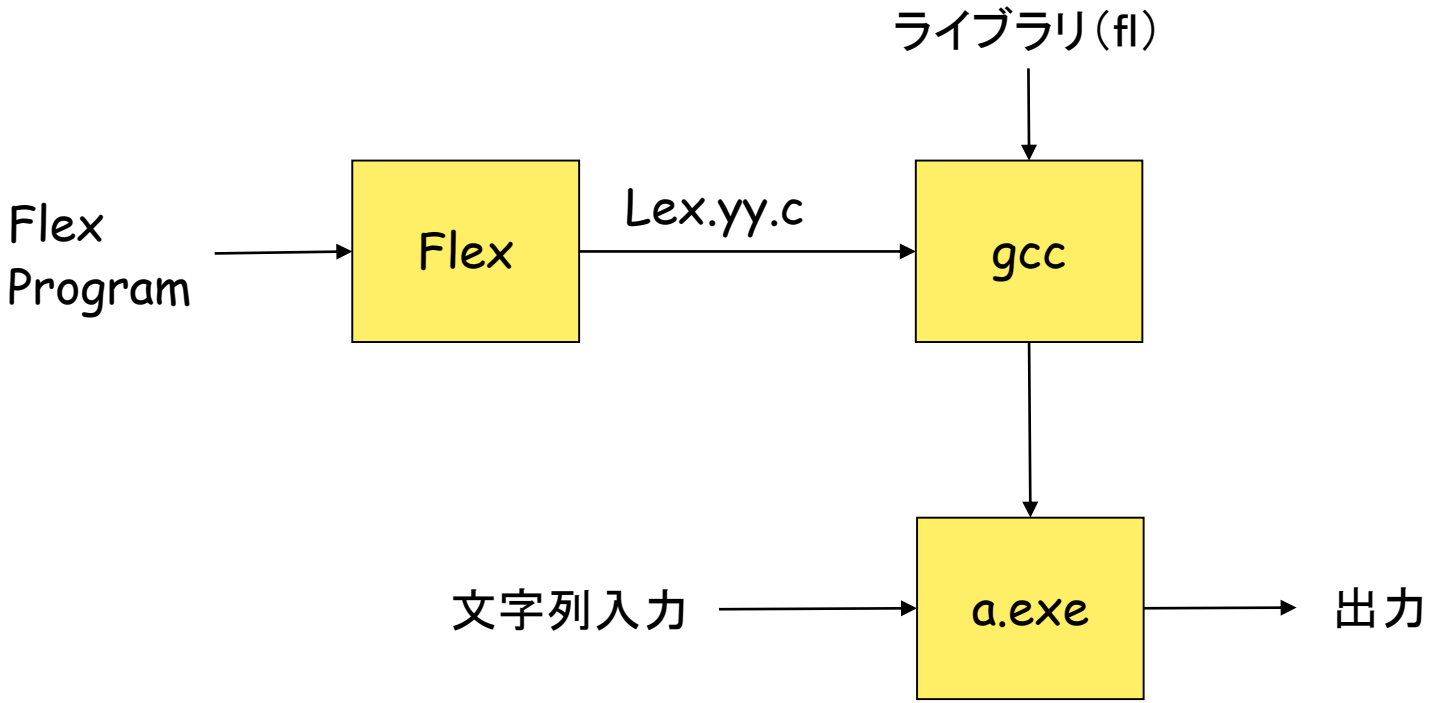
- 練習１
  - 入力: teika + teika * zei
  - 出力: var(teika) op(+) var(teika) op(*) var(zei)
- 練習２
  - 入力: a123 * xyz + 120 * h
  - 出力: var(a123) op(*) var(xyz) op(+) num(120) …
- 練習３
  - 入力: x1 + x2 * (y1 + y2)
  - 出力: var(x1) op(+) var(x2) op(*) lpa(() var(y1) …

手順

1. Flexのプログラムを書く。
2. Flexのプログラムをflexにかける。
3. 出力ファイルlex.yy.cをgccでコンパイルする。この際，ライブラリーを忘れずに！
4. 出力a.exeを実行する。
5. さまざまな文字列を入力する。



