

コンパイラ理論 8 LR構文解析 補足: 属性文法とconflicts

櫻井彰人

属性文法

- Racc (Yacc系のcc)は属性文法的
 - 非終端記号は、値 (semantic value)を持つ
 - パーザーは、パーザースタックをreduceするとき (使う規則を $X ::= s$ とする)、s に付随する semantic value (Racc では配列 valueにある)を用いて、action (意味動作)を実行する。結果は終端記号 X の値となる (Raccではresultに入れた値)
 - パーザーが成功で終了すると、開始記号に付随する値を返す

属性文法

- 属性文法(ぞくせいぶんぽう, Attribute Grammar)とは、形式文法の生成に関する属性を定義する形式的手法。属性には値を関連付けられる。その言語を構文解析やコンパイラで処理する際に、属性の評価(属性から値を得ること)が抽象構文木上のノードで行われる。
- 属性は2種類に分類される。合成(synthesized)属性と継承(inherited)属性である。合成属性とは、属性評価の結果として生成されるものであり、継承属性の値を使用することもある。継承属性とは、親ノードから継承される属性である。
- いくつかの手法では、合成属性は意味情報を構文解析木の上に渡すのに使われ、継承属性は逆に下に使われる。例えば、言語変換ツールを作成する場合、属性文法は構文要素に意味(値)を設定するのに使われる。また、文法(構文規則だけでは明示的に示されない言語の規則)に従って意味論的検証を行うことも可能である。

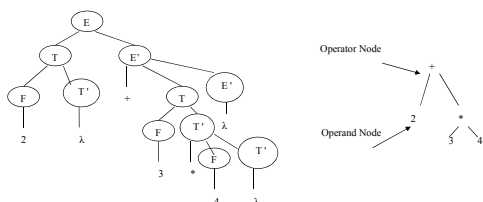
Wikipediaより

S属性文法とLR属性文法

- S属性文法
 - 継承属性を持たない属性文法。トップダウン構文解析でもボトムアップ構文解析でも使用可能。yacc は S属性文法に基づいている。
- LR属性文法
 - LR法を使った構文解析での属性文法。ボトムアップ構文解析で使用。L属性文法のサブセットであり、S属性文法のスーパーセットである。yacc は部分的に LR属性文法に基づいている。
 - In yacc, a common hack is to use global variables to simulate some kind of inherited attributes and thus LR-attribution.

Wikipediaより

構文木と抽象構文木



```
# $Id: calc.y,v 1.4 2005/11/20 13:29:32 aamine Exp $
#
# Very simple calculator.
class Calc
  prehigh
  nonassoc UMINUS
  left '* '/'
  left '+ '-'
  preclo
  rule
  target: exp
  /* none */ { result = 0 }
  exp: exp '+' exp { result += val[2] }
  | exp '-' exp { result -= val[2] }
  | exp '*' exp { result *= val[2] }
  | exp '/' exp { result /= val[2] }
  | (' exp ') { result = val[1] }
  | 'NUMBER' =UMINUS { result = -val[1] }
  | NUMBER
end
```

再びですが

```
rule
target: exp
| /* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
| exp '-' exp { result -= val[2] }
| exp '*' exp { result *= val[2] }
| exp '/' exp { result /= val[2] }
| '(' exp ')' { result = val[1] }
| '-' NUMBER =UMINUS { result = -val[1] }
| NUMBER
end
```

曖昧な文法

NUM + NUM * NUM \Rightarrow

再び

```
rule
target: exp
| /* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
| exp '-' exp { result -= val[2] }
| exp '*' exp { result *= val[2] }
| exp '/' exp { result /= val[2] }
| '(' exp ')' { result = val[1] }
| '-' NUMBER =UMINUS { result = -val[1] }
| NUMBER
end
```

