文法と言語
ー文脈自由文法とLR構文解析2ー

和田俊和

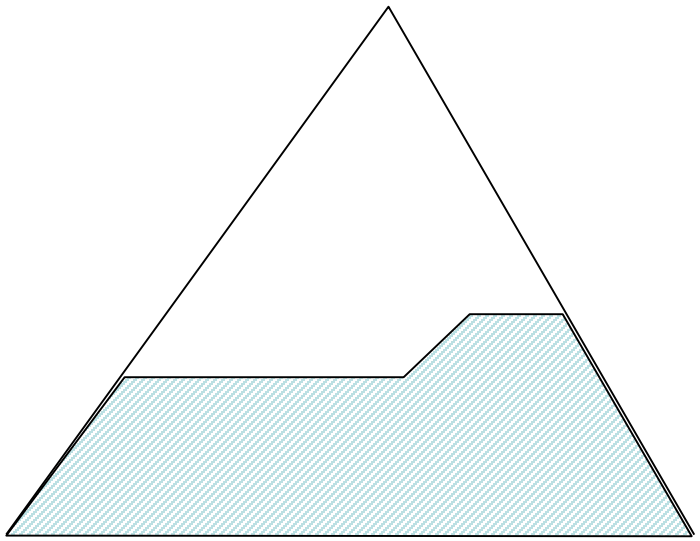
資料保存場所

http://vrl.sys.wakayama-u.ac.jp/~twada/syspro/

前回までの復習

最右導出と上昇型構文解析

- 最右導出を前提とした場合，上昇型の構文解析がしばしば用いられる.
- 上昇型構文解析では生成規則の右辺にマッチする部分を見つけ，それを左辺の非終端記号に置き換える「還元(reduction)」という操作を繰り返し，出発記号に到達するという



導出 $\longrightarrow$

$$A \rightarrow \alpha\beta\gamma$$

還元 $\longleftarrow$

還元操作を行うには手がかりが必要

- <算術式>→<算術式><加減演算子><項>
  という生成規則の右辺にマッチする部分を
  １－２－３－４
  という記号列から見つける問題を考える.
- 明らかに”１－２－３”が右辺の＜算術式＞に，次の”－”が＜加減演算子＞に，そして”4”が＜項＞に対応付けられる.
- これをどうやって見つけるかが問題

LR法例題

文法の例

(1) <E> →<E>*<B>

(2) <E> →<E>+<B>

(3) <E> →<B>

(4) <B> → 0

(5) <B> → 1

	アクション					GOTO	
状態	*	+	0	1	$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

アクション表と，GOTO表

アクション表は状態と終端記号でインデックス付けされている（終端記号 $ は入力の終わりを示す）。各マスには以下のようなアクションが記述されている:

– *shift*（s*n*）:　次の状態遷移先を *n* とする.

– *reduce*（r*m*）: *m*番の文法規則を実行する.

– *accept*（acc）: 入力バッファにある文字列を受理する.

GOTO表は状態と非終端記号でインデックス付けされており、各マスには非終端記号を解釈し終えた次に遷移する状態を示す.

LR法のアルゴリズム

- スタックを [**0**] と初期化する.（現在の状態は常にスタックトップの数字で表される）
- 現在の状態と入力にある記号からアクション表を参照する。ここで以下の4つのケースがある.
  - s*n* に従って状態遷移する.（これは，還元を行う文字列を一つ右に伸ばす）
    - 現在の終端記号を入力バッファから取り除く.
    - 状態 *n* をスタックにプッシュする.
  - r*m* に従って文法規則を適用する。（これは還元操作）
    - *m* 番の規則の右側にある各記号についてスタックから状態を削除する（例えば E * B なら3個ポップする）。
    - その時点のスタックのトップを状態とし、それと *m* 番の文法規則の左側からGOTO表を参照し、そこにある状態をスタックにプッシュして新たな状態とする.
  - 受理（acc）の場合、文字列の構文解析が完了。
  - アクションが指定されていない場合、文法エラーとなる.
- 上のステップを「受容」となるか「文法エラー」となるまで繰り返す。

前回の課題:0+1*1　の構文解析

状態	入力	スタック	アクション	意味	
0	0+1*1$	[0]	s1	‘0’を還元の候補にする	(0)+1*1
1	+1*1$	[1,0]	r4	‘0’を<B>に還元する	<B>+1*1
4	+1*1$	[4,0]	r3	<B>を<E>に還元する	<E>+1*1
3	+1*1$	[3,0]	s6	‘+’を還元の候補にする	<E>(+)1*1
6	1*1$	[6,3,0]	s2	‘1’を還元の候補にする	<E>(+)(1)*1
2	*1$	[2,6,3,0]	r5	‘1’を<B>に還元する	<E>(+)<B>*1
8	*1$	[8,6,3,0]	r2	<E>+<B>を<E>に還元する	<E>*1
3	*1$	[3,0]	s5	‘*’を還元の候補にする	<E>(*)1
5	1$	[5,3,0]	s2	‘1’を還元の候補にする	<E>(*)(1)
2	$	[2,5,3,0]	r5	‘1’を<B>に還元する	<E>(*)<B>
7	$	[7,5,3,0]	r1	<E>*<B>を<E>に還元する	<E>
3	$	[3,0]	acc	受理	