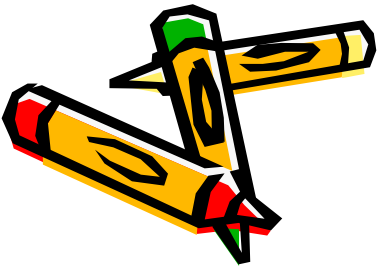
手順



実際の手順

```
C:¥> flex  sample01.l
C:¥> gcc lex.yy.c –lfl
C:¥> a.exe
```

それでは、実際にやってみよう。

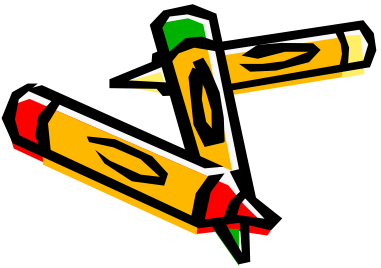


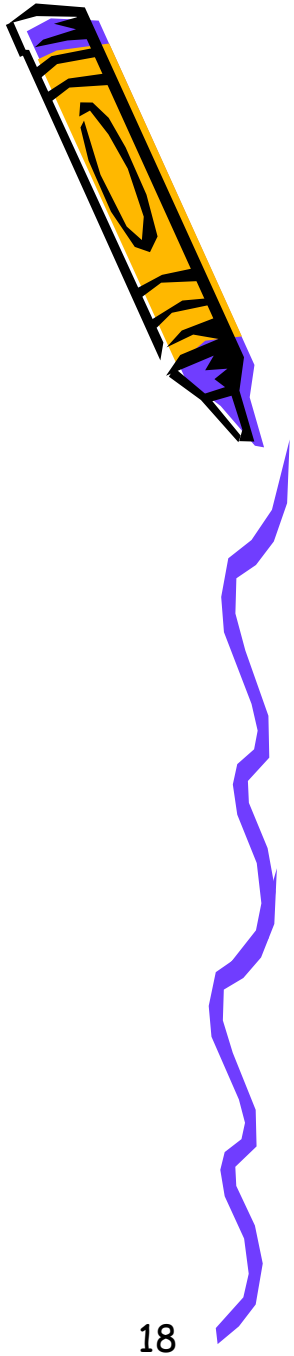
数式をトークンに分解する！（再）

- 練習1
  - 入力: teika + teika * zei
  - 出力: var(teika) op(+) var(teika) op(*) var(zei)
- 練習2
  - 入力: a123 * xyz + 120 * h
  - 出力: var(a123) op(*) var(xyz) op(+) num(120) …
- 練習3
  - 入力: x1 + x2 * (y1 + y2)
  - 出力: var(x1) op(+) var(x2) op(*) lpa(() var(y1) …

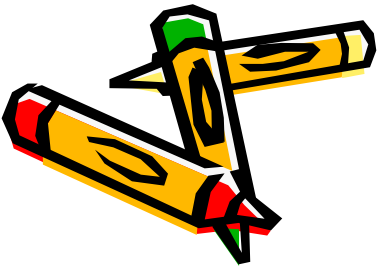
字句解析から構文解析へ

以上で、字句解析（入門）は一応終わり。
字句解析の次の処理は、構文解析でしたね。



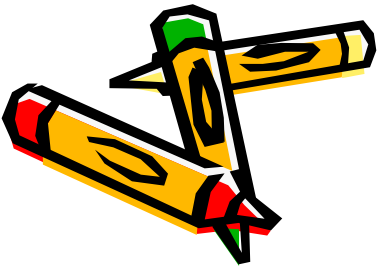


構文解析編



キーワード(構文解析)

- 上向き解析/下向き解析
  (bottom up & top down)
- Backtracking
- 括りだし(factoring)
- 左再帰性
- First集合/Follow集合　など



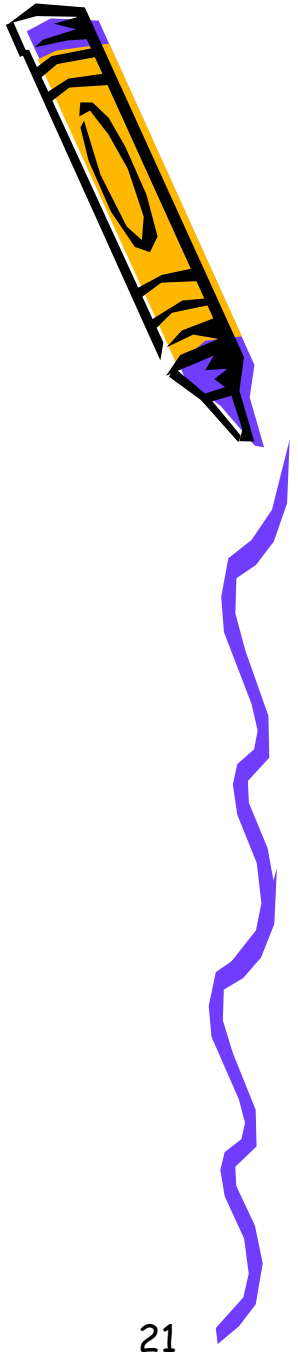
いろいろな構文解析法

- 構文解析手法はとてもよく研究されており、様々な手法が知られている。
- 例えば、
  - Early法
  - Chart法　etc.