曖昧ではあるが、分かりやすいし、複数の構文木が得られたとき、どちらが正しいかは(我々には)すぐ分かる。

NUM + NUM * NUM \Rightarrow

試してみよう

```
class Calcf
rule
target: exp
| /* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
| exp '-' exp { result -= val[2] }
| exp '*' exp { result *= val[2] }
| exp '/' exp { result /= val[2] }
| '(' exp ')' { result = val[1] }
| '-' NUMBER { result = -val[1] }
| NUMBER
end
```

```
$ racc -o calcf.rb calcf.y
16 shift/reduce conflicts

$ ruby calcf.rb

type "q" to quit.

? 1+2
= 3

? 1+2*3
= 7

? 1*2+3
= 5

?
```

LR パーザーの動き

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
| exp '*' exp { result *= val[2] }
| '(' exp ')' { result = val[1] }
| NUMBER
```

こちらが欲しい:

これから読む

入力: NUM + NUM * NUM

現在の状態: E + E

shift-reduce conflict あり!
正しく構文解析するにはどうしたらよいか?

これまで(に)パースした要素

LR パーザーの動き

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
| exp '*' exp { result *= val[2] }
| '(' exp ')' { result = val[1] }
| NUMBER
```

こちらが欲しい:

これから読む

入力: NUM + NUM * NUM

現在の状態: E + E *

shift-reduce conflict あり!
正しく構文解析するにはどうしたらよいか?
SHIFT

これまで(に)パースした要素

LR パーザーの動き

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
| exp '*' exp { result *= val[2] }
| '(' exp ')' { result = val[1] }
| NUMBER
```

こちらが欲しい:

これから読む

入力: NUM + NUM * NUM

現在の状態: E + E * NUM

SHIFT SHIFT

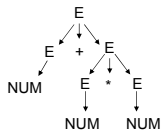
これまで(に)パースした要素

LR パーザーの動き

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | '(' exp ')' { result = val[1] }
    | NUMBER
```

こちらが欲しい:



これから読む

入力: NUM + NUM * NUM

現在の状態: E + E * E

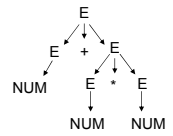
REDUCE

LR パーザーの動き

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | '(' exp ')' { result = val[1] }
    | NUMBER
```

こちらが欲しい:



これから読む

入力: NUM + NUM * NUM

現在の状態: E + E

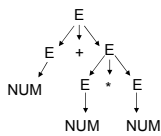
REDUCE

LR パーザーの動き

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | '(' exp ')' { result = val[1] }
    | NUMBER
```

こちらが欲しい:



これから読む

入力: NUM + NUM * NUM

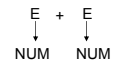
現在の状態: E

REDUCE

別の構文解析

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | '(' exp ')' { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM + NUM * NUM

現在の状態: E + E

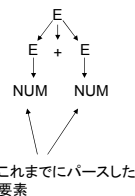
これまでにパースした要素

shift-reduce conflict あり!
先に REDUCE したらどうなるか

別の構文解析

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | '(' exp ')' { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM + NUM * NUM

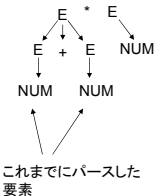
現在の状態: E

REDUCE

別の構文解析

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | '(' exp ')' { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM + NUM * NUM

現在の状態: E * E

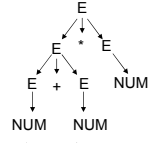
これまでにパースした要素

今度: SHIFT SHIFT REDUCE

別の構文解析

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | ('exp') { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM + NUM * NUM

