計算理論

Thomas Zeugmann

北海道大学 大学院情報科学研究科
アルゴリズム研究室

http://www-alg.ist.hokudai.ac.jp/~thomas/ToC/

第 7 回：続・文脈自由文法の性質
(日本語訳：大久保 好章，湊 真一)

バッカス-ナウア形 I

前回の講義で述べた通り，文脈自由文法はプログラミング言語の重要な基礎をなす．しかし，その言語仕様においては，我々が学んだ 文脈自由文法の定義とは異なる形式が通常用いられる．

バッカス-ナウア形 I

前回の講義で述べた通り，文脈自由文法はプログラミング言語の重要な基礎をなす．しかし，その言語仕様においては，我々が学んだ 文脈自由文法の定義とは異なる形式が通常用いられる．
この形式は**バッカス標準形 *(Backus Normal Form)***，あるいは，**バッカス-ナウア形 *(Backus-Naur Form)*** と呼ばれる．これは，ALGOL の文法を記述するために John Backus が作り出したものであり，後に，Peter Naur により文字セットを減らすために単純化され，この新たな形式を Donald Knuth が Backus-Naur 形と名付けた．幸いにも，Knuth の提案に従ってかどうかは別にして，この形式は BNF と広く略されている．

バッカス-ナウア形 II

その形式は，作業語彙 $T \cup N$ に出現しない 4 つのメタ文字を用いる．これらは

$$\langle \quad \rangle \quad ::= \quad |$$

であり，使いかたは次の通りである．(メタ文字を含まない) 文字列は，$\langle$ と $\rangle$ で囲われる．記号 $::=$ は ($\to$ と同様に) 置換操作を表し，$|$ は “または (or)” と解釈される．

バッカス-ナウア形 III

例. 1

プログラミング言語における，符号なし整数に関する文脈自由文法を考えよう．ここで，D は整数のクラスを，u は符号なし整数のクラスを表すとする．

$$
\begin{array}{lcl}
\mathrm{D} & \to & 0 \\
\mathrm{D} & \to & 1 \\
\mathrm{D} & \to & 2 \\
\mathrm{D} & \to & 3 \\
\mathrm{D} & \to & 4 \\
\mathrm{u} & \to & \mathrm{D}
\end{array}
\qquad
\begin{array}{lcl}
\mathrm{D} & \to & 5 \\
\mathrm{D} & \to & 6 \\
\mathrm{D} & \to & 7 \\
\mathrm{D} & \to & 8 \\
\mathrm{D} & \to & 9 \\
\mathrm{u} & \to & \mathrm{uD}\,.
\end{array}
$$

バッカス-ナウア形 III

例. 1

プログラミング言語における，符号なし整数に関する文脈自由文法を考えよう．ここで，D は整数のクラスを，u は符号なし整数のクラスを表すとする．

$$\begin{array}{lcl} D & \to & 0 \\ D & \to & 1 \\ D & \to & 2 \\ D & \to & 3 \\ D & \to & 4 \end{array} \qquad \begin{array}{lcl} D & \to & 5 \\ D & \to & 6 \\ D & \to & 7 \\ D & \to & 8 \\ D & \to & 9 \end{array}$$

$$u \to D \qquad u \to uD\,.$$

これを BNF に書換えると

$$\begin{array}{rcl} \langle \text{digit} \rangle & ::= & 0|1|2|3|4|5|6|7|8|9 \\ \langle \text{unsigned integer} \rangle & ::= & \langle \text{digit} \rangle | \langle \text{unsigned integer} \rangle \langle \text{digit} \rangle \end{array}$$

バッカス-ナウア形 IV

この例からわかるように，BNF は非常に簡潔な表現を与える．

適当と思われるときは，BNF における表記を組み合わせて用いる．例えば，⟨，⟩ と一緒に | を用いる．ただし，::= は用いない．

動機

文脈自由文法は，多くの応用において重要な役割を果たす．正規言語に関しては，有限オートマトンが非常に効率的な認識器であることを見てきた．では，文脈自由言語についてはどうであろうか？

動機

文脈自由文法は，多くの応用において重要な役割を果たす．正規言語に関しては，有限オートマトンが非常に効率的な認識器であることを見てきた．では，文脈自由言語についてはどうであろうか？

任意の文脈自由言語についても，同様に認識器を計算論的に構成することができる．形式的に述べると，これらはプッシュダウンオートマトン (*pushdown automata*) である．所与の言語に対して，適当なプッシュダウンオートマトンを構成する多くのソフトウェアが存在する．これらシステムは，コンパイラの構文解析部を高速に構成してくれることから重要であり，故に，高く評価されている．

動機

文脈自由文法は，多くの応用において重要な役割を果たす．正規言語に関しては，有限オートマトンが非常に効率的な認識器であることを見てきた．では，文脈自由言語についてはどうであろうか？

任意の文脈自由言語についても，同様に認識器を計算論的に構成することができる．形式的に述べると，これらはプッシュダウンオートマトン (*pushdown automata*) である．所与の言語に対して，適当なプッシュダウンオートマトンを構成する多くのソフトウェアが存在する．これらシステムは，コンパイラの構文解析部を高速に構成してくれることから重要であり，故に，高く評価されている．

これらについては，第 9 回・第 10 回の講義で学ぶ．

動機

まず最初に，構文解析のための別の重要なツールである構文解析器/パーサ (*parser*) について述べると都合がよい．

動機

まず最初に，構文解析のための別の重要なツールである構文解析器/パーサ (*parser*) について述べると都合がよい．
最も広く使われている構文解析器の生成器は，yacc (yet another compiler-compiler) と呼ばれるものである．構文解析器の生成，すなわち，ソースプログラムから構文木を作り出す関数は，すべての UNIX システムで利用可能な yacc コマンドに組織化されている．yacc への入力は CFG であり，その表記は，よく知られた BNF と細部においてのみ異なる．各生成規則には**アクション (*action*)** が結びつけられている．これは，この生成規則に対応するノード (とその子ノード) が作り出されるときに実行される C のコードの一部である．では，構文木について続けよう．

構文木 I

文法の優れた特徴は，それが定義する言語の文の階層的な構文構造を表す点にある．こうした階層的構造は構文木 (*parse tree*) により表される．

構文木 I

文法の優れた特徴は，それが定義する言語の文の階層的な構文構造を表す点にある．こうした階層的構造は構文木 (*parse tree*) により表される．

構文木は導出に関する表現である．コンパイラで用いられるときは，ソースプログラムを表現するためのデータ構造と言える．コンパイラでは，ソースプログラムの木構造は，そのソースの実行コードへの変換を容易にする．その際，こうした変換は自然な再帰関数で表現される．