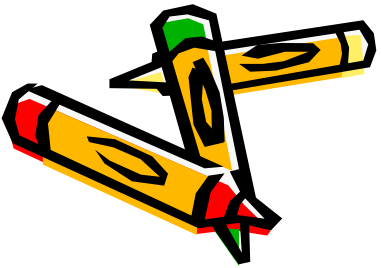
  （余裕のある人は是非勉強してください）

プチお知らせ

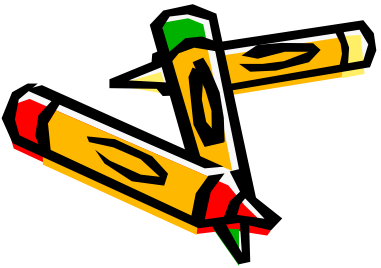
自然言語処理（CS学部3年後期開講科目，担当教員：亀田）



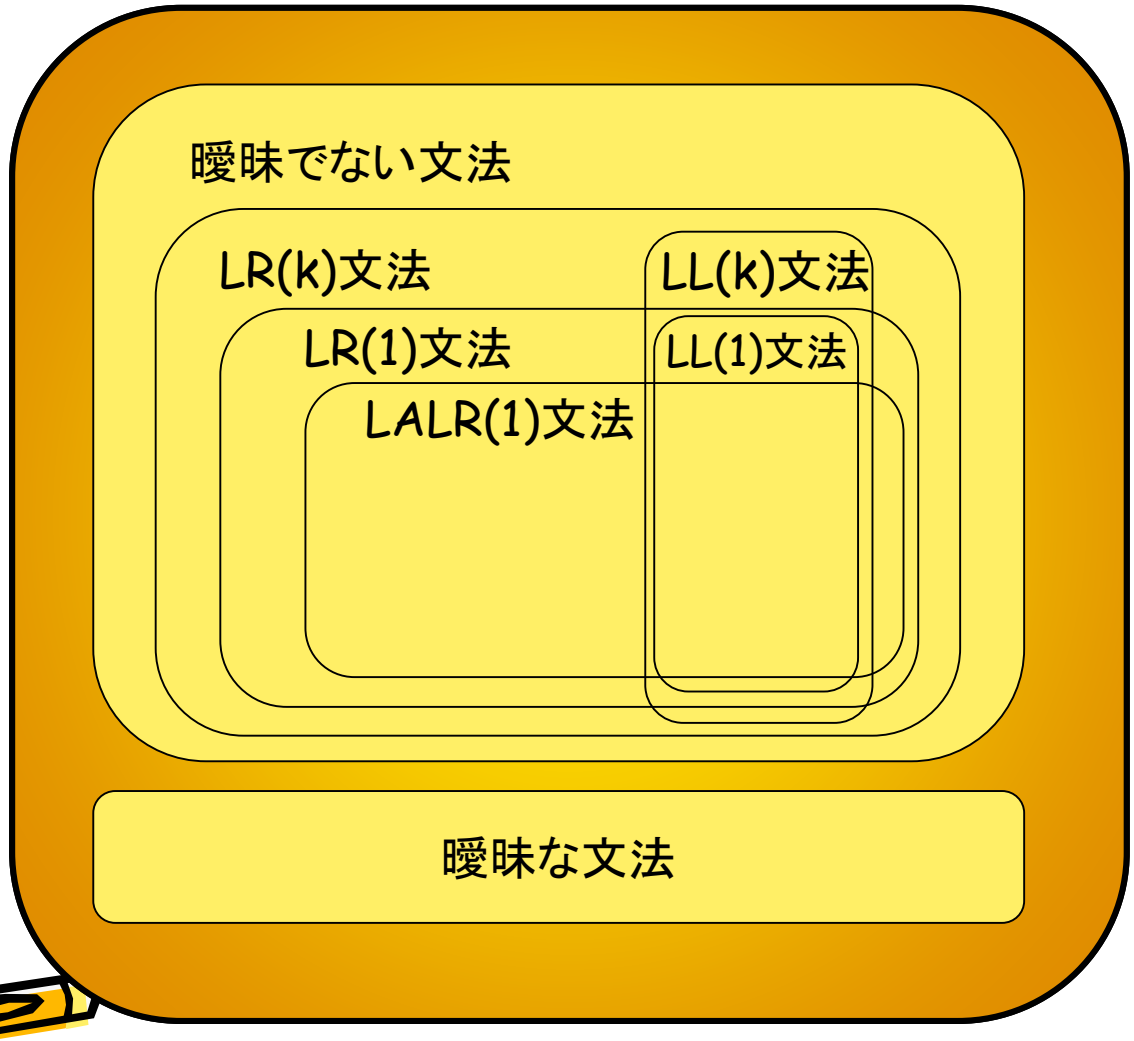
- 処理対象の文法の性質を利用して、より効率的な手法がいろいろと提案されている。



- 文脈自由文法
  - Early法・Chart法　など
- 通常のプログラミング言語は、文脈自由文法ではないが、その構成要素の多くは文脈自由文法で記述可能！
- 文法の制限の仕方にもいろいろある。



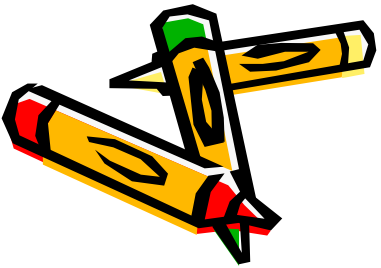
(参考)文脈自由文法の種類



LR文法とLL文法(1)

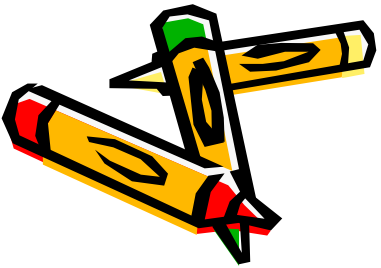
- LR文法に対する構文解析法(LR構文解析法)
  → bottom up 型
- LL文法に対する構文解析法(LL構文解析法)
  → top down 型

(教科書76-77ページ参照)



LR文法とLL文法(2)

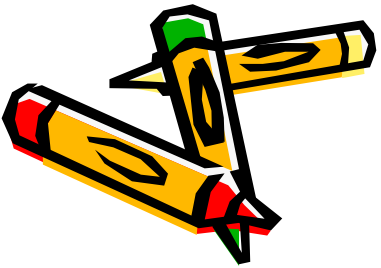
- LR文法に対する構文解析法(LR構文解析法)
  → bottom up 型 ＜＝　自動生成向き(Bison)
- LL文法に対する構文解析法(LL構文解析法)
  → top down 型　＜＝　手作業可能



(教科書76-77ページ参照)

LL(k)文法

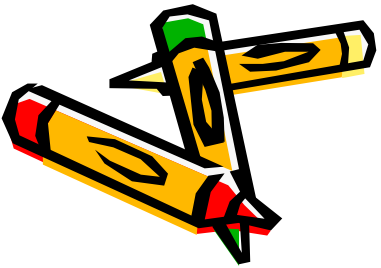
- 構文解析は、文法規則(書き換え規則)を適用しつつ進行。
- 適用すべき規則は、一般には複数個存在。
  → backtrack発生
  → 効率低下(回避すべき！)
- k文字先読で適用すべき規則が決定される文法がある！(LL(k)文法と呼ぶ)



