2010年度 計算論講義ノート (近山先生)

電気電子工学科 03100509 中谷 翔

2010年10月-2011年2月

目次

第 1 章

はじめに

1.1 この資料について

東京大学工学部電子情報工学科の 3 年冬学期に行われた授業,「計算論」の講義ノートです.

1.2 資料の方針

- 授業でやった内容はカバーします.
- 授業でやってないことは基本的にやりません.
- ただし，授業では扱ってないが，理解が深まりそうな例とかは扱います.
- 重要そうなワードは 赤字 にしてます．赤シート使って勉強すれば単位が来るかは知りません.

1.3 作った動機

最も大きな動機は，LaTeX の練習です．色々なスタイルやマクロを試してみたかったので．ソースも公開していますので，そちらもよろしければご参照ください.
`https://github.com/laysakura/TheoryOfComputation`
あと，個人的に内容にとても興味が持てました．特にコンピュータを触る人にとっては面白い話だと思います．だからこの講義で LaTeX の練習をすることにしたわけです.

1.4 お願い

この資料も不完全な部分が多いかと思います．もしもご意見などくださるのであれば，`lay.sakura@gmail.com` (中谷) までよろしくお願いします.

第2章

オートマトンと言語理論

2.1 計算とは

2.1.1 計算の仕様

計算とは，以下のような仕様を持つものである．

$$\text{仕様 } S : \mathbb{N} \to \mathbb{N}$$

つまり，「ある自然数をある自然数に対応させるのが計算」と言える．

この講義の文脈では，「ある自然数」を 0/1 の有限列で表す (何進数かは本質的ではない).

この仕様の性質から，以下のことが言える．

- 仕様も所詮自然数から自然数の対応なので，仕様自体も自然数で表現できる．
  ex:「000 → 011」という仕様を「1100」と表す (インデックスをつける)
- あらゆる仕様の集合は連続体濃度． [*1]
- プログラムは仕様に含まれるから，プログラムは加算集合. [*2]

2.1.2 計算の能力

実は出力が N ビットの計算も，出力 1 ビットの計算に帰着させることができる．例えば,

$$\text{計算 } A : \text{入力 } 010, \text{出力 } 110$$

は,

$$\text{計算 } B_1 : \text{入力 } 010, \text{出力 } 1$$
$$\text{計算 } B_2 : \text{入力 } 010, \text{出力 } 1$$
$$\text{計算 } B_3 : \text{入力 } 010, \text{出力 } 0$$

を逐次的に実行する計算 B に置き換えることができる．

こうして，出力を 1 ビットにできるよというと，この先の講義でよく出てくるオートマトンの話がちょっとありがたく思える．

[*1] よく分からないけど $\mathbb{R}$ の濃度らしい [**?**]

[*2] プログラムで記述できるのは，仕様の集合のごく一部

出力が 1 ビットということは，出力は真偽値をとると考えられる．

オートマトンは，「入力を受理するかどうか」を判定する．この真偽の判定を，0 を出力するか 1 を出力するかに対応させると，オートマトンは (ある程度) どのような計算でもできるということがわかる．

2.2 言語 (Language)

言語 とは，有限種の記号列の有限列の集合と定義できるが，その定義を理解するために必要な用語をまず下に記す．

Σ

用いる記号の有限集合． アルファベット [*3] ともいう．

例 2.2.1: 英字と数字を記号とするアルファベット

$$\Sigma = \{a, b, c, \cdots, z, A, B, C, \cdots, Z, 0, 1, \cdots, 9\}$$

記号列 (string)

記号の有限の並び． [*4]

例 2.2.2: 上の例での Σ 上の記号列

$$\varepsilon, a, b, c, \cdots, 9, aa, ab, \cdots, afbafe132, \cdots$$

空列 ϵ(下記) を含むことに注意．

長さ (length)

記号列中の記号の数．$|w|$ が w の長さを表す．

例 2.2.3: ある記号列の長さ

$$|afbafe132| = 9$$

空列 (empty string)

長さ 0 の記号列． ε で表す．

ここにきて， 言語 の定義，「有限種の記号列の有限列の集合」を見直すと，何となく分かるのではないかと思う．

[*3] 別に $abc\cdots zABC\cdots Z$ 以外でも，記号ならなんでも OK.「鬱」を記号に含むアルファベットも構成可

[*4] 言語論で「並び」というと，順序に意味のある列という意味.「集合」は，順序に意味のない列.

例 2.2.4: ある記号列集合で定義される言語

以下のような，英字 2 文字で成り立つような記号列の集合を考える．

$$aa, ab, ac, \cdots, ba, bb, bc, \cdots, zx, zz$$

この記号列の集合によって定義される言語には，例えば以下のようなものがある．

$$\Sigma_1 = \{aa\}$$

$$\Sigma_2 = \{ex, gy\}$$

$$\Sigma_3 = \{an, ax, if, up\}$$

ここで，今後の話でよく出てくる，Σ^* という言語を見てみる．

Σ^*

Σ 上の記号列全てを含む集合．これは，言語になっている．

例 2.2.5: アルファベット $\{a\}$ で定義される言語

いま，単純なアルファベット

$$\Sigma = \{a\}$$

を考える．このアルファベット上の記号列は，

$$\varepsilon, a, aa, \cdots$$

であるので，Σ に対しては

$$\Sigma^* = \{\varepsilon, a, aa, \cdots\}$$

が定義される．

2.2.1 言語で定義される言語

連接 (concatenation)

2 つの言語 L_1 と L_2 の連接は，

$$L_1L_2 = \{vw | v \in L_1, w \in L_2\}$$

で定義される．

例 2.2.6: 単純な連接

$$abc \in L_1, \quad defg \in L_2$$

であれば，

$$abcdefg \in L_1L_2$$

である．ただし，L_1L_2 は abc や $defg$ を含まないことに注意．

閉包 (closure)

Σ 上の言語 $L(\subset \Sigma^*)$ の要素をいくつでも並べたもの.

$$L^0 = \{\epsilon\}, \quad L^1 = \{w | w \in L\}, \quad L^2 = \{vw | v, w \in L\}, \quad \cdots$$

とするとき，閉包は

$$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots = \cup_{i=0}^{\infty} L^i$$

と定義される.

例 2.2.7: 単純な閉包

$$L = \{abc, efg, hij\}$$

であったとき,

$$L^* = \{\epsilon, abc, abcabc, \cdots, abcefg, \cdots, hijabcefg, \cdots\}$$

などである.

正閉包 (positive closure)

$$L^+ = \cup_{i=1}^{\infty} L^i$$

で定義される.

2.3 正規表現 (Regular expression, RE)

Σ 上の正規表現[*5]とは，以下のいずれかである.

- ϕ : 表す言語 (=記号列の集合) は空集合.
- 空列 ε : 表す言語は $\{\varepsilon\}$
- Σ の各要素 a に対して a : 表す言語は a
- r, s がそれぞれ言語 R, S を表す正規表現であるとき,
  - $(r + s)$: $R \cup S$ を表す.
  - (rs) : RS を表す.
  - $(r*)$: R^* を表す.

[*5] 言語論の業界では RE と略すようだが，コンピュータの業界では regex とか regexp とか略すことも多い.

2.4 有限オートマトン (Finite automaton, FA)

有限オートマトン [*6] (以下， FA) とは，有限の記憶を持つ計算機構の一種である．より具体的には，ある入力列を走査し，その入力列を受理するかを判定するものと言える．

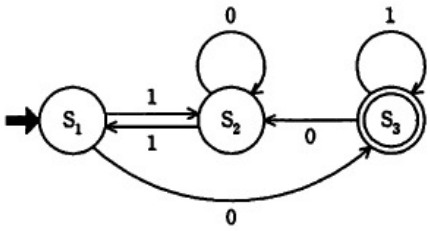


図 2.1　FA を表す遷移図

例 2.4.1:　簡単な FA

図 2.1 は，FA を表す一つの例である．このように，FA は 状態遷移図 で表せる．まずは，この FA の動作の概略を説明する．

- FA は，開始状態 (initial state) で開始する．今回の開始状態は S_1.
- 入力の 1 ビット目により，次に遷移する状態が変わる．0 だったら S_3 に，1 だったら S_2 に移る．
- 同様に 2 ビット目，$\cdots$ と入力列を読んでいき，読み終わった時点の状態，すなわち最終状態 (final state) が S_3 であれば，この FA は入力列を 受理 (accept) したといい，S_3 でなければ受理しなかったという．

ここで，実際に入力列 0100 を入れたときの動作を追ってみる．

- 開始状態 S_0 へ.
- 0 を受け取り，S_3 へ.
- 1 を受け取り，S_3 へ.
- 0 を受け取り，S_2 へ.
- 0 を受け取り，S_2 へ.
- 入力列を読み終わったが，S_3 でなく S_2 にいるので，入力列 0100 は受理しない.