構文木 II

構文木は次の通り定義される木である．

定義 1

$\mathcal{G} = [T, N, \sigma, P]$ を文脈自由文法とする．$\mathcal{G}$ に関する **構文木 *(parse tree)*** とは，次の条件を満たす木である．

(1) 各内部ノードと根 (ルート) ノードは N 中の変数でラベル付けされている．

(2) 各葉ノードは，非終端記号，終端記号，あるいは，空文字列のいずれかでラベル付けされている．特に，空文字列でラベル付けされた葉ノードは，その親の唯一の子供でなければならない．

(3) もし，ある内部ノードが h で，その子供が左から右へそれぞれ $x_1, \ldots, x_k$ でラベル付けされているならば，$h \to x_1 \cdots x_k$ は P 中の生成規則である．

構文木 III

このように，構文木の各部分木は，その文の抽象化の具体例を表している．次に，構文木の「成果」を定義する．いま，任意の構文木の葉ノードに注目し，それらを左から右へ連結すると，ある文字列が得られる．この文字列をこの構文木の 成果 (*yield*) と呼ぶ．

構文木 III

このように，構文木の各部分木は，その文の抽象化の具体例を表している．次に，構文木の「成果」を定義する．いま，任意の構文木の葉ノードに注目し，それらを左から右へ連結すると，ある文字列が得られる．この文字列をこの構文木の 成果 (*yield*) と呼ぶ．

明らかに，重要な点は，ルートノードは開始記号でラベル付けされ，成果は **終端文字列 (*terminal string*)**，すなわち，すべての葉ノードは，T 中の文字あるいは空文字列でラベル付けされていることである．このように，文法が定める言語は，ルートノードに開始記号を，成果として終端文字列を有する構文木の成果の集合で表すこともできる．

構文木 IV

次の文法の一部を考えよう．これは代入文がどのように生成されるかを記述したものである．

例． 2

$$
\begin{array}{rcl}
\langle \texttt{assign} \rangle & \to & \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle & \to & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle & \to & \langle \texttt{id} \rangle + \langle \texttt{expr} \rangle \\
& \mid & \langle \texttt{id} \rangle * \langle \texttt{expr} \rangle \\
& \mid & (\langle \texttt{expr} \rangle) \\
& \mid & \langle \texttt{id} \rangle
\end{array}
$$

構文木 V

ここで，次の代入文に注目しよう．$\mathtt{A} := \mathtt{B} * (\mathtt{A} + \mathtt{C})$ これは，以下に示す 導出により生成される．

$$
\begin{aligned}
\langle \mathtt{assign} \rangle &\Rightarrow \langle \mathtt{id} \rangle := \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \langle \mathtt{id} \rangle * \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \mathtt{B} * \langle \mathtt{expr} \rangle \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\langle \mathtt{expr} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\langle \mathtt{id} \rangle + \langle \mathtt{expr} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\mathtt{A} + \langle \mathtt{expr} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\mathtt{A} + \langle \mathtt{id} \rangle) \\
&\Rightarrow \mathtt{A} := \mathtt{B} * (\mathtt{A} + \mathtt{C})
\end{aligned}
$$

構文木 V

ここで導出した代入文の構造は，以下の構文木として示される．

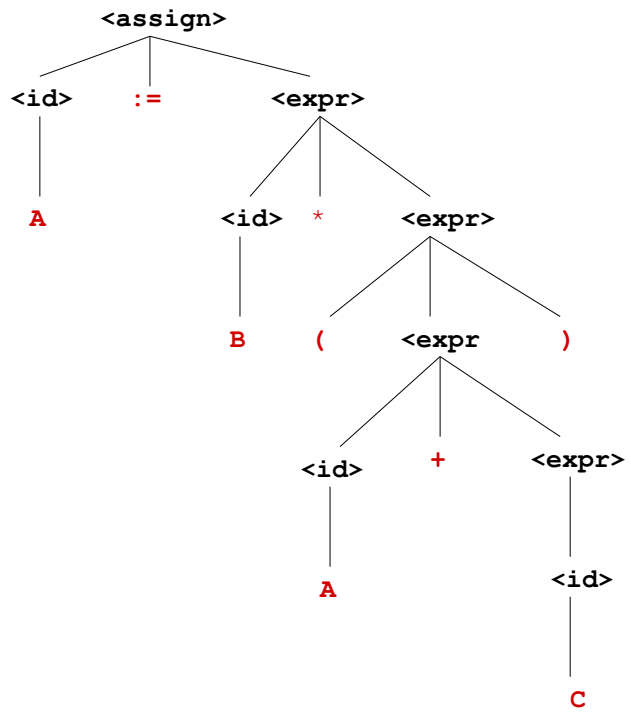

図 1: 文 A := B ∗ (A + C) の構文木

構文木 VI

プログラミング言語の構文解析器 (しばしばパーサ (*parsers*) と呼ばれる) は，所与のプログラムに関する構文木を構築する．いくつかのシステムは構文木を暗に構築するのみであるが，構文木から得られるすべての情報を構文解析の過程で利用する．構文木の構築の仕方には，ふたつの代表的なアプローチがある．ひとつは**トップダウン**，もうひとつは**ボトムアップ**である．トップダウン法では，構文木はルートノードから葉ノードへ向かって構築され，一方，ボトムアップ法では，葉ノードからルートノードへ向かって構築される．

構文木 VI

プログラミング言語の構文解析器 (しばしばパーサ (*parsers*) と呼ばれる) は，所与のプログラムに関する構文木を構築する．いくつかのシステムは構文木を暗に構築するのみであるが，構文木から得られるすべての情報を構文解析の過程で利用する．構文木の構築の仕方には，ふたつの代表的なアプローチがある．ひとつは**トップダウン**，もうひとつは**ボトムアップ**である．トップダウン法では，構文木はルートノードから葉ノードへ向かって構築され，一方，ボトムアップ法では，葉ノードからルートノードへ向かって構築される．

こうしたパーザを構成する際に考慮しなければならない大きな問題は**曖昧さ *(ambiguity)*** である．では，この問題について考えていこう．

曖昧さ I

定義 2

ふたつ以上の異なる構文木に対応する文を生成する文法は，曖昧である *(ambiguous)* と言われる．

曖昧さ I

定義 2

ふたつ以上の異なる構文木に対応する文を生成する文法は，曖昧である *(ambiguous)* と言われる．

例として，例 3に示す文法の次の部分について考えよう．一見，この文法は先の文法と非常に似ているように思える．唯一の違いは，生成規則であり，⟨id⟩ が ⟨expr⟩ に置き換えられている．