LR法例題

文法の例

(1) <E> →<E>*<B>

(2) <E> →<E>+<B>

(3) <E> →<B>

(4) <B> → 0

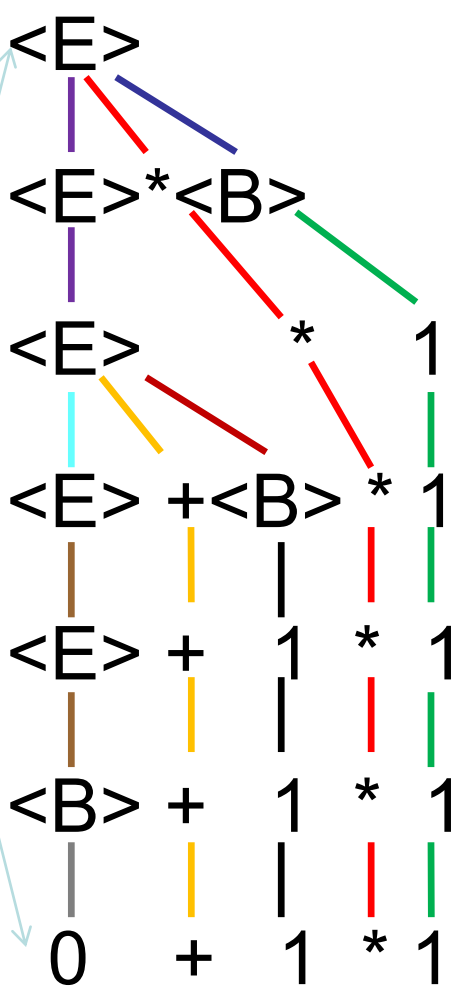
(5) <B> → 1

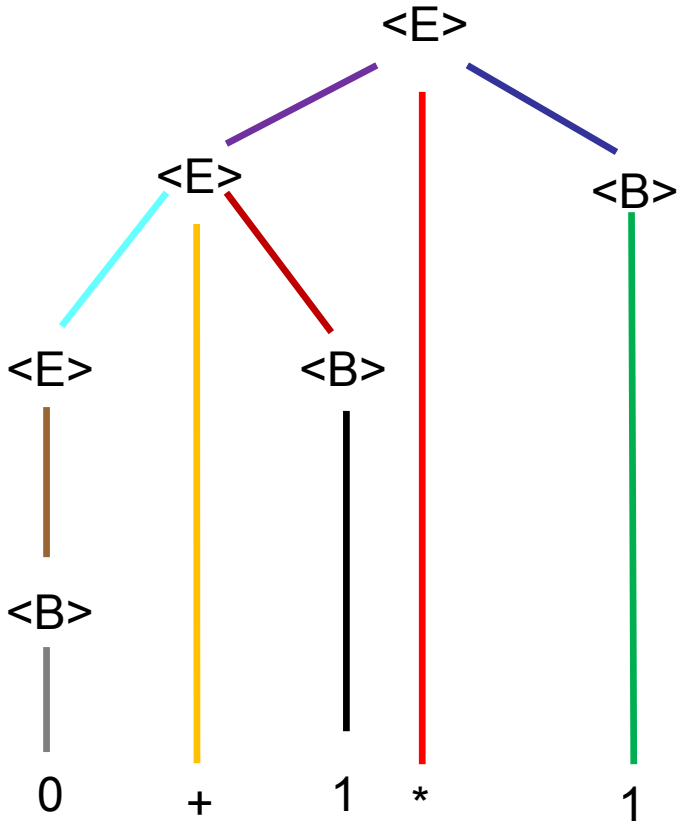
	アクション					GOTO	
状態	*	+	0	1	$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

前回の課題:0+1*1の導出木

上下逆転

(0)+1*1
<B>+1*1
<E>+1*1
<E>(+)1*1
<E>(+)(1)*1
<E>(+)<B>*1
<E>*1
<E>(*)1
<E>(*)(1)
<E>(*)<B>
<E>





今日の講義内容

アクション表・GOTO表の作り方

表の作成

- アイテム表記（構文解析の段階を表す表記法）

  E → • E + B

  E → E • + B

  E → E + • B

  E → E + B •

  E → E • + B は、E を認識した状態で，次に ‘+’ とBに対応する文字列がそれに続くことを期待していることを表している.