ここで一度，2.1.2 の記述を確認し，(FA に限らず) オートマトンの重要性を確認して欲しい．

[*6] 有限状態オートマトン (finite state automaton) と呼ぶこともある

2.4.1 FA の定式化

FA の能力，動作などを真面目に議論しようとすると，どうしても定式化する必要が出てくる．以下のように定式化されることが多い．

$M = (Q, \Sigma, \delta, q_0, F)$
: ある FA を表す式．

Q
: 状態の有限集合．上の例で言うと，S_1, S_2, S_3

Σ
: 入力記号の有限集合．上の例で言うと，$0, 1$

q_0
: 初期状態 $\in Q$．上の例で言うと，S_0

F
: 最終状態の集合 $\subset Q$．ここで，最終状態は 1 つでなくとも良いことに注意．上の例で言うと，S_2 のみ．

δ
: 遷移関数 (transition fucntion)．数式でいうと，

$$\delta : \quad Q \times \Sigma \to Q \qquad \cdots\cdots\cdots\cdots\cdots\cdots \text{(2.1)}$$

ここで，$\times$ 記号は直積[*7]を表す．すなわち，

$$Q \times \Sigma = \{(q, s) | q \in Q, s \in \Sigma\}$$

である．したがって，式 2.1 の表すところは，「ある状態 q において，次の入力記号が s であった場合，次の状態は Q に含まれるいずれかである．[*8]といったものである．

2.4.2 オートマトンの動作

以下の動作は，FA のみならず，オートマトン一般に成り立つことに注意．

- 初期状態 q_0 から初めて，入力 1 記号毎に状態遷移．
- 「入力が終わったときに最終状態の 1 つにある」$\Leftrightarrow$「オートマトンは入力を受理した」
- オートマトンを定義すると，それが受理する入力列の集合 (オートマトンの受理言語) が決まる．

[*7] Wikipedia[**?**] が意外とわかりやすいので，興味があれば調べてみてください．

[*8] ここで「いずれか」とはいえど，一意に定まっている．例えば，「状態 q_1 で入力 a を受け取ったら，状態 q_3 に移る．ただし，$q_i \in Q, a \in S$」ということ．下記の NFA の話を読むと気になるところだと思うので．

特に，FA の受理言語を 正則集合 (regular set) という．

2.5　FA と正規表現の関係

FA は計算機構であり，正規表現は文法であるが，どちらも言語を与える (前者は受理言語により)．両者の与える言語について，以下の定理が成り立つ．

- FA の受理言語は，正規表現で表せる．
- 正規表現が表す言語を受理する FA を構成できる．

この定理は授業中にも証明しておりません．気になる人は，「有限オートマトン正規表現 証明」とかでググると結構出てきます．気にならない人は「正規表現は FA と同程度の表現力なのか」と覚えておけば良いと思います．

2.6　非決定性有限オートマトン (Non-deterministic finite automaton, NFA)

非決定性有限オートマトン (以下， NFA) は，同じ状態にあり，同じ入力記号を受け取る時でも，次の状態が一意には決まらないような FA である．

2.6.1　NFA の定式化

基本的に，2.4.1 でしたのと同じ定式化ができる．ただし，遷移関数は異なり，以下のように定式化される．

$$\delta : \quad Q \times \Sigma \to 2^Q \qquad \cdots\cdots (2.2)$$

ここで，2^X は，冪集合 (power set) を表す記号である．[*9]2^X は，すべての「集合 X の部分集合」を元として集めた集合系 (集合族) のことである．[*10]

したがって，式 2.2 の意味するところは，「Q 中の状態にいるとき，Σ 上の記号を受け取ると，Q 上のいくつかの状態のうちの 1 つに移る」といったところだろうか．

2.6.2　NFA の動作

基本的には 2.4.2 に書いたオートマトンの動作と同様．ただし，

「NFA が受理する記号列」⇔「入力終了時にとり得る状態のうちに，最終状態に含まれるものが存在する」

という点が異なる．この定義は中々複雑に感じるので，以下で細かく説明する．

「入力終了時にとり得る状態」って??

NFA は，入力を 1 個取るごとに，何通りかの遷移の可能性を持つ．従って，入力終了時の状態は，一意に定まらない．

「最終状態に含まれるものが存在する」って??

[*9] [?] にそう書いてありました．
[*10] [?] にそう書いてありました．

NFA の定義によると，最終状態は複数あって良い．繰り返しになるが，FA でも最終状態は複数あって良い．

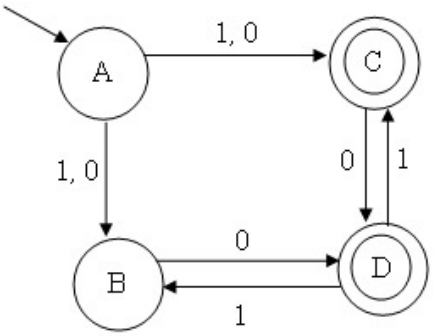


図 2.2　NFA を表す状態遷移図の例

例 2.6.1:　簡単な NFA

図 2.2 は，簡単な NFA の例を示している．[*11]NFA ならではの特徴を以下に指摘する．

- 状態と入力が決まっても，次状態が一意には決まらない場合がある．例えば，状態 D にいたとき，入力 1 を受け取った場合，次状態としては B も C もあり得る．

また，

- 終了状態が複数ある．C, D が終了状態である．

ことにも注意．

この NFA が，入力 001 を受理するかを考えてみる．

- 開始状態は A.
- 1 ビット目の入力 0 をとると，B にも C にも移れる．いま，B に移るとする．
- 2 ビット目の入力 0 をとり，D に移る．
- 3 ビット目の入力 1 をとると，B にも C にも移れる．いま，B に移るとする．
- 入力列を読み終わったが，最終状態の集合 (C, D) の元にいないので，とりあえず失敗．
- 気を取り直し，11 で B でなく C に移る．
- 入力列を読み終わり，最終状態の集合の元 C にいるので，この NFA は入力 001 を受理すると言える．

もちろん，11 で失敗が分かったとき，11 でなく 11 に戻っても良い．いずれにせよ，入力が読み終わった時点でいずれかの最終状態にいられるようなパスを 1 つ見つけてしまえば，NFA は入力を受理すると言える．

例 2.6.1 により，何となく掴みどころのなかった NFA を分かって頂けたかと思う．

ちなみに，NFA の「失敗したら分かれ道に戻る」という動作は，プログラミング手法でいう backtrack に対応する．さらにプログラミング的な話にすると，別に backtrack を使わずとも，分かれ道 (=分岐) が発生した時点で，スレッドなりプロセスなりを生成し，並列に入力を取っていってテストしても良い．

2.7 NFA と FA の等価性

ここでは，NFA と FA の等価性について説明する．もちろん「等価」といっても，NFA と FA は全く同じマシンであるということを主張しているわけではない． NFA と FA の受理言語の集合が一致する という点において等価ということだ．すなわち，

定理 2.7.1
どんな FA についても，同じ受理集合を持つ NFA が作れる．

定理 2.7.2
どんな NFA についても，同じ受理集合を持つ FA が作れる．

の両方が成り立つということだ．これを以下に示す．

証明 2.7.1 定理 2.7.1 の証明
これは，$FA \subset NFA$ なので簡単．

まず，与えられた FA(given FA と表記する) と全く同じ遷移状態を，構成する NFA(obj NFA と表記する) に持たせる．一般には NFA がある状態にいてある入力を受けるとき，遷移状態は 1 つに定まらないが，今回は given FA と全く同じ遷移状態を obj NFA に持たせる．

更に，obj NFA には given FA と同じ最終状態を持たせる．

以上の手順により，obj NFA は given NFA と全く同じものになり，すなわち同じ受理集合を持つことが示された．

証明 2.7.2 定理 2.7.2 の証明
*12

与えられた NFA を $M = (Q, \Sigma, \delta, q_0, F)$ *13とするとき，対応する FA として，$M' = (Q', \Sigma, \delta', q_0, F')$ を以下のように構成すれば良い．

$$Q' = 2^Q \quad (Q \text{ が有限だから } 2^Q \text{も有限に納まる})$$
$$q'_0 = q_0$$
$$F' = \{S \subset Q | F \cap S \neq \phi\}^{*14}$$

$$\delta' : 2^Q \times \Sigma \to 2^Q$$

つまり，

$$\delta'(\{q_1, q_2, \cdots, q_n\}, a) = \cup_{i=1}^{n} \delta(q_i, a)$$

*11 この図では，状態 B において入力 1 をとった場合の動作などが明示されていません．その時は状態遷移しない，すなわち，「同じ状態に遷移する」と考えて良いでしょう．なにせネットから引っ張ってきた図ですので :-)

ここで，$\{q_1, q_2, \cdots, q_n\}$ は 2^Q の部分集合, a は Σ の元，q_i は 2^Q の元.
以下略. [*15]

2.8 FA，NFA の限界

ここでは，FA と NFA に共通する限界について説明する.