曖昧さ I

定義 2

ふたつ以上の異なる構文木に対応する文を生成する文法は，曖昧である *(ambiguous)* と言われる．

例として，例 3に示す文法の次の部分について考えよう．一見，この文法は先の文法と非常に似ているように思える．唯一の違いは，生成規則であり，⟨id⟩ が ⟨expr⟩ に置き換えられている．

しかし，このわずかな違いが深刻な問題を引き起こす．なぜなら，この文法は，例 2でみた文法より，わずかに弱い構文構造を与えるからである．

曖昧さ II

例. 3

$$
\begin{aligned}
\langle \texttt{assign} \rangle &\rightarrow \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle &\rightarrow \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle &\rightarrow \langle \texttt{expr} \rangle + \langle \texttt{expr} \rangle \\
&\mid \langle \texttt{expr} \rangle * \langle \texttt{expr} \rangle \\
&\mid (\langle \texttt{expr} \rangle) \\
&\mid \langle \texttt{id} \rangle
\end{aligned}
$$

この文法が曖昧であることを確かめるために，次の代入文を考えよう：

$$\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A} .$$

ふたつの形式的な導出は省略し，直接構文木を考える．

曖昧さ III

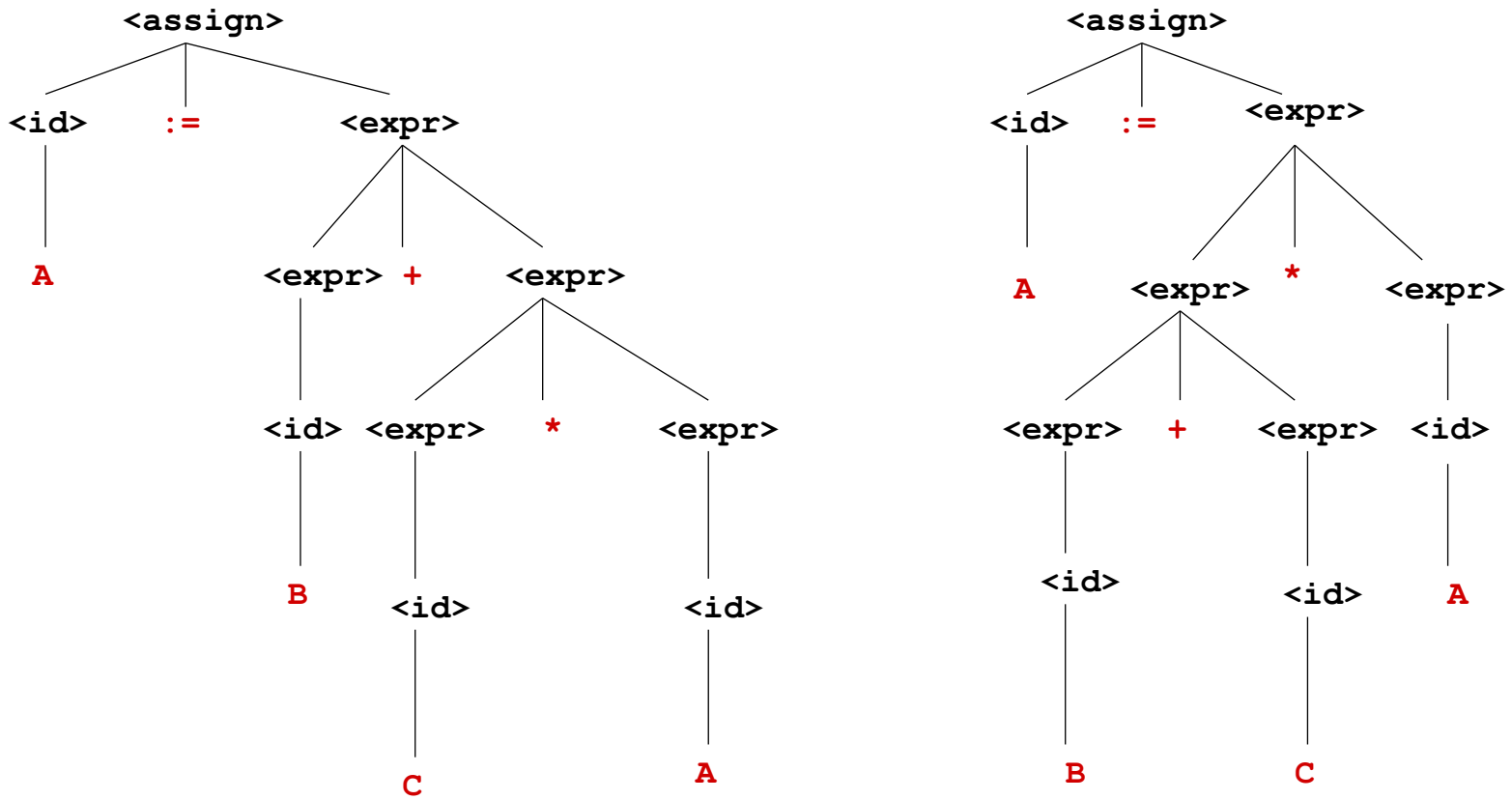

図 2: 文 $A := B + C * A$ に関する 2 つの構文木

$A := B + (C * A)$ $A := (B + C) * A.$

曖昧さ IV

これら 2 つの異なる構文木は問題を引き起こす．なぜなら，コンパイラは，文の意味をその構文構造に基づいて判断するからである．特に，コンパイラは，構文木を調べることによってどのコードを生成すべきかを決める．よって，この例では，意味 *(semantics)* が定まらない．この問題をもう少し詳細に見ていこう．

曖昧さ IV

これら 2 つの異なる構文木は問題を引き起こす．なぜなら，コンパイラは，文の意味をその構文構造に基づいて判断するからである．特に，コンパイラは，構文木を調べることによってどのコードを生成すべきかを決める．よって，この例では，意味 *(semantics)* が定まらない．この問題をもう少し詳細に見ていこう．図 2の左側の構文木では，積演算が木の下方 (深い方) で生成されている．これは，式中で積演算が和演算よりも優先することを示している．

曖昧さ IV

これら 2 つの異なる構文木は問題を引き起こす．なぜなら，コンパイラは，文の意味をその構文構造に基づいて判断するからである．特に，コンパイラは，構文木を調べることによってどのコードを生成すべきかを決める．よって，この例では，意味 *(semantics)* が定まらない．この問題をもう少し詳細に見ていこう．