Backtrackなし！

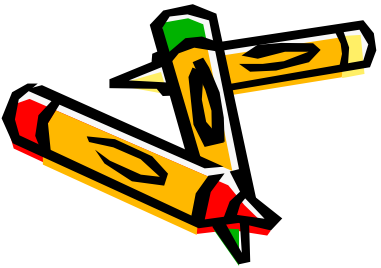
これはすごい！

以下、LL(1)を取り扱います



実例で考えよう！

1. 括りだし
2. 左再帰の回避

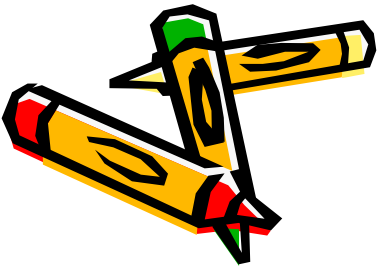


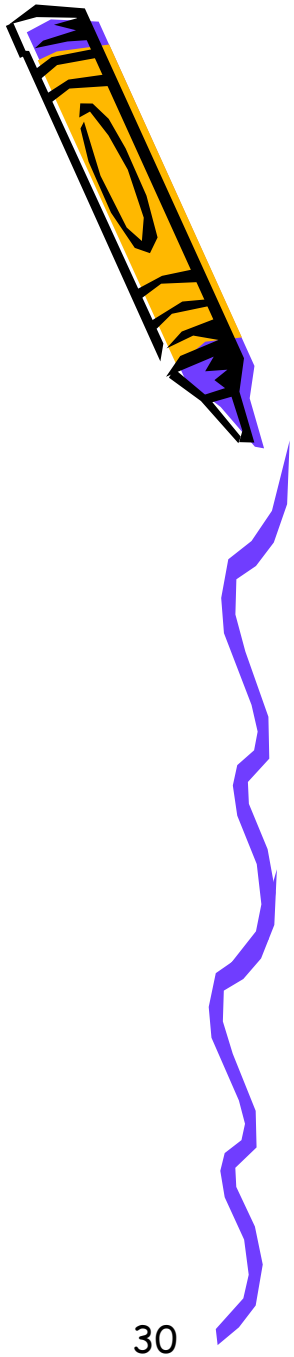
1. 括りだし

- 文法

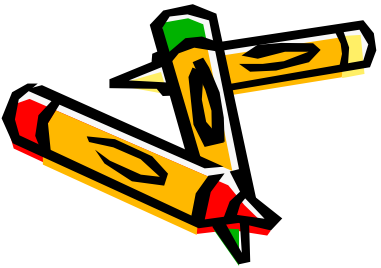
$$\begin{cases} S \rightarrow aBd \\ B \rightarrow b \mid bc \end{cases}$$