どちらも状態 Q は有限であり，それはつまり記憶が有限ということ．従って，有限の記憶で処理できない計算は，FA や NFA には処理できないということである.

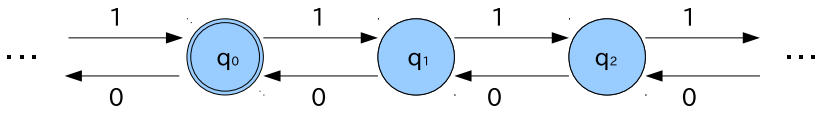


図 2.3　入力列の 0/1 の個数が同数かを判定する FA

例 2.8.1: 入力列の 0/1 の個数が同数かを判定する FA は構成できない

図 2.3 は，入力列の 0/1 の個数が同数かを判定する FA を表している.

開始状態が q_0 で，1 を受け取ると q_1 に，0 を受け取ると q_{-1} に移る．つまり，各状態は,

$$q_{[\text{受け取った 1 の個数}]-[\text{受け取った 0 の個数}]}$$

となる.

終了状態も q_0 であるので，受理言語は，「0,1 の個数が同数である 0/1 の列」となる.

いま，遷移図の一番左の状態を q_l，一番右の状態を q_r としよう．FA において，状態数は有限なので，必ずこのような状態は存在する.

もし，入力列がある程度短く,

$$|\text{入力列}| \leq min\{-l, r\}$$

であるとすると，この入力列を判定することはできるが，入力列が長く,

$$|\text{入力列}| > min\{-l, r\}$$

となるとき，状態が q_l もしくは q_r に達し，更に右か左に動くことができなくなる可能性がある (例: q_{-1}, q_0, q_1 の 3 状態から成る FA に，入力列 11 を与えたとき)．そのような長い入力列に対しては，FA が構成できていないことになる.

そして，FA に与える入力の長さに制限はなく，無限長でも良いので，全ての入力列を判定でき

[*12] 実は証明になってません．完全な証明を知りたければ，[?] の p.29 あたりに載ってます.

[*13] 引数は左から，状態集合，入力記号の集合，遷移関数，開始状態，終了状態の集合，でした.

[*14] ベン図で考えるとわかりやすい．構成する FA の最終状態の中に，与えられた NFA の最終状態であるものが存在すれば良い．これも数式で書くと,

$$\exists x(x \in F \wedge x \in S)$$

ってこと.

[*15] 先生もそんなに厳密性を気にされる方じゃなさそうです:-)

る FA は存在しない．

例えば，入力列が 110110···(無限に続く) では，どれだけ (有限の範囲で) 状態の多い FA でも対処できない．

2.9 文脈自由文法 (Context-free grammar, CFG)

文脈自由文法 (以下， CFG) とは，1995 年に Avram Noam Chomsky が提唱した，自然言語文法を記述するための枠組みであり，それは 生成文法 (Generative grammar) [*16]の形をしている．

定式化は後でするとして，まずは具体例を見てみる．

例 2.9.1: CFG で書かれた単純な生成文法

以下の規則 (生成文法) があるとする．

- ＜文＞ → ＜名詞句＞＜動詞句＞
- ＜名詞句＞ → ＜名詞＞
- ＜名詞句＞ → ＜形容詞＞＜名詞＞
- ＜名詞＞ → birds
- ＜形容詞＞ → big
- ＜動詞＞ → fly

この文法から生成される＜文＞は，{'birds fly', 'big birds fly'} である．

なお，＜○○＞を 非終端記号 或いは 変数 ，＜＞で囲まれていない記号列を 終端記号 という．[*17]

2.9.1 CFG の定式化

CFG は，$G = (V, T, P, S)$ で定式化される．

V

非終端記号 (Non-terminal symbol) の有限集合．非終端記号は， 変数 (variable) とも呼ぶ．非終端記号には，生成規則の右辺にはに来ないという特徴がある． [*18]

T

終端記号 (Terminal symbol) の有限集合．終端記号には，必ず生成規則の右辺に来るという特徴がある．[*19]

[*16] 生成規則 (Generative rule) と呼ぶことも多い．

[*17] 終端，非終端という呼び方は，構文解析をした結果できる構文木の葉 (leaf) にあるか，節 (node) にあるかという，図的なイメージから来ている．

[*18] 非終端記号は大文字で始まる記号で表すことが多い．$A, B, Expr$ など．

[*19] 終端記号は小文字で始まる記号で表すことが多い．a, b, if など．

P

生成規則 (Production rule) の集合.

S

開始記号 (Start symbol) .[*20] $S \in V$ である．上述の例では，S = ＜文＞.

2.9.2 CFG による構文木 (Syntacs tree) の生成

ある記号列が与えられたとき，それが CFG の生成規則に従っていれば，その記号列から 構文木 (Syntacs tree)，或いは 導出木 (Derivation tree) [*21]をつくることができる.

例 2.9.2: 記号列と CFG の生成規則から構文木をつくる

「x の加算と乗算から成る式」を表す CFG の生成規則を考えると，例えば以下のようなものができる.

- $V = \{S, Fact, Term, Expr\}$
- $T = \{x, +, \times, (,)\}$
- $S = Expr$ [*22]
- $Fact \to x | (Expr)$ [*23]
- $Term \to Fact | Term \times Fact$
- $Expr \to Term | Expr + Term$

こういう生成規則を考えるコツは,

- 小さいものから大きいものへ，ボトムアップで作っていく.
- どんな部品 (この例では，終端記号と $Fact, Term, Expr$) があるのかを意識する.

の 2 点かと思う.

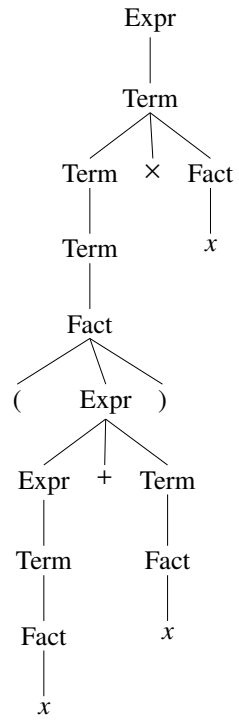
話が逸れたが，上記の生成規則と,

$$(x + x) \times x$$

という記号列から，構文木を生成してみる．これもボトムアップの方向性で，終端記号から上に伸ばしてつくっていくのが楽だろう.

[*20] 構文木の根 (root) に来る.

[*21] 下に続く例を見ると，導出木という名前がしっくりくるかもしれない．構文木という呼び方は，言語処理系 (コンパイラ・インタプリタ) の理論で出てくることが多い．以後，特に理由がなければ構文木という言葉を使う.