図 2の左側の構文木では，積演算が木の下方 (深い方) で生成されている．これは，式中で積演算が和演算よりも優先することを示している．

しかし，図 2の二番目の構文木はその逆を示している．明らかに，コンパイラが下した決定に依存して，代入の実際の評価は，期待した通り (積演算が和演算に優先する) か，誤りかのいずれかになる．

曖昧さ V

例 2 の文法は曖昧ではないが，演算の優先順位が通常とは異なる．その文法では，積演算を含む文の構文木は，最深点で最右演算子が出現し，他の演算子は式の左へ行くに従い位置が上になる．

曖昧さ V

例 2 の文法は曖昧ではないが，演算の優先順位が通常とは異なる．その文法では，積演算を含む文の構文木は，最深点で最右演算子が出現し，他の演算子は式の左へ行くに従い位置が上になる．
よって，この問題を解消し，積と和の通常の優先順位，あるいは，より一般的には，任意の所望の優先順位を明確に定義する必要がある．実際の問題として，この目的はこの例題に対しては，異なる優先順位を持つ演算子のオペランドに対して別の非終端記号を用いることで達成できる．これは，非終端記号の追加だけでなく，生成規則の追加も必要とする．

曖昧さ VI

問

常に曖昧さを検出できるのか？
もし，曖昧さが検出できた場合，必ずそれを除去できるのか？

曖昧さ VI

問

常に曖昧さを検出できるのか？
もし，曖昧さが検出できた場合，必ずそれを除去できるのか？

その答えは次の定理によって与えられる．

曖昧さ VI

問

常に曖昧さを検出できるのか？
もし，曖昧さが検出できた場合，必ずそれを除去できるのか？

その答えは次の定理によって与えられる．

定理 1

任意の文脈自由文法が曖昧か否かを決定できるアルゴリズムは存在しない．

曖昧さ VI

問

常に曖昧さを検出できるのか？
もし，曖昧さが検出できた場合，必ずそれを除去できるのか？

その答えは次の定理によって与えられる．

定理 1

任意の文脈自由文法が曖昧か否かを決定できるアルゴリズムは存在しない．

定理 2

曖昧な文法だけを有する文脈自由言語が存在する．

曖昧さ VII

しかし，実際には想像する程深刻ではない．プログラム言語において典型的に現れる類の曖昧さについては，それらを除去するための多くの方法が提案されている．

曖昧さ VII

しかし，実際には想像する程深刻ではない．プログラム言語において典型的に現れる類の曖昧さについては，それらを除去するための多くの方法が提案されている．

では，先の例に戻り，曖昧さを除去する方法をみていこう．ここでは，例 2，および，例 3の文法と同じ言語を生成し，かつ，通常の積・和演算の優先順位を明らかに反映した文法を用いて議論を続ける．

曖昧さ VIII

例. 4

$$
\begin{array}{rcl}
\langle \texttt{assign} \rangle & \rightarrow & \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle & \rightarrow & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle & \rightarrow & \langle \texttt{expr} \rangle + \langle \texttt{term} \rangle \\
& \mid & \langle \texttt{term} \rangle \\
\langle \texttt{term} \rangle & \rightarrow & \langle \texttt{term} \rangle * \langle \texttt{factor} \rangle \\
& \mid & \langle \texttt{factor} \rangle \\
\langle \texttt{factor} \rangle & \rightarrow & (\langle \texttt{expr} \rangle) \\
& \mid & \langle \texttt{id} \rangle
\end{array}
$$

曖昧さ VIII

例. 4

$$
\begin{array}{rcl}
\langle \texttt{assign} \rangle & \rightarrow & \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \\
\langle \texttt{id} \rangle & \rightarrow & \texttt{A} \mid \texttt{B} \mid \texttt{C} \\
\langle \texttt{expr} \rangle & \rightarrow & \langle \texttt{expr} \rangle + \langle \texttt{term} \rangle \\
& \mid & \langle \texttt{term} \rangle \\
\langle \texttt{term} \rangle & \rightarrow & \langle \texttt{term} \rangle * \langle \texttt{factor} \rangle \\
& \mid & \langle \texttt{factor} \rangle \\
\langle \texttt{factor} \rangle & \rightarrow & (\langle \texttt{expr} \rangle) \\
& \mid & \langle \texttt{id} \rangle
\end{array}
$$

次に，先と同じ文，すなわち，次の文を導出してみよう．

$$\texttt{A} := \texttt{B} + \texttt{C} * \texttt{A} \,.$$

曖昧さ IX

その導出は，以下の通り，曖昧さ無しに得ることができる．

$$
\begin{aligned}
\langle \texttt{assign} \rangle &\Rightarrow \langle \texttt{id} \rangle := \langle \texttt{expr} \rangle \Rightarrow \texttt{A} := \langle \texttt{expr} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{expr} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{term} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{factor} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \langle \texttt{id} \rangle + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{term} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{term} \rangle * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{factor} \rangle * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \langle \texttt{id} \rangle * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \texttt{C} * \langle \texttt{factor} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \texttt{C} * \langle \texttt{id} \rangle \\
&\Rightarrow \texttt{A} := \texttt{B} + \texttt{C} * \texttt{A}
\end{aligned}
$$

曖昧さ X

さらに，先に最左導出 (*leftmost derivation*)，すなわち，最も左の非終端記号が，それが終端記号に置き換わるまで常に最初に処理される様子をみたことに注意してほしい．このように，与えられた文は，ひとつ以上の導出を有する．もし，最右非終端記号を常に最初に処理すると，それは最右導出 (*rightmost derivation*) となる．

曖昧さ X

さらに，先に最左導出 (*leftmost derivation*)，すなわち，最も左の非終端記号が，それが終端記号に置き換わるまで常に最初に処理される様子をみたことに注意してほしい．このように，与えられた文は，ひとつ以上の導出を有する．もし，最右非終端記号を常に最初に処理すると，それは最右導出 (*rightmost derivation*) となる．

明らかに，生成規則の適用が可能な非終端記号を任意に選択することも可能である．それを試し，構文木を構築してみよ．通常は，最左導出を暗に仮定した上で，より正確には，ある文脈自由文法が生成する言語中に，少なくともふたつの異なる最左導出を有する文が存在するとき，その文法は曖昧であると言うことができる．そうでない場合，その文法は無曖昧である (*unambiguous*) と言われる．ある言語に対して無曖昧な文法が存在するとき，その言語は無曖昧であると言われる．

曖昧さ XI

プログラミング言語を記述する際の別の重要な問題は，演算子が**結合的**であることを示すことである．数学で学んだ通り，和と積は結合的である．このことは，計算機における算術においても正しいだろうか？整数の和・積を考えるならば，それらは結合的である．しかし，浮動小数点計算は必ずしも結合的でない．よって，一般に，正しい結合性は本質的である．テキストでは，さらに情報を与える．