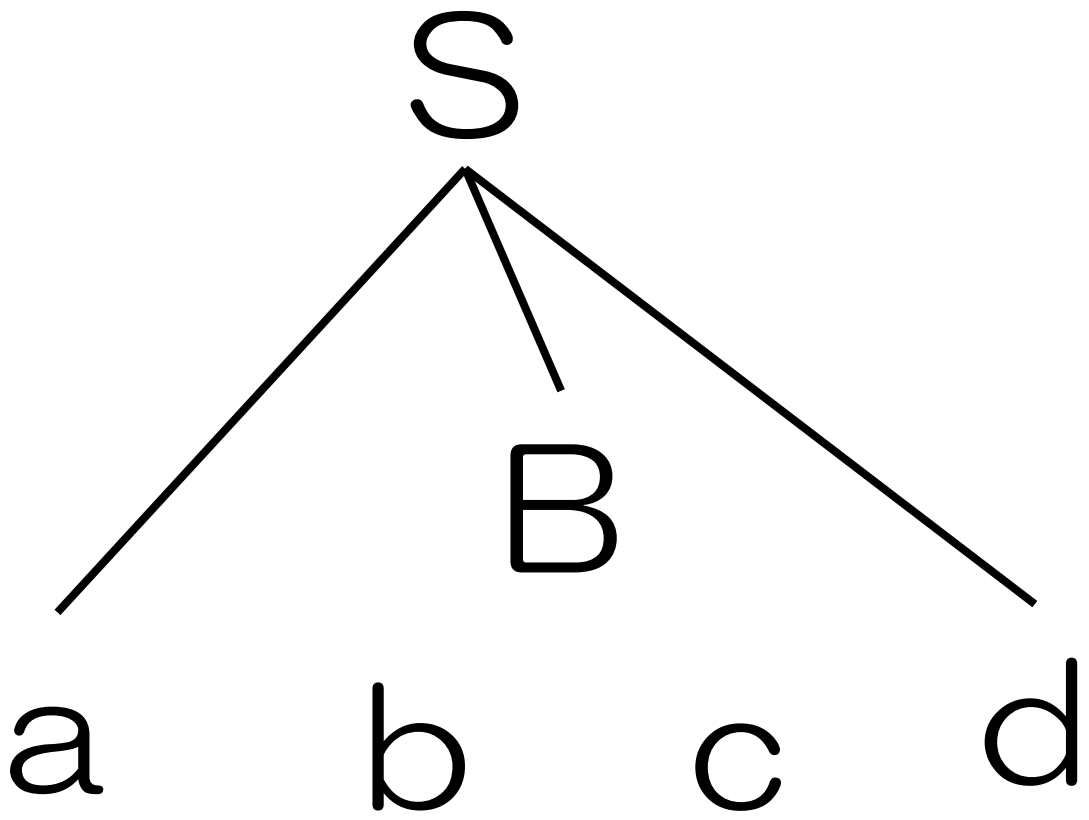
- 入力: abcd

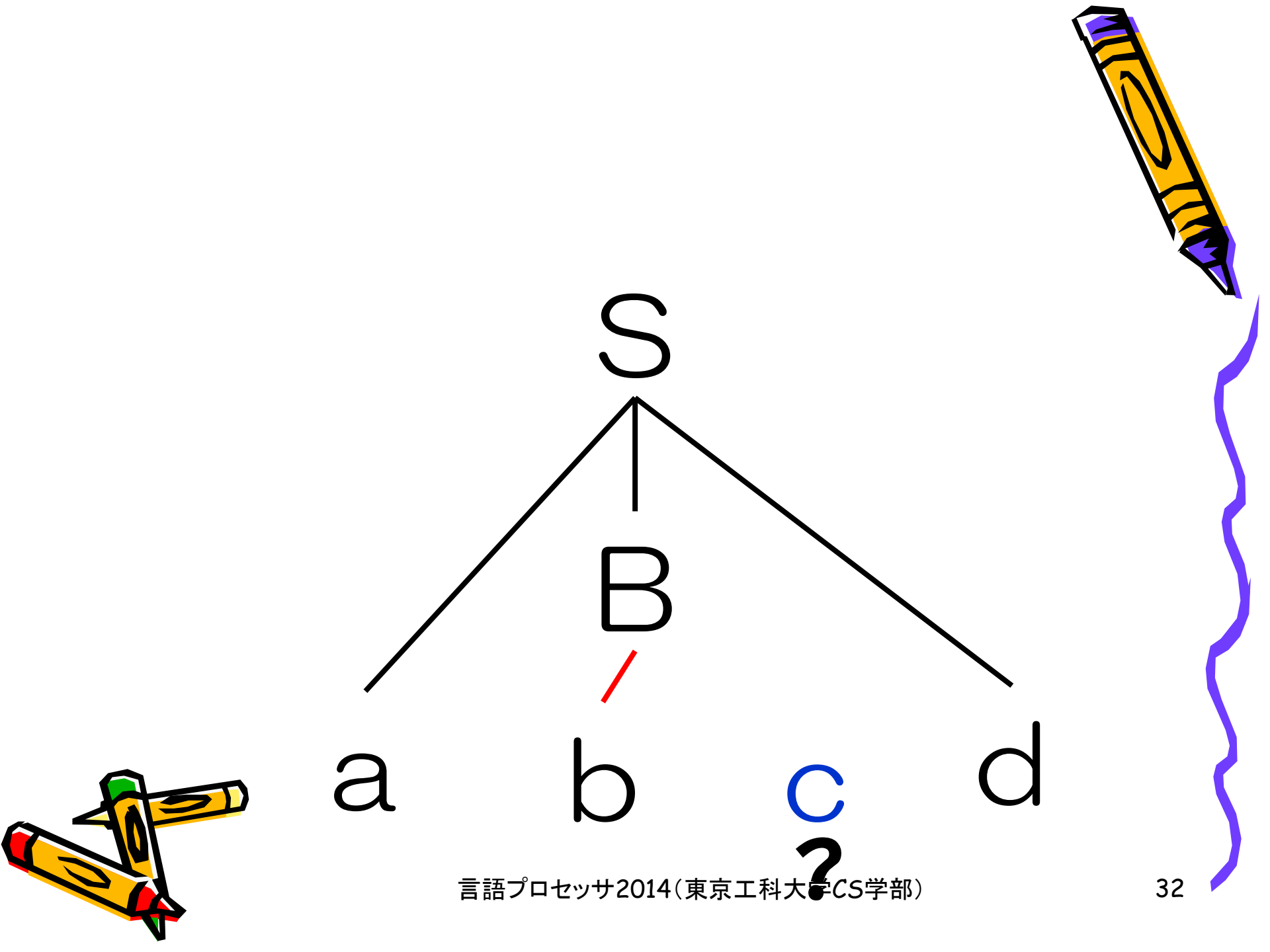