これが生成された構文ツリーである．

2.9.3 CFG の標準形

CFG は，その表記の自由さ故に使い勝手がよいが，あまり自由すぎると却って複雑な生成規則を作ることにもつながる．そこで，ある程度の制約を加えた CFG，すなわち CFG の標準形というものが以下のように定められている．

もし L が空でない文脈自由言語 (ある CFG によって生成することのできる言語) であるならば，L は次の条件を満たす CFG G(これが標準形) で生成できる．

- G のどの非終端記号，どの終端記号も，L の中の何かの語の導出に現れる．
- G は，非終端記号 A を非終端記号 B に置き換える規則 $A \to B$ を持たない．
- もし ϵ が L の中になければ，G は $A \to \epsilon$ という規則を持たない．

この条件が意味するところを端的に言うならば，「標準形は無駄な (終端，非終端) 記号を持っていない」ということになる．

標準形については次の定理が成り立つ．

[*22] 略語を解説する．Expr: Expression(式)，Fact: Factor(因子)，Term: Term(項)．

[*23] | 記号は，「または」を表す．つまり，

$$[A \to B, A \to x] \Leftrightarrow A \to B|x$$

定理 2.9.1
任意の CFG G は，標準形に置き換えることができる．

この証明は授業で扱っていない．
例 2.9.2 で使用した生成規則を，標準形に変換する．

例 2.9.3: CFG 生成規則の標準形への変換

例 2.9.2 の規則は，例えば $Expr \to Term$ などが標準形の条件を満たしていない．標準形に直すと，以下のようになる．

- $V = \{S, Fact, Term, Expr\}$
- $T = \{x, +, \times, (,)\}$
- $S = Expr$
- $Fact \to x | (Expr)$
- $Term \to x | (Expr) | Term \times Fact$
- $Expr \to x | (Expr) | Term \times Fact | Expr + Term$

標準形の中でもある程度の自由度があり，標準形に更に制約を加えたのが，以下で述べる Chomsky の標準形と Greibach の標準形である．

2.9.4 Chomsky の標準形

Chomsky の標準形は，実用性はあまりない．だが，CFG の創始者の考案した標準形なので，触れないわけにもいかないのだろう．
Chomsky の標準形においては，全ての生成規則は以下のように表せる．

- $A \to BC$
- $A \to a$

ただし，大文字は非終端記号を，小文字は終端記号を表す．
このように，非終端記号と終端記号が右辺で混ざることがないのが特徴である．
Chomsky の標準形においても，以下の定理が成立する．

定理 2.9.2
任意の CFG G は，Chomsky の標準形に置き換えることができる．

この証明も授業では取り扱わなかったが，[**?**] の p.128 に掲載されてあることを指摘しておく．
ここでも，例 2.9.2 の生成規則を，Chomsky の標準形に変換してみる．

例 2.9.4: CFG 生成規則の Chomsky 標準形への変換

- $V = \{S, Fact, Term, Expr, LParen, RParen, Mult, Plus, Fact', Term', Expr'\}$

- $T = \{x, +, \times, (,)\}$
- $S = Expr$
- $LParen \to ($
- $RParen \to)$
- $Mult \to \times$
- $Plus \to +$
- $Fact' \to LParen\, Expr$
- $Fact \to x \mid Fact'\, RParen$
- $Term' \to Term\, Mult$
- $Term \to x \mid Fact'\, RParen \mid Term'\, Fact$
- $Expr' \to Expr\, Plus$
- $Expr \to x \mid Fact'\, RParen \mid Term'\, Fact \mid Expr'\, Term$

2.9.5 Greibach の標準形

Greibach の標準形 は，非常に実用的である．Greibach の標準形においては，すべての生成規則が

$$A \to a\alpha$$

の形式になる．ここで，a は終端記号で，α が長さ 0 以上の非終端記号の並び．

生成規則の右辺が必ず終端記号で始まるというのが実用性を与えるポイントで，これにより，先読みが容易に行えるようになる．[*24]

やはり，Greibach の標準形についても以下の定理が成立する．

定理 2.9.3
任意の CFG G は，Greibach の標準形に置き換えることができる．

またしても授業では取り扱わなかったが，[**?**] の p.133 に掲載されているので，興味のある方はどうぞ．

では，例 2.9.2 の生成規則を，Greibach の標準形に変換してみる．

例 2.9.5: CFG 生成規則の Chomsky 標準形への変換

- $V = \{S, Fact, Term, Expr, RParen, Mult, Plus\}$
- $T = \{x, +, \times, (,)\}$
- $S = Expr$
- $RParen \to)$
- $Fact \to x \mid (Expr\, RParen$

[*24] コンパイラ実験をやった人はピンと来ると思います．ピンと来なくても，Greibach は実用的と頭の片隅に置いておけば全く問題ないかと思います．

- $Mult \to \times Fact$
- $Term \to x \mid (Expr\ RParen \mid x\ Mult \mid (Expr\ RParen\ Mult$
- $Plus \to + Term$
- $Expr \to x \mid (Expr\ RParen \mid x\ Mult \mid (Expr\ RParen\ Mult$
- $Expr \to x\ Plus \mid (Expr\ RParen\ Plus \mid x\ Mult\ Plus \mid (Expr\ RParen\ Mult\ Plus$

2.9.6 CFG の能力

CFG では，FA や NFA が受理するか判定できないような言語も記述することができる．例 2.8.1 では，0 と 1 の個数が同じであるような記号列の集合 (=言語) は FA では処理しきれないということを述べたが，CFG ではこのような言語を構成することができる．

例 2.9.6: CFG による 0/1 が同数な言語の生成

[*25] E は 0 と 1 を同数含む 2 進列を表すとする．文法の変数はこのように何らかの意味付けをして考えれば分かりやすい．

空列は，同数の 0 と 1 から成る言語 L に属するので，$E \to \epsilon$ という規則が考えられる．E は ϵ でなければ 0 か 1 から始まる．そこで A を変数として，$E \to 0A$ という規則を作ると，A は 0 の個数が 1 の個数より 1 つ少ない 2 進列ということになる．同様に，$E \to 1B$ という規則を作ると，B は 1 の個数が 0 の個数より 1 つ少ない 2 進列ということになる．

同じような規則を A について作ってみる．A は空列ではないので，0 か 1 から始まる．1 から始まる場合は，残りの部分において 0 と 1 が同数現れる．したがって，$A \to 1E$ という規則が作れる．A の意味をもつ 2 進列が 0 から始まると，残りの部分で 0 が 1 より 2 つ少ないということになる．また新しい変数がいるだろうか．0 の個数が 1 の個数より 2 つ少ないということは，うまく分割すれば 0 の個数が 1 の個数より 1 つ少ない列を 2 つつなげたものになっている．これは AA というパターンで表せる．したがって，$A \to 0AA$ という規則も作れる．

B について同様の考察をすると，$B \to 0E$, $B \to 1BB$ という規則が作れる．

以上により，以下のような CFG を構成できる．

- $V = \{E, A, B\}$
- $T = \{\epsilon, 0, 1\}$
- $S = E$
- $E \to \epsilon \mid 0A \mid 1B$
- $A \to 0AA \mid 1E$
- $B \to 0E \mid 1BB$

[*25] この例は，一部表記を除いて [**?**] の p.15,16 の引用である．

2.10　プッシュダウンオートマトン (Push-down automaton, PDA)

前節で，CFG は FA や NFA の扱えない言語も扱えることが分かった．ここでは，CFG の生成する言語を受理することのできるオートマトン， プッシュダウンオートマトン (以下， PDA) について説明する．

2.10.1　PDA の機構の概略

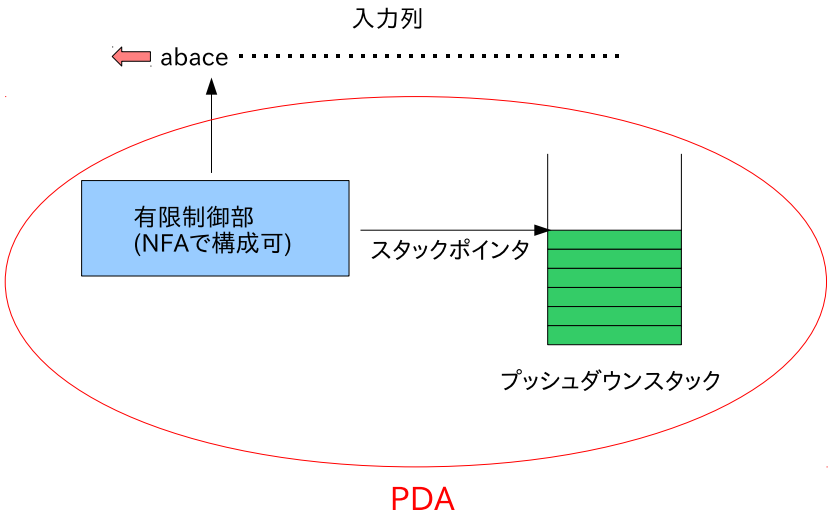


図 2.4　PDA の概略図

図 2.4 に，PDA の概略図を示した．図の説明を簡単にしていく．

有限制御部 (Finite control)