曖昧さ XI

プログラミング言語を記述する際の別の重要な問題は，演算子が**結合的**であることを示すことである．数学で学んだ通り，和と積は結合的である．このことは，計算機における算術においても正しいだろうか？整数の和・積を考えるならば，それらは結合的である．しかし，浮動小数点計算は必ずしも結合的でない．よって，一般に，正しい結合性は本質的である．テキストでは，さらに情報を与える．

練習問題として，多くのプログラミング言語にみられる
if-then-else 文の無曖昧な文法を考えてみるとよい．

曖昧さ XI

プログラミング言語を記述する際の別の重要な問題は，演算子が**結合的**であることを示すことである．数学で学んだ通り，和と積は結合的である．このことは，計算機における算術においても正しいだろうか？整数の和・積を考えるならば，それらは結合的である．しかし，浮動小数点計算は必ずしも結合的でない．よって，一般に，正しい結合性は本質的である．テキストでは，さらに情報を与える．

練習問題として，多くのプログラミング言語にみられる
if-then-else 文の**無曖昧な**文法を考えてみるとよい．

文脈自由文法の性質についてさらに続けよう．

分離的文法 I

文脈自由文法においては，生成規則の右辺のサイズに**あらかじめ定められた**制限はない．このことは多くの証明を複雑なものにするかもしれない．しかし，右辺の長さを高々 2 に制限した**文脈自由文法の標準形**が存在する．これを適用すると，多くの証明が大幅に簡単化される．

まず，分離的文法の概念を定義する．

分離的文法 I

文脈自由文法においては，生成規則の右辺のサイズに**あらかじめ定められた**制限はない．このことは多くの証明を複雑なものにするかもしれない．しかし，右辺の長さを高々 2 に制限した**文脈自由文法の標準形**が存在する．これを適用すると，多くの証明が大幅に簡単化される．

まず，分離的文法の概念を定義する．

定義 3

文法 $\mathcal{G} = [T, N, \sigma, P]$ を考える．任意の $(\alpha \rightarrow \beta) \in P$ について，$\alpha, \beta \in N^*$，あるいは，$\alpha \in N$ かつ $\beta \in T$ であるとき，$\mathcal{G}$ は**分離的である *(separated)*** と言われる．

分離的文法 I

文脈自由文法においては，生成規則の右辺のサイズに**あらかじめ定められた**制限はない．このことは多くの証明を複雑なものにするかもしれない．しかし，右辺の長さを高々 2 に制限した**文脈自由文法の標準形**が存在する．これを適用すると，多くの証明が大幅に簡単化される．

まず，分離的文法の概念を定義する．

定義 3

文法 $\mathcal{G} = [T, N, \sigma, P]$ を考える．任意の $(\alpha \rightarrow \beta) \in P$ について，$\alpha, \beta \in N^*$，あるいは，$\alpha \in N$ かつ $\beta \in T$ であるとき，$\mathcal{G}$ は**分離的である *(separated)*** と言われる．

定理 3

任意の文脈自由文法 $\mathcal{G} = [T, N, \sigma, P]$ について，それと等価で分離的な文脈自由文法 $\mathcal{G}' = [T, N', \sigma, P']$ が存在する．

分離的文法 II

証明： まず，各 $t \in T$ について，新たな非終端記号 h_t を導入する．ここで，任意の $t \in T$ について $h_t \notin N$ であるとする．さらに，$N' = \{h_t \mid t \in T\} \cup N$ とする．

分離的文法 II

証明： まず，各 $t \in T$ について，新たな非終端記号 h_t を導入する．ここで，任意の $t \in T$ について $h_t \notin N$ であるとする．さらに，$N' = \{h_t \mid t \in T\} \cup N$ とする．
次に，$(\alpha \to \beta) \in P$ について，β 中の各終端記号 t を h_t に置き換えて得られる生成規則を $(\alpha \to \beta)[t/\!/h_t]$ と表す．生成規則の集合 P' は次の通り定義される．

$$P' = \{(\alpha \to \beta)[t/\!/h_t] \mid (\alpha \to \beta) \in P\} \cup \{h_t \to t \mid t \in T\}.$$

分離的文法 II

証明： まず，各 $t \in T$ について，新たな非終端記号 h_t を導入する．ここで，任意の $t \in T$ について $h_t \notin N$ であるとする．さらに，$N' = \{h_t \mid t \in T\} \cup N$ とする．
次に，$(\alpha \to \beta) \in P$ について，β 中の各終端記号 t を h_t に置き換えて得られる生成規則を $(\alpha \to \beta)[t/\!/h_t]$ と表す．生成規則の集合 P' は次の通り定義される．

$$P' = \{(\alpha \to \beta)[t/\!/h_t] \mid (\alpha \to \beta) \in P\} \cup \{h_t \to t \mid t \in T\}.$$

構成の仕方から，$\mathcal{G}'$ が分離的であることは直ちにわかる．さらに，$\mathcal{G}'$ が文脈自由であることも保証される．

分離的文法 III

$L(\mathcal{G}) = L(\mathcal{G}')$ であることを示す.

主張 1. $L(\mathcal{G}) \subseteq L(\mathcal{G}')$.

$s \in L(\mathcal{G})$ とすると，次の導出が存在する.

$$\sigma \overset{1}{\Rightarrow} w_1 \overset{1}{\Rightarrow} w_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n \overset{1}{\Rightarrow} s\,,$$

ここで，$w_1, \ldots, w_n \in (N \cup T)^+ \setminus T^*$，かつ，$s \in T^*$ である．w_i, $i = 1, \ldots, n$ を生成する際に用いた生成規則を P_i としよう.

分離的文法 III

$L(\mathcal{G}) = L(\mathcal{G}')$ であることを示す．

主張 1. $L(\mathcal{G}) \subseteq L(\mathcal{G}')$.