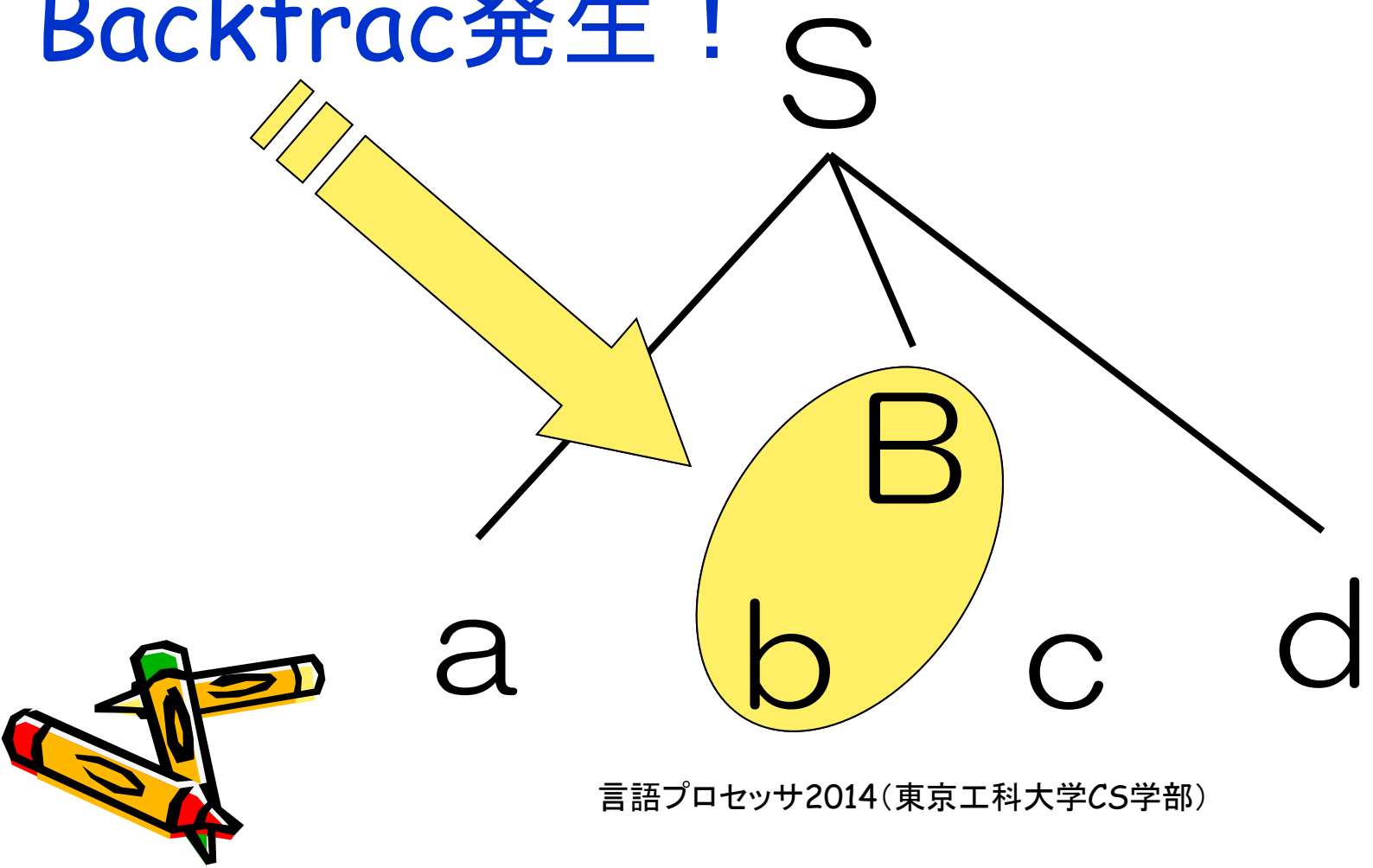
a b c d

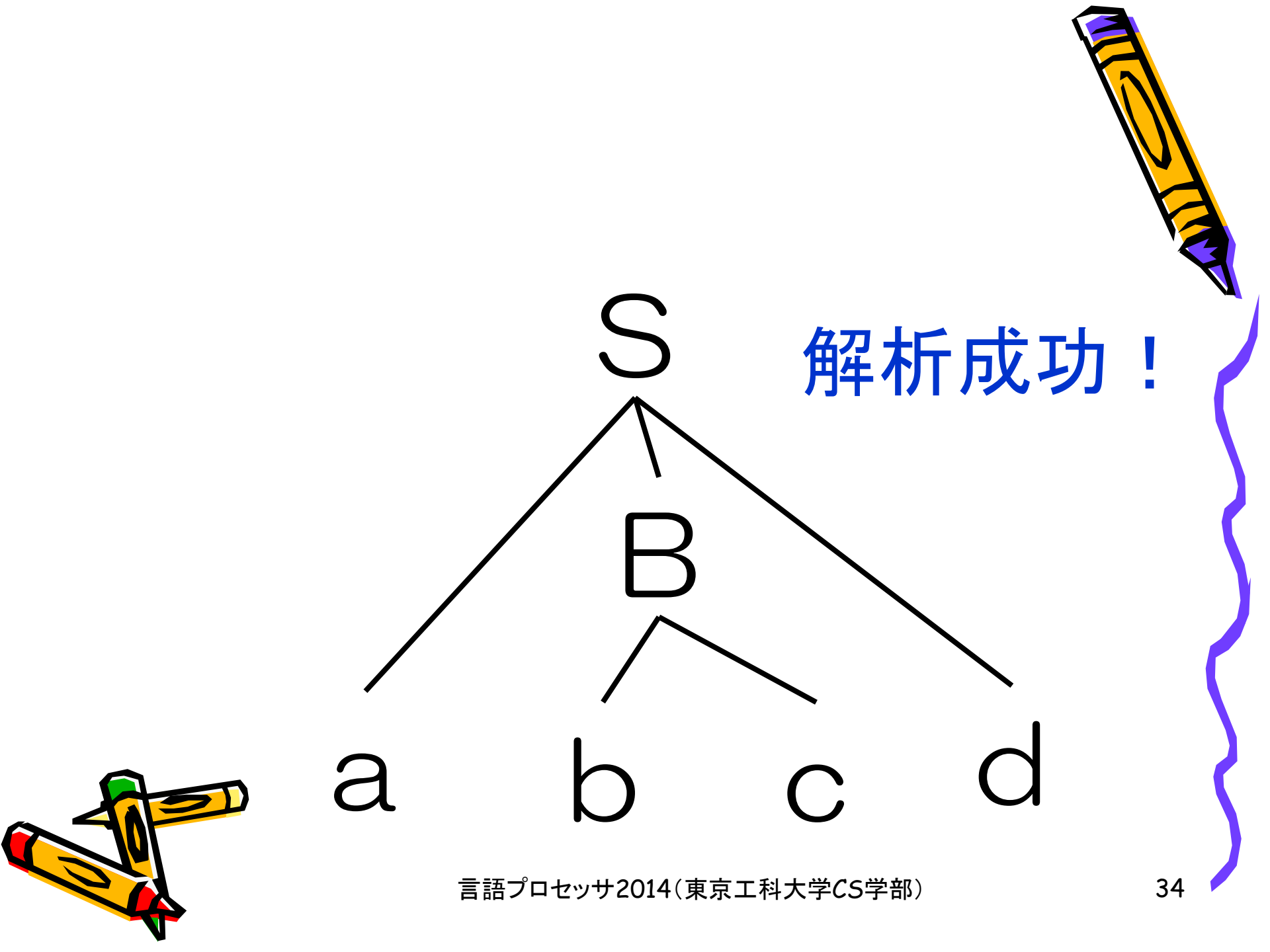