NFA で構成できる． [*26]当然，有限制御部の持つ状態数は有限になるので，この機構だけでは無限に長い入力列を処理できない．

プッシュダウンスタック (Push-down stack)

いわゆるスタックの一種．有限制御部が入力を読み取ると，スタックポインタが指す要素をポップする．また，読み取った入力に応じ，適当な記号列をプッシュしていく．このプッシュダウンスタックが無限に伸ばせる ということが，PDA に CFG と同等の表現力を持たせている．

スタックポインタ (Stack pointer)

プッシュダウンスタックの先頭を常に指している．PDA は，プッシュダウンスタックの要素すべてを

[*26] 後の 2.11 で説明するように，FA で構成することも可能．その場合，特にその PDA を DPDA(Deterministic push-down automaton) という．これと対比して，有限制御部を NFA で構成する PDA を NPDA(Non-deterministic push-down automaton) と呼ぶこともある．

常に把握するのではなく，スタックポインタが指す要素だけを把握していればよい．この制約により，PDA は有限の記述で定式化される．

2.10.2 PDA の定式化

PDA は，以下のように定式化される．

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

Q 状態の有限集合．

Σ 入力アルファベット．

Γ スタックアルファベット．つまり，スタックにプッシュすることのできる記号集合．

δ 遷移関数．数式的には，

$$\delta : Q \times \Sigma \times \Gamma \to 2^{Q \times \Gamma}$$

と表せる．この表記は，NFA との類似性を思わせる．解釈すると，「状態・入力記号・スタックトップによって，次状態と新たなスタックトップが非決定的に定まる」といったところか．

q_0 初期状態．

Z_0 初期記号．スタックの底にある記号 ($\in \Gamma$)

F 最終状態の集合．

2.10.3 PDA の動作

PDA M の遷移関数が具体的に

$$\delta(q, a, z) = \{(p_1, \gamma_1), (p_2, \gamma_2), \cdots, (p_n, \gamma_n)\}$$

と与えられたときの動作を詳細に見てみる．

状態 q にある PDA M は，スタックトップに記号 z がある時に入力記号 a を読むと，状態 p_i $(1 \leq i \leq n)$ に移る (非決定性)．このとき，スタックトップの z はポップし，スタック記号列 γ_i をプッシュする．

2.10.4 PDA の受理言語

PDA M が受理する言語 $L(M)$ は，以下の 2 種類である．

最終状態で受理する言語

$$L_f(M) = \{w| \quad \text{初期状態 } q_o, \text{スタック空から始めて,} \\ \text{入力終了時に最終状態 } F \text{ の 1 つに到達する可能性があるような入力 } w\}$$

空スタックで受理する言語

$$L_e(M) = \{w| \quad \text{初期状態 } q_o, \text{スタック空から始めて,} \\ \text{入力終了時にスタックが空になる可能性があるような入力 } w\}$$

前者は NFA の受理言語と同じようなものなので分かるだろうが，後者は真新しく思えるだろう．しかし，実は次のような定理があり，L_f, L_e は同等であると言える．

定理 2.10.1
$L = L_f(M_1)$ であれば，$L = L_e(M_2)$ となるような PDA M_2 を構成できる．

定理 2.10.2
$L = L_e(M_2)$ であれば，$L = L_f(M_1)$ となるような PDA M_1 を構成できる．

この証明は授業で取り扱わなかったので，ここでも取り扱わない．[*27]

2.10.5 PDA と CFG の等価性

正規表現と FA が，「正規表現の生成する言語と，FA の受理言語は一致する」という点で等価だったのと同様に，CFG と PDA も，「CFG の生成する言語と，PDA の受理言語は一致する」という点において等価である．より具体的には，以下の定理が成り立つ．

定理 2.10.3
どんな CFG に対しても，その生成言語を受理する PDA を構成できる．

定理 2.10.4
どんな PDA に対しても，その受理言語を生成する PDA を構成できる．

証明 2.10.1 定理 2.10.3 の証明
[*28]まず，CFG としては Greibach 標準形のものを仮定する (定理 2.9.3 にあるとおり，どのような CFG でも Greibach 標準形にできる).

CFG $G = (V, T, P, S)$ に対して，ただ 1 つの内部状態を持つ Σ 上の PDA $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ を，次のように定義する．

- M の内部状態はただ 1 つなので，

$$Q = F = \{q_0\}$$

[*27] [**?**] の p.153 辺りに証明があります.

- M が受け取る記号の集合 Σ は，G の用いる終端記号の集合 T と一致させ，

$$\Sigma = T$$

- M のスタックに入れる記号の集合 Γ は，G の用いる非終端記号の集合 V と一致させ，

$$\Gamma = V$$

- M のスタックの初期の底は，G の開始記号 S と一致させ，

$$Z_0 = S$$

更に，M の遷移関数 δ は次のように定める．

G の生成規則 P が

$$\begin{aligned} A \to\ & a_1\ \gamma_{11} \,|\, a_1\gamma_{12} \,|\, \cdots \,|\, a_1\gamma_{1m_1} \\ & a_2\ \gamma_{21} \,|\, a_2\gamma_{22} \,|\, \cdots \,|\, a_2\gamma_{2m_2} \\ & \cdots \end{aligned}$$

と与えられたとき，遷移関数を

$$\begin{aligned} \delta(q_0, a_1, A) &= \{(q_0, \gamma_{11}), (q_0, \gamma_{12}), \cdots, (q_0, \gamma_{1m_1})\} \\ \delta(q_0, a_2, A) &= \{(q_0, \gamma_{21}), (q_0, \gamma_{22}), \cdots, (q_0, \gamma_{2m_2})\} \\ &\cdots \end{aligned}$$

の様に定める．この意味を解釈すると，「PDA M の状態が (1 状態だけなので当然だが)q_0 で，スタックポインタは A を指しているときに，入力 a_i を読み取る．すると，次状態は変わらず q_0 だが，スタックからは A をポップし，その後で γ_{ij} のいずれかをプッシュすることになる．」といったところか．ここで，スタックから A をポップしたのは，入力を読み取るときの PDA の共通動作である．

以上のように構成された PDA M は，元の CFG G の生成言語と同じ受理言語を持つ．[*29]

例 2.10.1: 与えられた CFG から等価な PDA を構成する

次のような Greibach の標準形の CFG G

- $V = \{A, B, C, D, X\}$
- $T = \{a, b, c, d, x\}$
- $S = A$
- $A \to aBC \,|\, aD$
- $B \to b$
- $C \to c$
- $D \to dX$

[*29] 基本授業通りですが，前提条件らへんで [**?**] を参考にしました．

[*29] ここに飛躍があるが，それを埋めたければ [**?**] をどうぞ．特に気にならなければ，下の例で勘弁してください．

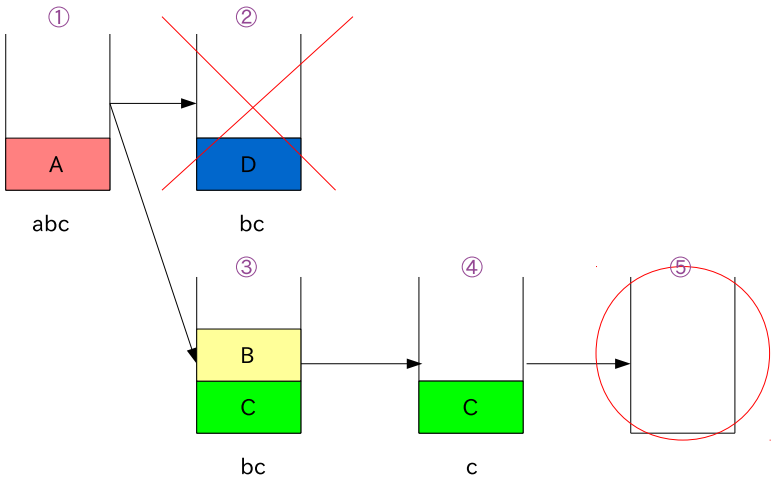


図 2.5　CFG の生成記号列を PDA が受理する様子

- $X \to x$

が与えられたとして，これと等価な PDA を構成する．証明 2.10.1 によると，そのような PDA M は，以下のように構成できる．

- $Q = F = q_0$
- $\Sigma = T = \{a, b, c, d, x\}$
- $\Gamma = V = \{A, B, C, D, X\}$
- $Z_0 = S = A$
- δ :

$$\delta(q_0, a, A) = \{(q_0, BC), (q_0, D)\}$$
$$\delta(q_0, b, B) = \{(q_0, \epsilon)\}$$
$$\delta(q_0, c, C) = \{(q_0, \epsilon)\}$$
$$\delta(q_0, d, D) = \{(q_0, X)\}$$
$$\delta(q_0, x, X) = \{(q_0, \epsilon)\}$$