$s \in L(\mathcal{G})$ とすると，次の導出が存在する．

$$\sigma \overset{1}{\Rightarrow} w_1 \overset{1}{\Rightarrow} w_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n \overset{1}{\Rightarrow} s\,,$$

ここで，$w_1, \ldots, w_n \in (N \cup T)^+ \setminus T^*$，かつ，$s \in T^*$ である．w_i, $i = 1, \ldots, n$ を生成する際に用いた生成規則を P_i としよう．

このとき，次の通り P' の生成規則を用いて s を生成することができる．$s = s_1 \cdots s_m$ とする．ここで，任意の $j = 1, \ldots, m$ について，$s_j \in T$ である．P_i を適用する代わりに，P' 中の $P_i[t/\!/h_t]$ を用いて次を得る．

$$\sigma \overset{1}{\Rightarrow} w'_1 \overset{1}{\Rightarrow} w'_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w'_n \overset{1}{\Rightarrow} h_{s_1} \cdots h_{s_m}\,,$$

ここで，任意の $i = 1, \ldots, n$ について，$w'_i \in (N')^+$ である．

分離的文法 III

$L(\mathcal{G}) = L(\mathcal{G}')$ であることを示す．

主張 1. $L(\mathcal{G}) \subseteq L(\mathcal{G}')$.

$s \in L(\mathcal{G})$ とすると，次の導出が存在する．

$$\sigma \overset{1}{\Rightarrow} w_1 \overset{1}{\Rightarrow} w_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w_n \overset{1}{\Rightarrow} s\,,$$

ここで，$w_1, \ldots, w_n \in (N \cup T)^+ \setminus T^*$，かつ，$s \in T^*$ である．w_i, $i = 1, \ldots, n$ を生成する際に用いた生成規則を P_i としよう．

このとき，次の通り P' の生成規則を用いて s を生成することができる．$s = s_1 \cdots s_m$ とする．ここで，任意の $j = 1, \ldots, m$ について，$s_j \in T$ である．P_i を適用する代わりに，P' 中の $P_i[t/\!/h_t]$ を用いて次を得る．

$$\sigma \overset{1}{\Rightarrow} w'_1 \overset{1}{\Rightarrow} w'_2 \overset{1}{\Rightarrow} \cdots \overset{1}{\Rightarrow} w'_n \overset{1}{\Rightarrow} h_{s_1} \cdots h_{s_m}\,,$$

ここで，任意の $i = 1, \ldots, n$ について，$w'_i \in (N')^+$ である．このように，s を得るためには，$j = 1, \ldots m$ について生成規則 $h_{s_j} \to s_j$ を適用すれば十分である．これで主張 1 が証明された．

分離的文法 IV

主張 2. $L(\mathcal{G}') \subseteq L(\mathcal{G})$.
これは，主張 1 の証明での方法を逆にすることで同様に証明できる．∎

チョムスキー標準形 I

これで文脈自由文法の標準形を定義する準備ができた.

定義 4

文法 $\mathcal{G} = [T, N, \sigma, P]$ において，P のすべての生成規則が $h \rightarrow h_1 h_2$ の形式であるとき，**チョムスキー標準形 (*Chomsky normal form*)** であると言われる．ここで，$h, h_1, h_2 \in N$，あるいは，$h \rightarrow x$ であり，$h \in N$ とする $x \in T$.

チョムスキー標準形 I

これで文脈自由文法の標準形を定義する準備ができた.

定義 4

文法 $\mathcal{G} = [T, N, \sigma, P]$ において，P のすべての生成規則が $h \to h_1 h_2$ の形式であるとき，***チョムスキー標準形 (Chomsky normal form)*** であると言われる．ここで，$h, h_1, h_2 \in N$，あるいは，$h \to x$ であり，$h \in N$ とする $x \in T$.

定義から直ちに次の系を得る.

系 4

$\mathcal{G} = [T, N, \sigma, P]$ をチョムスキー標準形の文法とする．このとき，

(1) $\mathcal{G}$ は文脈自由である.

(2) $\mathcal{G}$ は λ-自由である.

(3) $\mathcal{G}$ は分離的である.

チョムスキー標準形 II

いよいよ以下を示すことができる：

定理 5

$\lambda \notin L(\mathcal{G})$ である任意の文脈自由文法 $\mathcal{G} = [T, N, \sigma, P]$ について，それと等価なチョムスキー標準形の文法 $\mathcal{G}'$ が存在する．

チョムスキー標準形 II

いよいよ以下を示すことができる：

定理 5

$\lambda \notin L(\mathcal{G})$ である任意の文脈自由文法 $\mathcal{G} = [T, N, \sigma, P]$ について，それと等価なチョムスキー標準形の文法 $\mathcal{G}'$ が存在する．

証明： $\mathcal{G} = [T, N, \sigma, P]$ が与えられたとしよう．一般性を失うことなく，$\mathcal{G}$ は既約であると仮定できる．

チョムスキー標準形 III

最初に，$h \to h'$ なる形式の生成規則をすべて除去する．これは，次の通りに行なうことができる．

$$\text{任意の } h \in N \text{ について } W_0(h) = \{h\}$$

とし，任意の $i \geqslant 0$ について以下を定義する．

$$W_{i+1}(h) = W_i \cup \{\tilde{h} \mid \tilde{h} \in N, \text{ ある } \hat{h} \in W_i(h) \text{ について } (\hat{h} \to \tilde{h}) \in P\}.$$

チョムスキー標準形 III

最初に，$h \to h'$ なる形式の生成規則をすべて除去する．これは，次の通りに行なうことができる．

$$\text{任意の } h \in N \text{ について } W_0(h) = \{h\}$$

とし，任意の $i \geqslant 0$ について以下を定義する．

$$W_{i+1}(h) = W_i \cup \{\tilde{h} \mid \tilde{h} \in N, \text{ ある } \hat{h} \in W_i(h) \text{ について } (\hat{h} \to \tilde{h}) \in P\}.$$

このとき，以下の事実は明らかである：

(1) 任意の $i \geqslant 0$ について，$W_i(h) \subseteq W_{i+1}(h)$

(2) もし $W_i(h) = W_{i+1}(h)$ ならば，任意の $m \in \mathbb{N}$ について $W_i(h) = W_{i+m}(h)$，

(3) $n = \mathrm{card}(N)$ について $W_n(h) = W_{n+1}(h)$，

(4) $W_n(h) = \{B \mid B \in N \text{ かつ } h \overset{*}{\Rightarrow} B\}$.

