第4章 文脈自由文法とプッシュダウン・オートマトン

4.1 バッカス・ナウア記法

BNF (Backus=Naur Form, __________) は言語の文法を記述するための形式である。まず例で説明する。

例 1

$$\begin{aligned} expr &\rightarrow expr \text{ “+” } num \\ &\mid\; expr \text{ “−” } num \\ &\mid\; num \\ num &\rightarrow \text{“0”} \mid \text{“1”} \mid \ldots \mid \text{“9”} \end{aligned}$$

記号の読み方は以下の通りである。

$\rightarrow$	左辺は右辺の形からなる。左辺を右辺の形で書き換える。
$\mid$	または

ようするに、BNF は書き換え規則の集まりである。BNF の場合、左辺は______に限られ、右辺は記号列（空の場合も含む）である。この規則で $expr$ から書き換えられるものを $expr$ （すなわち expression — 式）とみなす。再帰的である（右辺の記号列の中に、左辺の記号が現れても良い）ところが、大事な正規表現との違いである。 BNF で記述される文法を________(context-free grammar) と言う。文脈自由文法で生成される言語を文脈自由言語 (context-free language) という。

ついでに英語も覚えよう。

expression	式
number	数

ちなみに、文脈依存文法 (context-sensitive grammar) とは書き換え規則が、次のような形をしている文法である。

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

ここで A は一つの記号で、α, β, γ は記号列である。

文脈依存文法は文脈自由文法よりも強い表現力を持つが、プログラミング言語の文法は、通常、文脈自由文法の範囲に収まってしまうので、文脈依存文法が必要となることは、ほとんどない。

「“1 + 2 * 3” は $expr$ である。」（「$expr$ は “1 + 2 * 3”を______。」ともいう）ことを示すには、次のような列を示せばよい。

$$\begin{aligned} expr &\Rightarrow expr + num \Rightarrow expr + 3 \\ &\Rightarrow expr + num + 3 \Rightarrow expr + 2 + 3 \\ &\Rightarrow num + 2 + 3 \Rightarrow 1 + 2 + 3 \end{aligned}$$

このような $\Rightarrow$ の列が存在することを、まとめて「$expr \overset{*}{\Rightarrow} 1+2+3$」と書く。

ここで記号の意味は次の通りである。

$\Rightarrow$　　$\rightarrow$ を記号列（の一部）に適用すること
$\overset{*}{\Rightarrow}$　　$\Rightarrow$ を 0 回以上適用すること

Q 4.1.1 $expr$ は "7 + 8 - 9" を導出することを示せ。

4.2 用語

________ (terminal)
　実際にプログラム中に現れる記号、これ以上書き換えできない記号
　BNF では引用符に囲んで表す（が、明らかなら省略する）
　さっきの例では 0, 1, +, - など

________ (non-terminal)
　$\rightarrow$ の左辺に現れる記号、書き換えできる記号
　文法上の分類（式・文など）を表す。さっきの例では、$expr$ と num

__（イプシロン）
　空の記号列を表す。

以上の 3 つを合わせて________という。一方、「$\rightarrow$」と「|」はメタ記号（meta symbol）という。

「メタ」は「高次の」「超」という意味の接頭辞である。

________ (production rule)
　「非終端記号 $\rightarrow$ 構文記号の列」のかたちのこと

________ (start symbol)
　「プログラム」など、BNF で主に表現したい非終端記号のこと
　（特に断らない限り、BNF の一番上に書かれた非終端記号が開始記号になる。）

習慣として、非終端記号は*斜体 (italic)* や 筆記体 abc で、終端記号は**太字 (bold)**や活字体 abc あるいは引用符囲み（" ～ "）で書き分けることが多い。

例 2

$$
\begin{aligned}
expr &\rightarrow term \mid expr + term \mid expr - term \\
term &\rightarrow factor \mid term * factor \mid term \,/\, factor \\
factor &\rightarrow const \mid var \mid (expr) \\
const &\rightarrow \ldots \\
var &\rightarrow \ldots
\end{aligned}
$$