実際，この PDA M が，CFG G の生成する記号列を正しく受理し，G が生成しない記号列を受理しないことを確かめる．

まず，G の生成する記号列，'abc' を M が受理する過程を，図 2.5 で見ていく．ただし，M は状態を一つしか持たないので，図中の PDA は簡略化してスタックと入力列のみの関係を描いている．

- スタックの底には A が積まれた状態でスタート．入力の先頭 a を読む．このとき，適応可能な遷移関数は，$\delta(q_0, a, A) = \{(q_0, BC), (q_0, D)\}$．また，スタックポインタの指していた A は

ポップされる．

- 遷移関数から，$(q_0, a, A) \to (q_0, D)$ という規則を適応したとする．すると，スタックには D がプッシュされる．しかし，次の入力は b であり，$\delta(q_0, b, D)$ という遷移関数は存在しないため，ここで (この path では) 受理失敗となる．
- 気を取り直して，1 の状態から規則 $(q_0, a, A) \to (q_0, BC)$ を適応したとする．すると，スタックには B, C が この順番で [*30]積まれる．入力の先頭 b を読み，B はポップされる．そして，唯一適応可能な規則 $\delta(q_0, b, B) \to (q_0, \epsilon)$ を適応する．
- 3 で適応した規則によると，ポップされる記号はないので，スタックには C だけが残る．この状態で，規則 $\delta(q_0, c, C) \to (q_0, \epsilon)$ を適応し，c を読んで C はポップされる．
- スタック空で入力記号列を読み終えたので，'abc' は受理された．

次に，G が生成しない記号列，'abx' を M が弾く過程を考える．それは，上記のプロセス 1-3 までは全く同じで，4 で (q_0, x, C) に対して適応可能な関数がなくなり，ここで完全に受理を失敗するという過程になる．

証明 2.10.2 定理 2.10.4 の証明

[*31]CFG の非終端記号の集合を V として，全て $q, p \in Q$ と $A \in \Gamma$ の組 $[q, A, p]$ に対応するものを用意する．

各 $q \in Q$ に対して，$S \to [q_0, Z_0, q]$ という生成規則を作る．

$\delta(q, a, A)$ が $(q_1, B_1 B_2 \cdots B_m)$ を含むとき，任意の $q_1, q_2, \cdots, q_{m+1} \in Q$ に対して，

$$[q_1, A, q_{m+1}] \to a[q_1, B_1, q_2][q_2, b_2, q_3] \cdots$$

という生成規則を設けると OK.

2.10.6 PDA の能力

PDA は CFG と等価であることは，2.10.5 で述べた．この説では，FA/NFA や正規表現では受理・生成できない[*32]が，PDA や CFG では受理・生成可能な「奇数文字の回文[*33]」を例に取り，PDA の能力を示す．

例 2.10.2: 2 文字から成る奇数文字の回文

まずは，アルファベット $\{a, b\}$ から成る奇数文字の回文を生成する CFG を作り，次にそれを PDA に翻訳するという手順を取る．その CFG は，以下のように考えて構成できる．

- 開始記号 S がそのまま回文全体を表すことにする．
- a, b1 文字も回文なので，$S \to a \mid b$

[*30] この資料中は，この順番でスタックを積めという規則は明記していないのだが，そう言うものだと覚えておいてください．
[*31] この証明は理解できてないので，板書の丸写しです．おかしな所があったり，こう解釈するんだよってのがあったら，掲示板にでも書き込んでくれると助かります．
[*32] 受理・生成する言語が無限長を持つと，どうしても無理になる．
[*33] 偶数文字の回文だと，ϵ も回文となってしまい，それを例外的に扱うのが面倒なだけ．

- S は回文なのだから，a で始まる回文は，aSa で全て尽くせる．
- 同様に，$S \to bSb$

これより，以下のように CFG G を定義できる．

$$G = (\{S\}, \{a, b\}, P, S)$$

生成規則 P は，次のとおり．

$$S \to a \mid b \mid aSa \mid bSb$$

例 2.10.1 で見たのと全く同じ手順で G から等価な PDA を構成するために，まず G を Greibach の標準形に変換する．変換後の CFG G' は，以下のようになる．

$$G' = (\{S, A, B\}, \{a, b\}, P, S)$$

ただし，P は下記の生成規則．

$$\begin{aligned} S &\to a \mid b \mid aSA \mid bSB \\ A &\to a \\ B &\to b \end{aligned}$$

後は本当に例 2.10.1 と同じ手順で PDA を構成するだけである．結果の PDA M は，

$$M = (\{q_0\}, \{a, b\}, \{S, A, B\}, \delta, q_0, S, F)$$

ただし，遷移関数 δ は，以下のものである．

$$\begin{aligned} \delta(q_0, a, A) &= \{(q_0, \epsilon)\} \\ \delta(q_0, b, B) &= \{(q_0, \epsilon)\} \\ \delta(q_0, a, S) &= \{(q_0, \epsilon), (q_0, SA)\} \\ \delta(q_0, b, S) &= \{(q_0, \epsilon), (q_0, SB)\} \end{aligned}$$

実際に，この M が，記号列 aba を空スタックで受理する path を示す．(もちろん受理できない path もあるが，それは各自探してみてください．)

- まず，開始状態 q_0，スタックの底 S から始まる．なお，状態は q_0 の 1 通りであるから，今後はスタックについてのみ言及する．
- aba の先頭 a を取り，遷移関数

  $$(q_0, a, S) \to (q_0, SA)$$

  に従い遷移．S がポップされた後に SA がプッシュされる．
- ba の先頭 b を取り，遷移関数

  $$(q_0, a, S) \to (q_0, \epsilon)$$

  に従い遷移．スタック SA から S がポップされるので，A が残る．

- 残りの a を取り，

$$(q_0, a, A) \to (q_0, \epsilon)$$

に従い遷移．A がポップされ，スタックは空になり，aba は受理された．

2.11 決定性プッシュダウンオートマトン (Deterministic push-down automaton, DPDA)

遷移関数の集合要素が高々 1 個であるような PDA は，特に 決定性 PDA(Deterministic push-down automaton, DPDA) と呼ばれる．

例えば，先の例 2.10.2 で，

$$\delta(q_0, a, A) = \{(q_0, \epsilon)\}$$
$$\delta(q_0, b, B) = \{(q_0, \epsilon)\}$$
$$\delta(q_0, a, S) = \{(q_0, \epsilon), (q_0, SA)\}$$
$$\delta(q_0, b, S) = \{(q_0, \epsilon), (q_0, SB)\}$$

という遷移関数があったが，この上 2 つの遷移関数のみを持てばそれは DPDA であり，下 2 つのどちらかを持てばそれは (非決定性)PDA である．

DPDA は全ての CFG を扱うことはできない[*34](例えば回文は無理) が，プログラミング言語の文法などには決定性 PDA で扱える文法を用いることが多い．[*35]

[*34] FA は NFA と等価であったこととは対照的である．

[*35] 後期実験でプログラミング言語を取っている人は覚えがあると思います．もし DPDA で扱えないような文法を田浦先生サイドが提供していたら，構文解析器は backtrack(あるいは並列処理:-D) の機構を持っている必要があったはずです．実際にはそんなことはなく，「'if' トークンで始まる構文は'if-statement' しかない」と断言してコーディングできますね．

第3章

チューリングマシン

まだここの部分書いてない!

3.1 チューリングマシン (Turing machine, TM)

3.1.1 時点表示 (Instaneous description)

時点表示 とは，ある時点での TM の状態の記述のことである．その表記は，以下のようになる．

α_1 : テープ状のヘッドより左の記号列
α_2 : テープ状のヘッドの位置およびそれより右の記号列 (ただし，テープの右端まで続くブランク (以下，B) は除く)
q : 有限制御部の状態

このときの TM の時点表示は，

$$\alpha_1 \ q \ \alpha_2$$

ある時点では TM は全て有限の状態であることに注意 (テープの右端まで続く B は無限長であり得るが，α_2 でそれを無視している).