チョムスキー標準形 IV

ここで，次を定義する．

$$P_1 = \{h \to \gamma \mid h \in N,\ \gamma \notin N,\ \text{ある } B \in W_n(h) \text{ について } (B \to \gamma) \in P\}.$$

$\mathcal{G}_1 = [T, N, \sigma, P_1]$ とすると，

チョムスキー標準形 IV

ここで，次を定義する．

$$P_1 = \{h \to \gamma \mid h \in N,\ \gamma \notin N,\ \text{ある } B \in W_n(h) \text{ について } (B \to \gamma) \in P\}.$$

$\mathcal{G}_1 = [T, N, \sigma, P_1]$ とすると，構成の仕方から，P_1 は $h \to h'$ なる形式の生成規則を**含まない**．こうした生成規則は $h \to \gamma$ で置き換えられている．すなわち，もし P 中の生成規則と $B \to \gamma$ を用いて $h \overset{*}{\Rightarrow} B$ を得たならば，P_1 中に $h \to \gamma$ なる生成規則が存在する．P_1 はもとのすべての生成規則 $(h \to \gamma) \in P$ を含むことにも注意しよう．ここで，W_0 の定義より $\gamma \notin N$ である．
$L(\mathcal{G}_1) = L(\mathcal{G})$ を形式的に確かめることは，演習問題として残しておく．

チョムスキー標準形 V

次に，上述したアルゴリズムを用いて，G_1 から，等価な分離的文法 G_2 を構成する．ここで，まだ修正が必要な P_2 中の生成規則は次の形式である．

$$h \rightarrow h_1 h_2 \cdots h_n \text{ , ここで } n \geqslant 3 \text{ .}$$

チョムスキー標準形 V

次に，上述したアルゴリズムを用いて，G_1 から，等価な分離的文法 G_2 を構成する．ここで，まだ修正が必要な P_2 中の生成規則は次の形式である．

$$h \to h_1 h_2 \cdots h_n \text{ , ここで } n \geqslant 3 \text{ .}$$

任意のこうした生成規則を次の生成規則で置き換える．

$$\begin{aligned} h &\to h_1 h_{h_2 \cdots h_n} \\ h_{h_2 \cdots h_n} &\to h_2 h_{h_3 \cdots h_n} \\ &\vdots \\ h_{h_{n-1} h_n} &\to h_{n-1} h_n \end{aligned}$$

チョムスキー標準形 V

次に，上述したアルゴリズムを用いて，G_1 から，等価な分離的文法 G_2 を構成する．ここで，まだ修正が必要な P_2 中の生成規則は次の形式である．

$$h \to h_1 h_2 \cdots h_n \text{ , ここで } n \geqslant 3 .$$

任意のこうした生成規則を次の生成規則で置き換える．

$$\begin{aligned} h &\to h_1 h_{h_2 \cdots h_n} \\ h_{h_2 \cdots h_n} &\to h_2 h_{h_3 \cdots h_n} \\ &\vdots \\ h_{h_{n-1} h_n} &\to h_{n-1} h_n \end{aligned}$$

よって，結果として得られる文法 G' はチョムスキー標準形であり，構成法から G と等価である．なお詳細については省略する．∎

反復補題 I

チョムスキー標準形を用いて証明可能な重要な結果を示して今回の講義を終える．この結果は，通常は，文脈自由言語の反復補題，Bar-Hillel の補題，あるいは qrsuv-定理と言われる．

反復補題 I

チョムスキー標準形を用いて証明可能な重要な結果を示して今回の講義を終える．この結果は，通常は，文脈自由言語の反復補題，Bar-Hillel の補題，あるいは qrsuv-定理と言われる．

qrsuv-定理は，言語が文脈自由であるための必要条件を与えていることに注意しよう．それは十分条件ではない．よって，これは，言語が文脈自由でないことを示すためにしばしば利用される．

反復補題 II

定理 6 (qrsuv-定理，反復補題)

任意の文脈自由言語 L について，整数 k, ℓ が存在し，$|w| \geqslant k$ なる任意の $w \in L$ について以下の文字列 q, r, s, u, v が存在する．

(1) $w = qrsuv$,

(2) $|rsu| \leqslant k$,

(3) $ru \neq \lambda$,

(4) 任意の $i \in \mathbb{N}$ について $qr^i su^i v \in L$.

反復補題 II

定理 6 (qrsuv-定理，反復補題)

任意の文脈自由言語 L について，整数 k, ℓ が存在し，$|w| \geqslant k$ なる任意の $w \in L$ について以下の文字列 q, r, s, u, v が存在する．

(1) $w = qrsuv$,

(2) $|rsu| \leqslant k$,

(3) $ru \neq \lambda$,

(4) 任意の $i \in \mathbb{N}$ について $qr^i su^i v \in L$.

証明： $\mathcal{G} = [T, N, \sigma, P]$ を，L に関する任意のチョムスキー標準形の文脈自由文法とする．このとき，すべての構文木は，最後の導出ステップを除いて二分木となる．

反復補題 III

$n = \mathrm{card}(N)$ および $k = 2^n$ とし，$|w| \geqslant k$ なる任意の文字列 $w \in L(\mathcal{G})$ の構文木を考える．すると，w の任意の構文木は少なくとも n の高さを持つから，根ノード σ からある葉ノードへの長さ n 以上のパスが必ず存在する．このパス上には，σ を含めて $n+1$ の非終端記号が含まれるので，ある 非終端記号 h は少なくとも 2 回現れることになる (図 3，Part (a) を参照).

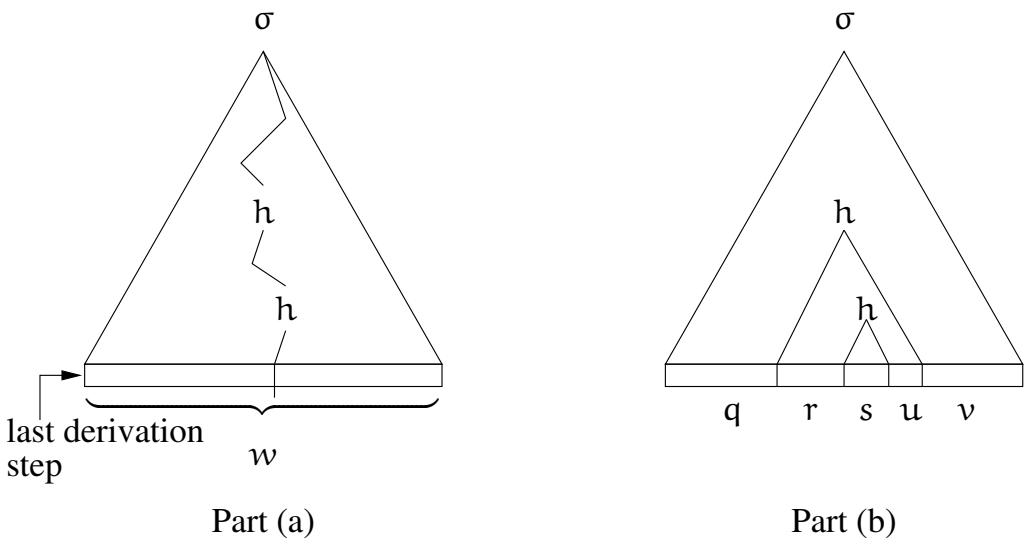

図 3: qrsuv-定理を例証する構文木

反復補題 IV

いま，葉ノードから上に向かって進みながら，h を 2 回含むあるパスにおいて，その最初 2 つの h を固定する．このようにして，上方の h が葉ノードから高々 n ステップの位置にあることを保証する．

反復補題 IV

いま，葉ノードから上に向かって進みながら，h を 2 回含むあるパスにおいて，その最初 2 つの h を固定する．このようにして，上方の h が葉ノードから高々 n ステップの位置にあることを保証する．

次に，これら 2 つの h から作られる部分文字列を考える．これが部分文字列 qrsuv となる (図 3，Part (b) を参照).