- アイテムの集合を状態として表を作る.

アイテム集合の作り方 : 1

- E → E • * B とアイテム E → E • + B は共に非終端記号 E を読んだ後で適用可能である. 従ってこれらのアイテムは1つの集合にまとめることができる.
- E → E + • B のように非終端記号の前にドットのあるアイテム がある場合, B 自体の構文解析を表すアイテム集合 B → • 1 や B → • 0 を 集合に加える必要がある. つまり,
  - *A* → *v*•*Bw* という形式のアイテムが集合内にあり、文法に *B* → *w′* という規則がある場合、アイテム *B* → • *w′* をアイテム集合に含めなければならない。

アイテム集合の作り方:2

- 文法の拡張(新たな出発記号の導入)を行う

  (0) <S> →<E>

  (1) <E> →<E>*<B>

  (2) <E> →<E>+<B>

  (3) <E> →<B>

  (4) <B> → 0

  (5) <B> → 1

アイテム集合

– **アイテム集合 0**

S → • E

+ E → • E * B

+ E → • E + B

+ E → • B

+ B → • 0

+ B → • 1

– **アイテム集合1（ ‘0’ ）**

B → 0 •

– **アイテム集合2（ ‘1’ ）**

B → 1 •

– **アイテム集合3（ ‘E ’)**

S → E •

E → E • * B

E → E • + B

アイテム集合

– **アイテム集合 4(‘B’)**

E → B •

– **アイテム集合5(‘*’)**

E → E * • B

+ B → • 0

+ B → • 1

– **アイテム集合6(‘＋’)**

E → E + • B

+ B → • 0

+ B → • 1

– **アイテム集合7(‘B’)**

E → E * B •

– **アイテム集合8(‘B’)**

E → E + B •

アイテム集合から状態遷移表

アイテム集合	*	+	0	1	E	B
0			1	2	3	4
1						
2						
3	5	6				
4						
5			1	2		7
6			1	2		8
7						
8						

状態遷移表から構文解析表，GOTO表へ

- 非終端記号に関する列はGOTO表に転記する.
- 終端記号に関する列はアクション表の shift アクションに転記する.
- 入力の終わりを示す ‘$’ の列をアクション表に追加し、アイテム S → E・を含むアイテム集合に対応するマスに *acc* を書く.
- アイテム集合 *i* が *A* → *w*・という形式を含み，対応する文法規則 *A* → *w* の番号 *m* が *m* > 0 なら，状態 *i* に対応するアクション表の行には全て reduce アクション r*m* を書く.

生成された構文解析表とGOTO表

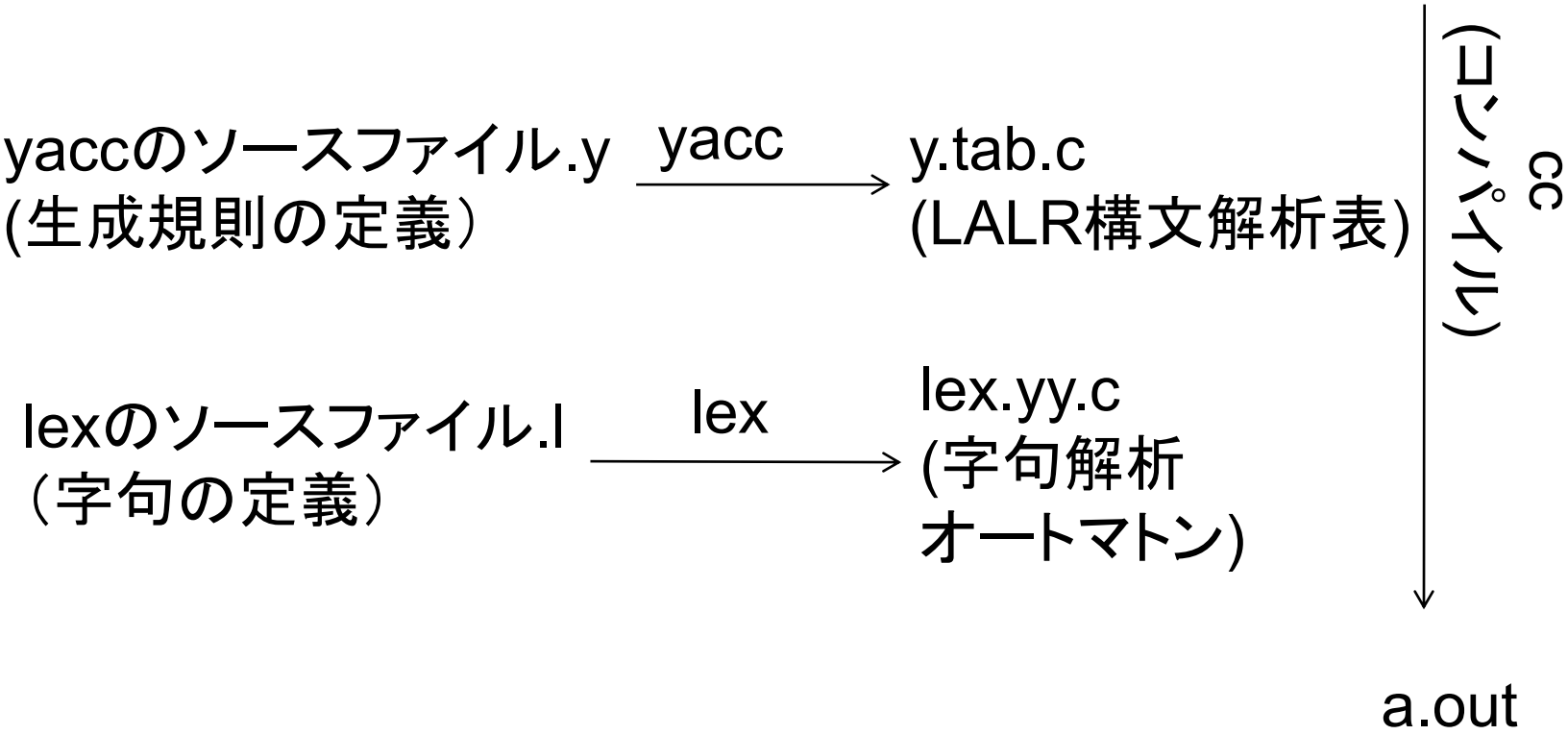
	アクション					GOTO	
状態	*	+	0	1	$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