S
B
a
b
c
d

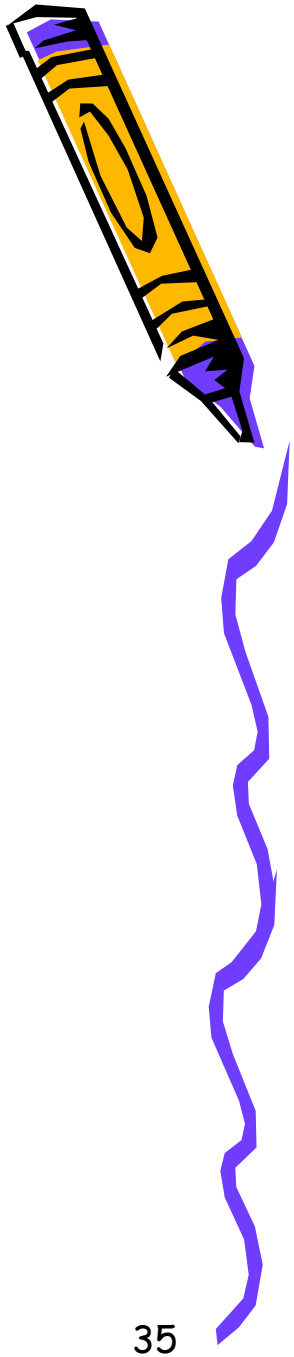

S
B
a
b
c
d
?