現在の状態: E

これまでにパースした要素

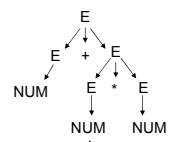
REDUCE

ここまでのまとめ

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '*' exp { result *= val[2] }
    | ('exp') { result = val[1] }
    | NUMBER
```

こちらが欲しい:



これから読む

入力: NUM + NUM * NUM

現在の状態: E + E

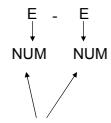
これまでにパースした要素

shift-reduce conflict あり!
スタック上に E + E あり, そして *.
Shift したい, こんなときいつでも shift したい
というの * の優先度 precedence は + より上.

第二例

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '-' exp { result -= val[2] }
    | exp '*' exp { result *= val[2] }
    | ('exp') { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM - NUM - NUM

現在の状態: E - E

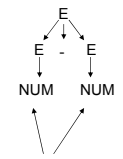
これまでにパースした要素

shift-reduce conflict あり!
スタック上に E - E あり, 次にあるのは -.
何をすべきか?

第二例

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '-' exp { result -= val[2] }
    | exp '*' exp { result *= val[2] }
    | ('exp') { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM - NUM - NUM

現在の状態: E

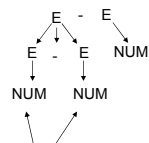
これまでにパースした要素

shift-reduce conflict あり!
スタック上に E - E あり, 次にあるのは -.
何をすべきか?
REDUCE

第二例

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '-' exp { result -= val[2] }
    | exp '*' exp { result *= val[2] }
    | ('exp') { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM - NUM - NUM

現在の状態: E - E

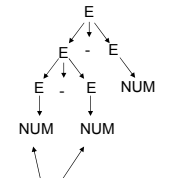
これまでにパースした要素

SHIFT SHIFT REDUCE

第二例

一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '-' exp { result -= val[2] }
    | exp '*' exp { result *= val[2] }
    | ('exp') { result = val[1] }
    | NUMBER
```



これから読む

入力: NUM - NUM - NUM

現在の状態: E

これまでにパースした要素

REDUCE

第2例: まとめ

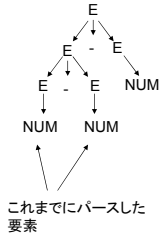
一部だけ取り出す

```
exp: exp '+' exp { result += val[2] }
    | exp '-' exp { result -= val[2] }
    | exp '*' exp { result *= val[2] }
    | (' exp ')' { result = val[1] }
    | NUMBER
```

入力: NUM - NUM - NUM

現在の状態: E

shift-reduce conflict あり!
 スタック上に E - E あり, 次にあるのは -.
 何をすべきか? いつでも
 reduce をしたい. というのも - は left-associative.



優先度と結合性

- 優先度と結合性を扱う3つの方法がある:
 - 1) Racc に文句を言わせたままにする.
 - Raccは(Yaccは), shift-reduce conflict があると shift する
 - ただし, そうすると, プログラマの意図が分かりにくくなる; 他の部分をデバッグするのに支障がでるかも; 一般的にはエレガントではない
 - 2) あいまい性がない文法に書き換える
 - 複雑かつ分かりにくくなるおそれあり
 - 3) Racc (Yacc) の優先度指定を用いる
 - もっとも普通. left, right, nonassoc

優先度と結合性

- 優先度が指示されると, Racc は終端記号と規則に優先度を割り当てる
 - 終端記号の優先度は, 左右結合が書かれている順番(またはその逆. 指示に従う)
 - 書換規則の優先度は, 最右端終端記号の優先度である
 - 例: 規則 $E ::= E + E$ の優先度 $== \text{prec}(+)$
- shift-reduce conflict の解消方法:
 - $\text{prec}(\text{terminal}) > \text{prec}(\text{rule}) \implies \text{shift}$
 - $\text{prec}(\text{terminal}) < \text{prec}(\text{rule}) \implies \text{reduce}$
 - $\text{prec}(\text{terminal}) = \text{prec}(\text{rule}) \implies$
 - $\text{assoc}(\text{terminal}) = \text{left} \implies \text{reduce}$
 - $\text{assoc}(\text{terminal}) = \text{right} \implies \text{shift}$
 - $\text{assoc}(\text{terminal}) = \text{nonassoc} \implies \text{エラー}$

入力: 終端記号 T が次:.....T E

スタック上の、規則の右辺:.....E % E

優先度と結合性

```
prechigh
nonassoc UMINUS
left '*' '/'
left '+' '-'
preclow
rule
target: exp
/* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
    | exp '-' exp { result -= val[2] }
    | exp '*' exp { result *= val[2] }
    | exp '/' exp { result /= val[2] }
    | (' exp ')' { result = val[1] }
    | ' NUMBER =UMINUS { result = -val[1] }
    | NUMBER
end
```