3.1.2 TM の動作例

例 3.1.3: 同数の記号から成る記号列を受理する TM

$$L = \{a^n b^n \mid n \geq 1\}$$

，すなわち，同数の $aa\cdots, bb\cdots$ から成る記号列を受理 (=最終状態に達する) する TM を考える．

これは，「入力記号 a, b と B の他に X, Y を使う．$a \to X, b \to Y$ に書き換えていく (それにより，もう読んだ文字を判別する).」という方針によって構成できる．まずは，どんな動作にすれば良いかを箇条書きにしてみる．

- ヘッド位置の a を X に書き換え．
- 右に動いていき，最も左側にある b を Y に書き換える．(先にブランク B が見つかれば，a の方が多い)

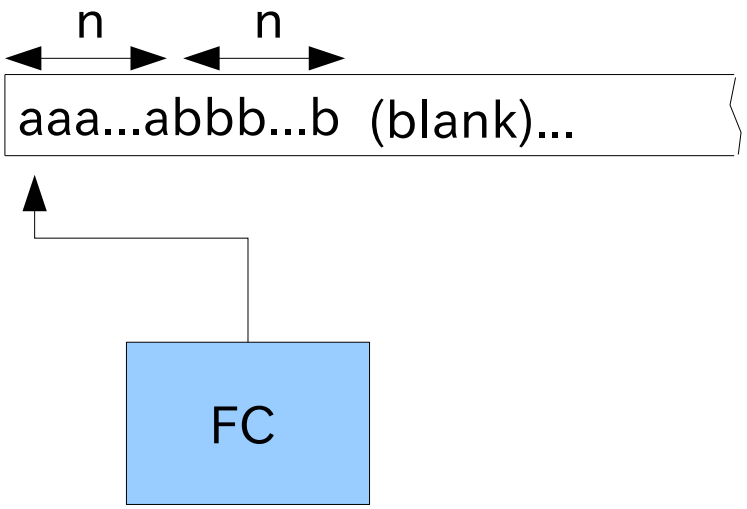


図 3.1　同数の $aa\cdots, bb\cdots$ から成る記号列を受理する TM

- 左に動いていき X を探し，その 1 つ右を見る．
- そこが a であれば，X に書き換えて 3.1.2 に戻って繰り返し．
- そこが a でなければ，右に動いて b を見つける前に B に達したら OK.(b が見つかれば b の方が多い)

この TM を定式化する．それは，次の有限の大きさの動作表を作ることに対応する．結果として，表 3.1 のように，確かに表にできる．ただし，表において，q_0 は TM の開始状態，q_f は TM の終了状態，'-' は何もしないことを表す記号，$(q_i, A, \{L, R\})$ は，「状態 q_i に移り，テープ上の現在ヘッダの指す文字を A に書き換えた後，左か右に動く」ことを表すのに注意．

TM において「受理しない」とは，「最終状態に達しない」ということなので，例えばテープの 1 文字目に b がきた場合はその時点で受理しないことが確定することを表から確かめて欲しい．

さて，この TM が入力 $aabb$ を受理する様子を，時点表示を用いて確かめる．

- $q_0\ aabb$
- $X\ q_1\ abb$
- $Xa\ q_1\ bb$
- $X\ q_2\ aYb$
- $q_2\ XaYb$
- $X\ q_3\ aYb$

表 3.1　同数の記号から成る記号列を受理する TM を表す表 (横:読み取る文字, 縦:状態)

	a	b	X	Y	B
q_0	(q_1, X, R)	-	-	-	
q_1	(q_1, a, R)	(q_2, Y, L)	-	(q_1, Y, R)	
q_2	(q_2, a, L)	-	(q_3, X, R)	(q_2, Y, L)	-
q_3	(q_1, X, R)	-	-	(q_4, Y, R)	-
q_4	-	-	-	(q_4, Y, R)	(q_f, B, R)
q_f	-	-	-	-	-

- $XX\ q_1\ Yb$
- $XXY\ q_1\ b$
- $XX\ q_2\ YY$
- $X\ q_2\ XYY$
- $XX\ q_3\ YY$
- $XXY\ q_4\ Y$
- $XXYY\ q_4$
- $XXYYB\ q_f$

3.1.3　TM の拡張

TM に様々な拡張を施しても，拡張前の TM で同じことができる．より具体的には，同じ受理言語を持つということ．

まず，以下のような比較的単純な拡張を考える．

メモリを積んだ TM

FC に有限ビットの記憶があって，それを制御に使えると考えて良い．[*1]

多重トラックを持つ TM(図 3.2)

この遷移関数は

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}$$

で表される写像となるが，これは，テープ記号を $|\Gamma|^k$ 種にすれば，単一トラックで実現可能なことである．[*2]

[*1] FC の各状態に対して，記憶する m ビットの内容に対応する 2^m 個の状態を作れば良い．それで，q_0 を 0 に，q_5 を 101 にとかすればよい．

[*2] 例えば，多重トラックで「一気に a, b, c を読み取る」ことを，テープ記号を増やした単一トラックで「ζ を読み取ること」に対応させたりすれば良い．

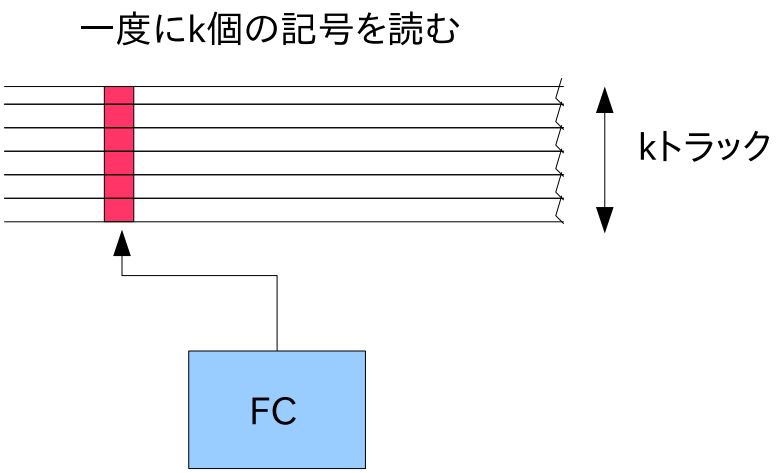


図 3.2　多重トラックを持つ TM

このように TM を拡張すると，TM は次のことができるのが容易に分かる．[*3]

印つけ

2 重トラックを持つ TM を使えば，片方のトラックを印つけに使える．例えば，もう読んだところの 2 トラック目を 1 で埋めるなど．

記号の挿入

- テープ位置の記号を FC に覚える (FC の状態数がテープ記号集合 $|\Gamma|$ よりも十分に大きければ，FC の状態をテープ記号に対応させることができる)．
- 挿入したい記号を書き込んで右へ．
- 1 へ戻って繰り返し．

ただしこのやり方は，挿入部より右の記号を全てずらしているので時間は掛かる． [*4]

記号の削除

- 右に動いてヘッド位置の記号を覚える．
- 左に動いて覚えた記号を書いて右へ．
- 1 に戻って繰り返し．

[*3] このように拡張しないでももちろんできます．例えば，印付けは，テープ状でデータ部分とデータを指す部分を大きく離せばできますね．

[*4] データ構造のベクタ (配列) が挿入に弱いとされるのと同じことですね．

更なる拡張を見てみる．

多テープの TM

図 5．k 本のテープを持つ多テープの TM は，$2k$ 本のトラックを持つ他トラックの TM でシミュレート可能．

多テープの各テープに対応するトラックを，以下のようにペアにする (図 6)

- ひとつはテープの内容
- もうひとつは，多テープのヘッド位置の印づけ

各トラックの印づけした位置を順次探し，そこの記号を FC に記憶していけば良い．

以上，いくつかの拡張を見たが，これらはは所詮テープを増やしたりしただけであり，多項式時間におさまる計算である． ここの記述は弱い．もうちょっと授業進んだら補完します．

3.1.4 非決定性 TM(Non-deterministic turing machine, NTM)