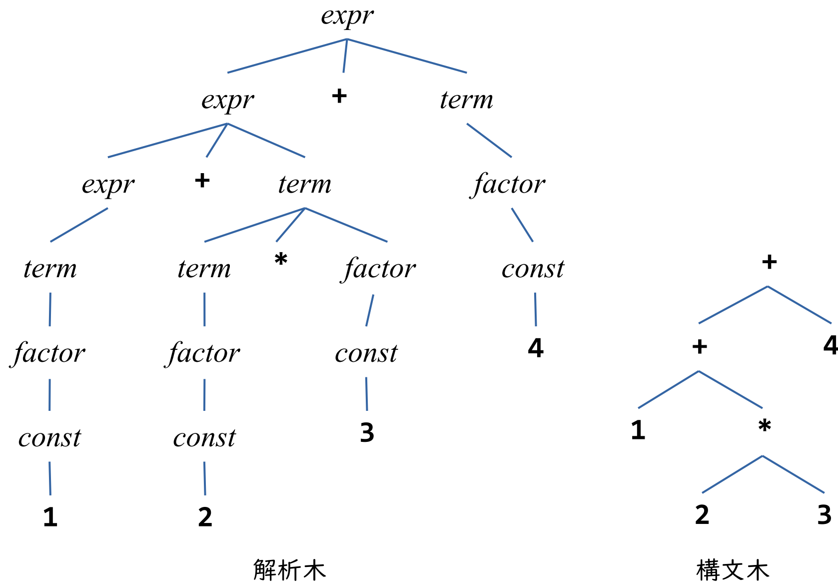
ついでに英語も覚えよう。

term	項
factor	因子
constant	定数
variable	変数

「1 + 2 * 3 + 4」は $expr$ である。

$$
\begin{aligned}
expr &\Rightarrow expr + term \Rightarrow expr + factor \Rightarrow expr + const \\
&\Rightarrow expr + 4 \Rightarrow expr + term + 4 \Rightarrow \ \ldots
\end{aligned}
$$

______(parse tree)	導出列を木のかたちにあらわしたもの。 本質的でない書き換え順の違いを無視できる。
______(syntax tree)	解析木を圧縮したもの 構文木の節は演算子などである。（解析木の節は非終端記号である。）



解析木　　　　構文木

注: 例 2 の BNF は、「*」が「+」より、**結合力が強い**ことなどを表現できている。

問 4.2.1 以下の記号列に対する解析木を書け。

1. `1 + 2 * 3 / 4`（例 2 の BNF で $expr$ から）
2. `while (i > 0) { n = n * i; i = i - 1; }`（教科書 p.14 の BNF で Statement から）

例 3

$$
\begin{aligned}
expr &\rightarrow const \mid expr + expr \mid expr * expr \\
const &\rightarrow \ \ldots
\end{aligned}
$$

「`1 + 2 * 3`」の解析木は?

解析木 (1)　　　　解析木 (2)

このような構文規則は＿＿(ambiguous) である、という

曖昧 (あいまい、ambiguous)	一つの記号列に対して、解析木が何通りもあること

曖昧な文法が役に立たない、というわけではなく、BNF に加えて＿＿と＿＿を指定して、曖昧でなくすことができる（こともある）。

例

「3 - 2 - 1」、「1 + 2 * 3」など

左結合
: 同じ優先順位の演算子は左を優先する。ほとんどのプログラミング言語の「+」,「-」演算子などは左結合である。

右結合
: 同じ優先順位の演算子は右を優先する。例えば C 言語の「=」演算子などは右結合である。（「x = y = 0」は「x = (y = 0)」である。）

非結合
: 同じ優先順位の演算子が隣りあうとエラーになる。プログラミング言語によっては「<」,「>」演算子などは非結合である

4.3 プッシュダウン・オートマトン

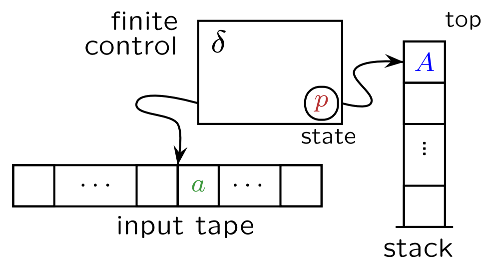
正規表現に対して有限オートマトンが対応するように、文脈自由文法に対しても（非決定性）プッシュダウン・オートマトンという抽象機械が対応することが知られている。以下では、知られている結果のみを証明なしに述べる。

4.3.1 非決定性プッシュダウン・オートマトン

プッシュダウン・オートマトンとは、入力と状態のほかに＿＿を持ち、

- 入力の先頭の記号（ϵ でも良い）と状態の他に、＿＿も考慮に入れて、次の動作が決まる。
- 次の動作のときに、スタックの操作（一番上の記号をポップして、0 個以上の記号をプッシュする）を伴っても良い。

という点が有限オートマトンと異なる。非決定性プッシュダウン・オートマトンは、入力の先頭とスタックの一番上と状態で次の動作が一意に定まらない（つまり、いくつかの選択肢がある）プッシュダウン・オートマトンである。



Jochgem / CC BY-SA (https://creativecommons.org/licenses/by-sa/3.0)

次の事実が知られている。

- 任意の文脈自由文法に対して、それが生成する記号列をちょうど受理する非決定性プッシュダウン・オートマトンを構成することができる。
- 任意の非決定性プッシュダウン・オートマトンに対して、それが受理する記号列を生成する文脈自由文法を構成することができる。

非決定性プッシュダウン・オートマトンをそのままプログラムとして実装しようとすると、効率が良くない。そのため、プログラミング言語の構文解析で非決定性プッシュダウン・オートマトンをそのまま用いることはほとんどない。