Backtrac発生！

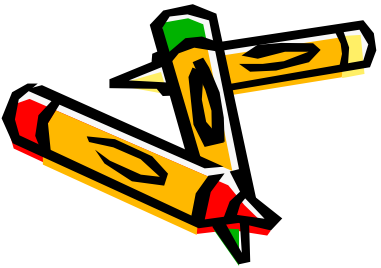




解析成功！



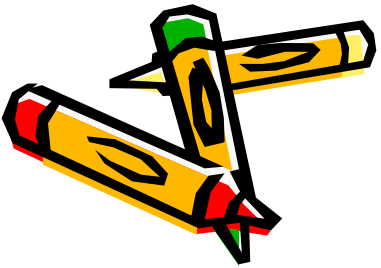
- backtrack回避の方法
  → 括りだし



1．括りだし

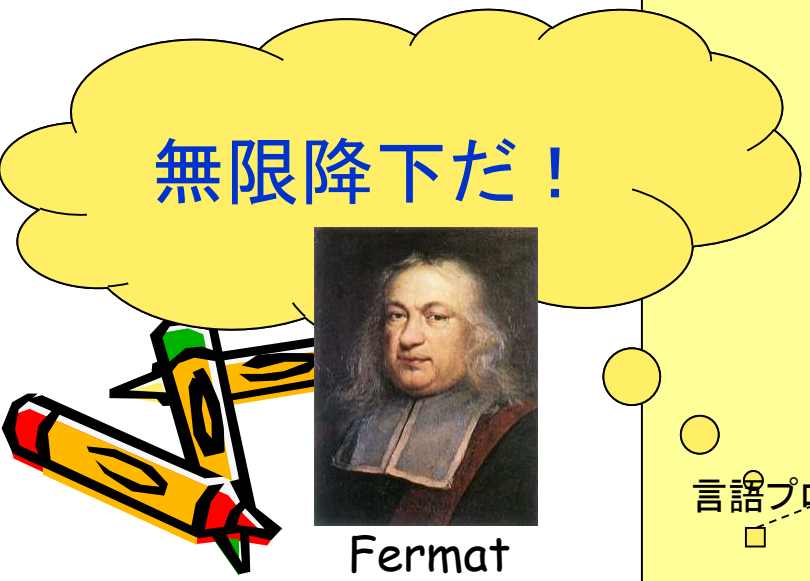
- 文法

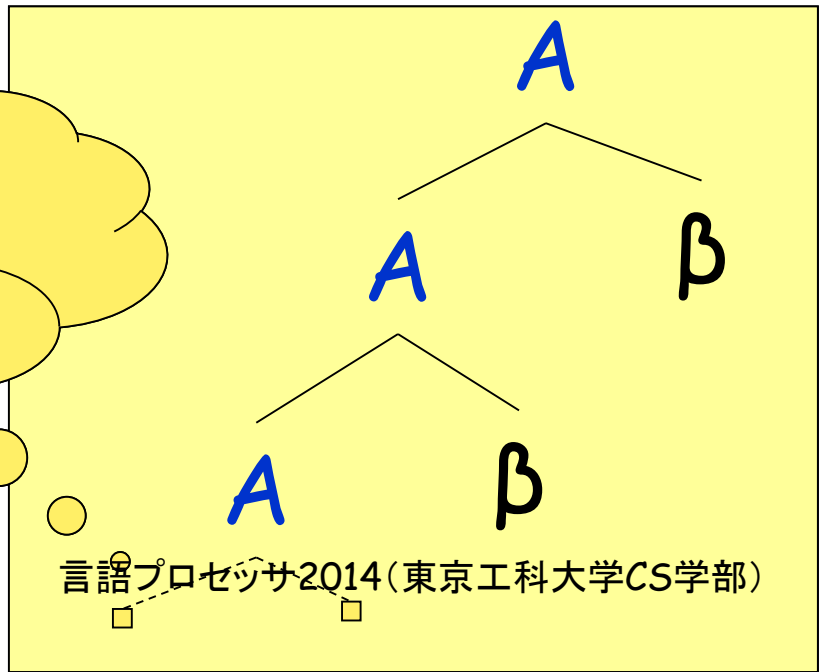
$$\begin{cases} S \to aBd \\ B \to b \mid bc \end{cases} \Longrightarrow \begin{cases} S \to aBd \\ B \to b\,(c \mid \varepsilon) \end{cases}$$



左再帰の回避

$$A \longrightarrow A\beta$$



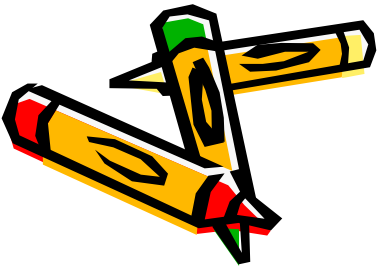


左再帰の回避方法

- $A \rightarrow A\alpha \mid \beta$

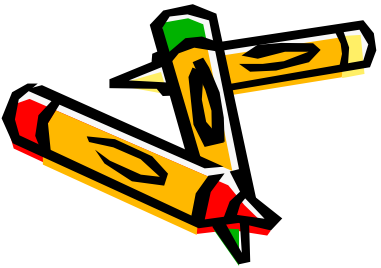
$$\Leftrightarrow \begin{cases} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{cases}$$

（教科書81ページ参照）



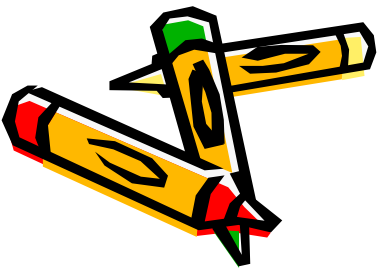
左再帰の例

$$
\begin{cases}
E \rightarrow E + T \mid T \\
T \rightarrow T * F \mid F \\
F \rightarrow (E) \mid id
\end{cases}
$$



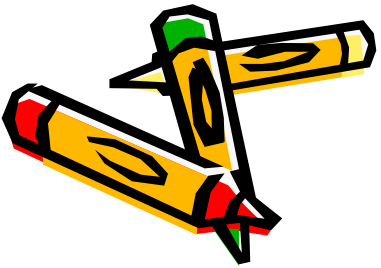
左再帰の回避

$$\begin{cases} E \rightarrow E + T \mid T \\ T \rightarrow T * F \mid F \\ F \rightarrow (E) \mid id \end{cases} \Rightarrow \begin{cases} E \rightarrow T\ E' \\ E' \rightarrow + T\ E' \mid \varepsilon \end{cases}$$



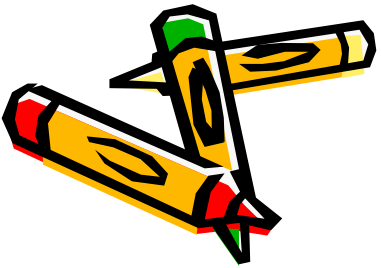
左再帰の回避

$$
\begin{cases} E \to E + T \mid T \\ T \to T * F \mid F \\ F \to (E) \mid \mathrm{id} \end{cases}
\Rightarrow
\begin{cases} E \to T\ E' \\ E' \to + T\ E' \mid \varepsilon \\ \\ T \to F\ T' \\ T' \to * F\ T' \mid \varepsilon \\ \\ F \to (E) \mid \mathrm{id} \end{cases}
$$

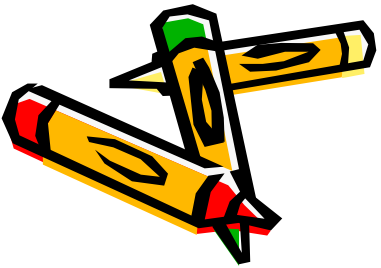


LL(1)文法

- LL(1)文法は、1文字先読みすることで、適用すべき規則が一意に決まる、という性質を備え持っている。
- つまり、「$A \rightarrow \alpha \mid \beta$」に対して、1文字先読みすれば、「$A \rightarrow \alpha$」と「$A \rightarrow \beta$」のどちらを適用すればいいのかが決まる。
（効率のよい処理が望める）



でも、与えられた文法がLL(1)文法であることをどうやって知ることができるのだろうか？



LL(1)文法の判定法

- First
- Follow