反復補題 IV

いま，葉ノードから上に向かって進みながら，h を 2 回含むあるパスにおいて，その最初 2 つの h を固定する．このようにして，上方の h が葉ノードから高々 n ステップの位置にあることを保証する．

次に，これら 2 つの h から作られる部分文字列を考える．これが部分文字列 $qrsuv$ となる (図 3，Part (b) を参照)．

$\mathcal{G}$ は CNF であるから，上方の h において，$h \to h_1 h_2$ なる形式の生成規則が適用されなければならない．よって，$r \neq \lambda$ あるいは $u \neq \lambda$ であり，言明 (3) が示される．

反復補題 V

言明 (2)は，上方の h が葉ノードから高々 n ステップであることからわかる．よって，この h から導出される文字列 rsu は，高々 $2^n = k$ の長さとなる．

反復補題 V

言明 (2)は，上方の h が葉ノードから高々 n ステップであることからわかる．よって，この h から導出される文字列 rsu は，高々 $2^n = k$ の長さとなる．

あとは言明 (4)を示すのみである．構文木を修正することで証明できる．まず，上方の h を根ノードとする部分木を削除し，それを，下方の h を根ノードとする部分木と置き換える (図 4, Part (a) 参照)．こうすることで，qsv の導出，すなわち，$qr^0su^0v \in L(\mathcal{G})$ が得られる．

反復補題 VI

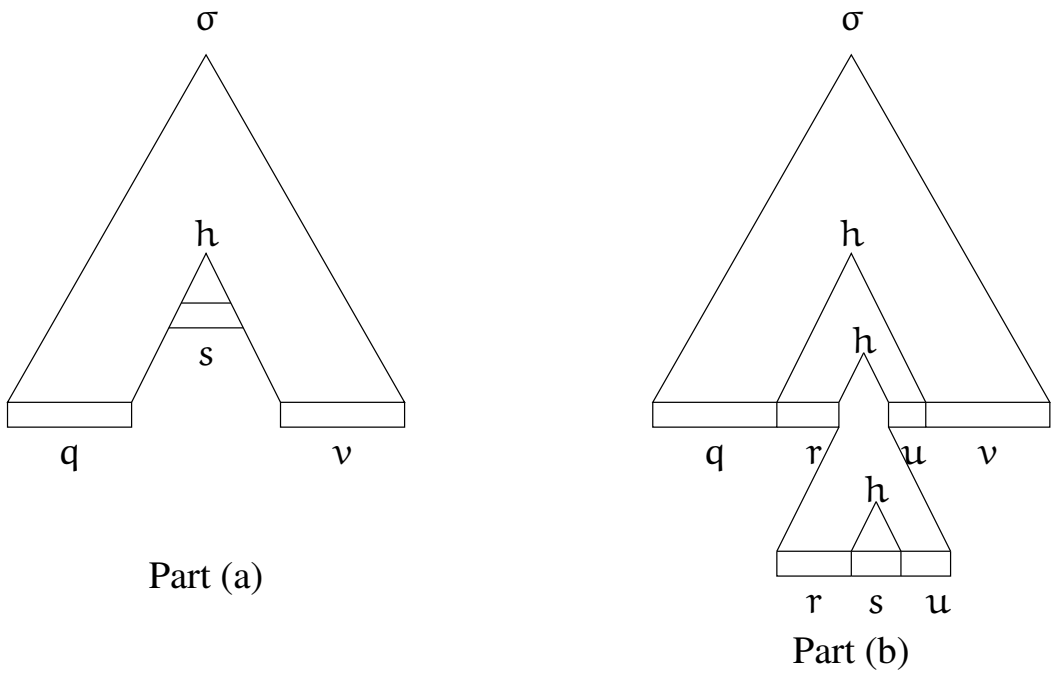

図 4: qrsuv-定理の言明 (4) の例示

反復補題 VII

次に，下方の h を根ノードとする部分木を削除し，上方の h を根ノードとする部分木と置き換える．(図 4，Part (b) 参照)．こうすることで，qr^2su^2v の導出が得られ，よって，$qr^2su^2v \in L(\mathcal{G})$ となる．この考え方を繰り返すことで，すべての $i \in \mathbb{N}^+$ について，$qr^isu^iv \in L(\mathcal{G})$ を得る．

反復補題 VII

次に，下方の h を根ノードとする部分木を削除し，上方の h を根ノードとする部分木と置き換える．(図 4，Part (b) 参照)．こうすることで，qr^2su^2v の導出が得られ，よって，$qr^2su^2v \in L(\mathcal{G})$ となる．この考え方を繰り返すことで，すべての $i \in \mathbb{N}^+$ について，$qr^isu^iv \in L(\mathcal{G})$ を得る．

定理 6の応用を考えるために，定理 6.5 の証明の完成を兼ねて，次の定理を示す．

反復補題 VIII

定理 7

$L = \{a^n b^n c^n \mid n \in \mathbb{N}\} \notin \mathcal{CF}$.

証明： 否定を仮定する．すると，定理 6より，$|w| \geqslant k$ なるすべての $w \in L$ について，以下となるような k が存在する：

$|ru| > 0$ なる部分文字列 q, r, s, u, v が存在し，すべての $w = qrsuv \in L$ およびすべての $i \in \mathbb{N}$ について，$qr^i su^i v \in L$ でなければならない．いま，$w = a^k b^k c^k$ を考える．すると，$|w| = 3k > k$ であり，定理 6の言明 (2) より，$|rsu| \leqslant k$ となる．よって，ru は a，b および c を含むことができない．$|ru| > 0$ および ($i = 0$ における) 主張 (4) より，$qr^0 su^0 v = qsv \in L$ となるが，先に議論した通り，すべての $\ell \in \mathbb{N}$ について，$qsv \neq a^\ell b^\ell c^\ell$ であるから，矛盾が生じる． ∎

Thank you!