文脈自由言語を判定するアルゴリズムは、動的計画法を用いる Cocke-Younger-Kasami 法なども知られている。最悪の場合、記号列の長さの 3 乗に比例する計算時間がかかることが知られている。

4.3.2 決定性プッシュダウン・オートマトン

決定性プッシュダウン・オートマトンは、入力の先頭とスタックのトップと状態で次の動作が一意に定まるプッシュダウン・オートマトンである。有限オートマトンの場合は、非決定性有限オートマトンと決定性有限オートマトンの表現力は同じであった。しかし、非決定性プッシュダウン・オートマトンと決定性プッシュダウン・オートマトンの場合は同じではない、つまり、非決定性プッシュダウン・オートマトンでは受理できるが、決定性プッシュダウン・オートマトンでは受理できない言語があることが知られている。

また、決定性プッシュダウン・オートマトンにより受理される言語は、 (ただし、k ≥ 1) という方法で受理される言語と一致することが知られている。 LR 法については後期の「コンパイラ」で触れる。

現在、実用的に効率の良い構文解析法が知られているのは、LR(1) のさらにサブセットになる LALR(1) という構文解析法である。

4.4 （文脈自由言語の）反復補題

文脈自由文法では表現できない言語（記号列の集合）がある。ある言語が文脈自由言語で表せないことを示すためには、次の補題が役に立つ。

文脈自由言語の反復補題 (pumping lemma for context-free languages)

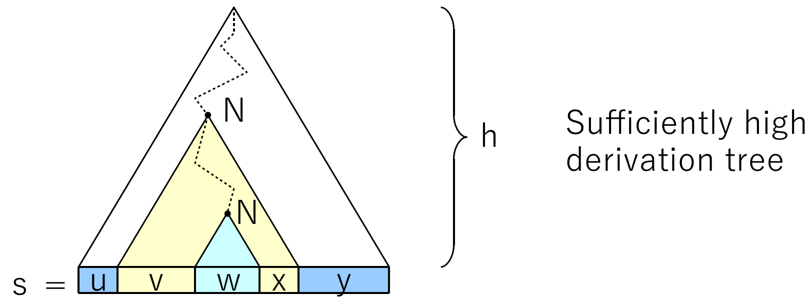
ポンプの補題ともいう。

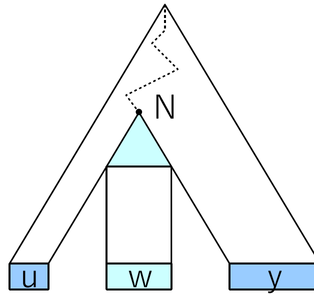
文脈自由文法が表現する言語 L に対して、次のような条件を満たす自然数 $p\ (\geq 1)$ が存在する。

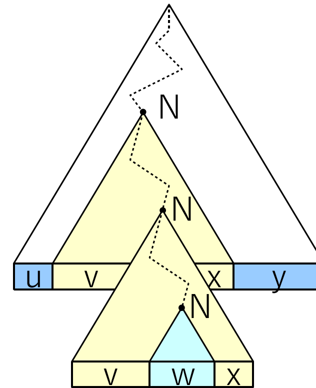
L に属する長さ p 以上の任意の文字列 s は $s = uvwxy$ と書けて、

1. vx の長さは 1 以上である。つまり、v, x の少なくとも一方は空ではない。
2. vwx の長さは p 以下である。
3. すべての $n\ (\geq 0)$ に対して、uv^nwx^ny も L に属する。

正確な証明をするためには、文脈自由文法の標準形の議論をしなければいけないので割愛する。概略は以下のような図を見るとわかりやすい。







Generating uv ^{0}wx y^0　　Generating uv ^{2}wx y^2

Jochen Burghardt / CC BY-SA (https://creativecommons.org/licenses/by-sa/3.0)

つまり、L の記号列がある程度長くなると、解析木の高さも必然的に大きくなり、どこかの経路で同じ非終端記号（上の図の場合は N）が 2 回現れる。この N から N まで部分をカットしたり（左下の図）、繰り返したり（右下の図）することで得られる記号列もやはり L に属する。

例

$\{a^ib^ic^i \mid i \geq 0\}$ つまり $\{\epsilon, abc, aabbcc, aaabbbccc, aaaabbbbcccc, \ldots\}$ という言語 L は文脈自由文法では表せない。

証明: 文脈自由文法で表せたとして矛盾を導く。 L を文脈自由文法で表せたとすると、反復補題により、上のような条件を満たす自然数 $p\ (\geq 1)$ が存在する。 $s = a^pb^pc^p$ とする。すると、$s = uvwxy$ と分解したときに vwx の長さは p 以下なので、 a と c の両方を含むことはない。すると uwy は L に属するが、 a と c の出現回数が異なり矛盾する。