Yet Another Compiler Compiler

- 文法を与えて，上昇型構文解析を行う構文解析器を生成するツール
- このソフトウエアが発表された当時，類似したソフトが多数あったため，皮肉でつけられた名前.
- lexによって生成した字句解析オートマトンを利用するように作られている.

yacc と lexを使ったプログラミング

- コンパイルの流れ



yaccソースファイルの例

```
%{
#include <stdio.h>
#include <ctype.h>
int yylex();
main() { yyparse(); }
%}
%token NUMBER
%left '+' '-'
%left '*' '/'
%right UMINUS
%%
lines :
    lines expr ‘¥n’ {printf("%d¥n",$2);}
    | lines '¥n'
    | /* empty */ ;
```

```
expr : expr '+' expr {$$ = $1 + $3;}
   |expr '-' expr {$$ = $1 - $3;}
   |expr '*' expr {$$ = $1 * $3;}
   |expr '/' expr {$$ = $1 / $3;}
   |'(' expr ')' {$$ = $2;}
   |'-' expr %prec UMINUS {$$ = - $2;}
   |NUMBER ;
%%
#include "lex.yy.c"
yyerror(mes)
char *mes;
{
fprintf(stderr,"%s¥n",mes);
}
```

%token はトークン，%leftは左結合的であること，%rightは右結合的であること，
$$は戻り値，$nは生成規則n番目の非終端記号の値を示す．

lexソースファイルの例・実行例

```
%%
[   ]      {}
[0-9]+     {sscanf(yytext,"%d",&yylval);
             return NUMBER;}
¥n         {return yytext[0];}
.          {return yytext[0];}
```

- 空白 は読み飛ばす
- 数字の列
  １０進数に変換し、値をyylvalに代入し、戻り値としてはNUMBERを返す
- 改行文字はそのまま返す
- その他の文字(ピリオドは任意の文字を表す) もそのまま返す

```
[twada@host]$ ls
lex.l  syntax.y
[twada@host]$ yacc syntax.y
[twada@host]$ ls
lex.l  syntax.y  y.tab.c
[twada@host]$ lex lex.l
[twada@host]$ ls
lex.l  lex.yy.c  syntax.y  y.tab.c
[twada@host]$ cc y.tab.c -lfl
```

```
[twada@host]$ ls
a.out  lex.l  lex.yy.c  syntax.y  y.tab.c
[twada@host]$ ./a.out
9/3/3-(8-4-3)
0
1+2+3+4+5+6+7+8+9+10
55
1+2 4+5
syntax error
```

下記文法の構文解析表とGOTO表を求めると，矛盾が生じることを示しなさい.

(1) <E> →<E>+<B>

(2) <E> →<B>

(3) <B> →<B>*<T>

(4) <B> →<T>

(5) <T> → 0

(6) <T> → 1