優先度と結合性

```
prechigh
nonassoc UMINUS
left '*' '/'
left '+' '-'
preclow
```

入力: 終端記号 T が次:.....E E

スタック上の、規則の右辺: ...E '+' E

$\text{prec}('*') > \text{prec}('+)$

優先度と結合性

```
prechigh
nonassoc UMINUS
left '*' '/'
left '+' '-'
preclow
```

入力: 終端記号 T が次:.....E E

スタック上の、規則の右辺: ...E '+' E

$\text{prec}('*') > \text{prec}('+)$

SHIFT

優先度と結合性

```

prehigh
nonassoc UMINUS
left   '*' '/'
left   '+' '-'
preclow
    
```

入力: 終端記号 T が次:^{これから読む} '^' E

スタック上の、規則の右辺: ...E '^' E prec('+') = prec('^')

優先度と結合性

```

prehigh
nonassoc UMINUS
left   '*' '/'
left   '+' '-'
preclow
    
```

入力: 終端記号 T が次:^{これから読む} '^' E

スタック上の、規則の右辺: ...E '^' E prec('+') = prec('^')

REDUCE

もう一例

```

prehigh
left   '*' '/'
left   '+' '-'
preclow
rule
target: exp
| /* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
| exp '-' exp { result -= val[2] }
| exp '*' exp { result *= val[2] }
| exp '/' exp { result /= val[2] }
| '(' exp ')' { result = val[1] }
| '^' NUMBER { result = -val[1] }
| NUMBER
    
```

.....^{これから読む} '^' E

...'^' E

どうなるか？

もう一例

```

prehigh
left   '*' '/'
left   '+' '-'
preclow
rule
target: exp
| /* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
| exp '-' exp { result -= val[2] }
| exp '*' exp { result *= val[2] }
| exp '/' exp { result /= val[2] }
| '(' exp ')' { result = val[1] }
| '^' NUMBER { result = -val[1] }
| NUMBER
    
```

.....^{これから読む} '^' E

...'^' E

どうなるか？

prec('*') > prec('^') ==> SHIFT

修正

```

prehigh
nonassoc UMINUS
left   '*' '/'
left   '+' '-'
preclow
rule
target: exp
| /* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
| exp '-' exp { result -= val[2] }
| exp '*' exp { result *= val[2] }
| exp '/' exp { result /= val[2] }
| '(' exp ')' { result = val[1] }
| '^' NUMBER=UMINUS { result = -val[1] }
| NUMBER
    
```

.....^{これから読む} '^' E

...'^' E

どうなるか？

修正

```

prehigh
nonassoc UMINUS
left   '*' '/'
left   '+' '-'
preclow
rule
target: exp
| /* none */ { result = 0 }

exp: exp '+' exp { result += val[2] }
| exp '-' exp { result -= val[2] }
| exp '*' exp { result *= val[2] }
| exp '/' exp { result /= val[2] }
| '(' exp ')' { result = val[1] }
| '^' NUMBER=UMINUS { result = -val[1] }
| NUMBER
    
```

.....^{これから読む} '^' E

...'^' E

どうなるか？

prec('^') > prec('*') ==> REDUCE

宙ぶらりん else (dangling else)

- 文法:
S ::= if E then S else S
S ::= if E then S
S ::= ...
- 例題: if a then if b then S else S
 - 解析 1: if a then (if b then S else S)
 - 解析 2: if a then (if b then S) else S
- Raccは shift-reduce conflict を報告
 - デフォルト: shift (望みのもの)

宙ぶらりん else (dangling else)

- 文法:
S ::= if E then S else S
S ::= if E then S
S ::= ...
- 別解: 文法の書き換え:
S ::= M
S ::= U
M ::= if E then M else M
M ::= ...
U ::= if E then S
U ::= if E then M else U

Racc のデフォルト動作

- Shift-Reduce conflict
 - shift
- Reduce-Reduce conflict
 - 最初の規則で reduce
 - 一般的には、本当のバグ!