TM の拡張として，NFA に類似した以下の特徴を持たせることができる．

- 動作が複数種類あり得る．
- 最終状態に至る可能性があれば受理．

このような TM を， 非決定性 TM (以下， NTM) と呼ぶ．

NTM は，決定性 TM(DTM)[*5]でシミュレート可能である．

以下で，実際のシミュレートの過程を追ってみる．方針を示すために，図 3.3 を見て欲しい．

この木は TM の探索空間を表していて，各ノードは一意な (状態，ヘッド位置，テープ状態)，すなわち，TM の一状態に対応している．

これらノードのうち，(いくつかある) 最終状態を素直に 深さ優先探索 で探すのが NTM と言える．各ノードから子ノードに移るときに，候補がいくつかあるのが NTM たる所以だ．

一方，これらノードを順番を決めて 幅優先探索 で探すことにより，DTM で NTM をシミュレートできる．以下，具体的なシミュレート法を記述する．

まず，次の 3 本のテープを持つ DTM を用意する．

テープ 1 入力記号列．読み込み専用．

テープ 2 1 から r までの数の有限列．ここで r は，現ノードから見た子ノードの数．例えば，図 3.3 の各深さのノードに左から $1, 2, \cdots$ と番号が付いていたとすると，深さ 3 段目の左から 2 番目のノードにいることを表すには，テープ 2 に 113 を書く．このように，長さ n の列は，深さ n のノードにいることを示す．

テープ 3 作業用．

この DTM は，以下のアルゴリズムに従って動作する．

[*5] 拡張していない TM を NTM と対比させるとき，特にこう呼ぶ．

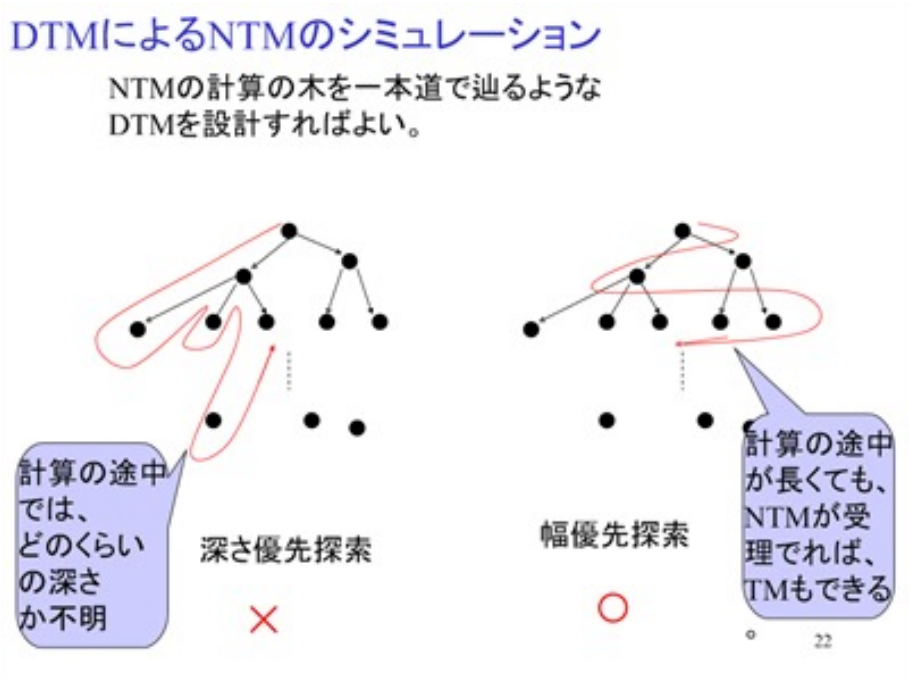


図 3.3　DTM による NTM のシミュレーション

DTM により NTM をシミュレートするアルゴリズム

```
for (n = 0; true; ++n) {  // n はツリーの深さに対応
  長さ n の 1 から r の数の列を初期化 on テープ 2;
  while (全てのノードを表す列をテープ 2 に生成し終わっていない) {
    テープ 1 の内容をテープ 3 にコピー;  // 各ループで，元の入力記号列を破壊しないため
    テープ 2 の内容に沿ってノードを移動;  // そのようなノード移動になるように TM は動作する
                                          // この際，テープ 1 には手を触れず，テープ 3 を読み
書きする
    if (受理ノードに達した) 成功;
    else 次の長さ n の 1 から r の列を生成 on テープ 2;
  }
}
```

このアルゴリズムを見て分かったとおり，終了状態までのステップが n だとすると，NTM/NTM をシミュレートした DTM の計算量は r^n になる．

3.1.5　Random Access Machine

TM は，基本的にヘッド位置の左右一つ分までしか読み書きしないマシンであったが，以下のようにランダムアクセス可能なマシンも TM でシミュレートできる．

- 任意の自然数を保持できる，$0, 1, 2, \cdots$ の番号のついた無限個のメモリ語

- 任意の自然数を保持できる有限本のレジスタ
- 有限種の命令，個々の命令は TM でシミュレート可能
- 命令はメモリに収める．次命令の選択は命令で可能

このシミュレートは，多テープの TM で可能.

構成
テープ 0 メモリを表現

テープ 1 プログラムカウンタ (メモリの番地を指す)

テープ 2 から (2+k) k 本のレジスタ

動作
- テープ 1 の内容と合うものをテープ 0 から探し出す
- その内容を命令として実行 (TM 内でシミュレート可能)
- 3.1.5 に戻り繰り返し

この計算も多項式時間で収まる．このように，TM では現代のコンピュータのシミュレートまでできてしまう.

3.2 チャーチ=チューリングのテーゼ (Church-Turing Thesis)

チャーチ=チューリングのテーゼ (Church-Turing Thesis)，或いは チャーチ=チューリングの仮説 (Church-Turing Hypothesis) とは,

計算可能なものは，TM で処理可能

という主張である．これは，そもそも「計算」という概念をはっきり定義していないので，証明するような事柄ではなく，ただの主張らしい.

3.3 決定性 2 スタックマシン

図 3.4 で表される， 決定性 2 スタックマシン を考える．これは，TM を完全にシミュレートできる 2 本のスタックを持つ DPDA(決定性プッシュダウンオートマトン) であり,

- 1 本のスタックは，TM のヘッドより右の記号を収納
- もう一本のスタックは，TM のヘッドより左の記号を収納

という構成を取る．便宜的に，前者を右，後者を左と呼ぶことにすると，例えば TM のヘッドを右に動かす動作は，右のスタックからポップした記号を左のスタックにプッシュする動作になる.

図 3.4　決定性 2 スタックマシン

次に，決定性 2 スタックマシンをシミュレートすることのできる，計数機械を見る．

3.4　計数機械 (Counter machine)

計数機械 (Counter machine) は，以下のような構成を持つ．

- 読み込み専用の入力テープ
- 有限本の ±1 できるカウンタ
- 有限制御部
- 以下のような遷移関数

$$(\text{カウンタが 0 か否か})^n \times \Sigma \times Q \to (\text{カウンタの増減})^n \times Q \times L, R$$

計数機械には，以下のような能力がある．

加算

$A + B$ を考えたとき，A の数だけあるカウンタを -1 し，その後で B の数だけそのカウンタを $+1$ すればよい．

減算

$A + B$ を考えたとき，A の数だけあるカウンタを -1 し，その後で B の数だけそのカウンタを -1 すればよい．

定数倍の乗算

カウンタ A に j が入っているとき，カウンタ B に $k \times j$ を入れる．
これは，A をループ回数をカウンタ変数とみなし，ループのような形で実装できる．
すなわち，A が 0 になるまで -1 ずつ減らしながら，B を k ずつ増やす．k ずつ増やすのは，有限制御部の記憶を利用すれば可能．

定数での除算

カウンタ A に j が入っているとき，カウンタ B に j/k を入れる．
これは，A が 0 になるまで k ずつ減らしながら，B を 1 つずつ増やすことで実装できる．

定数での剰余

カウンタ A に j が入っているとき，FC の状態を $j \mod k$ に対応させる．
これは，A が 0 になるまで -1 ずつ減らしながら，FC の状態を k 種の中で順に変えていくことでできる．

3.4.1 計数機械による TM のシミュレート

この小節で示すことを始めにまとめておく．ここで，⇒ 記号は，「左辺は右辺でシミュレートされる」ことを示す．

決定性 2 スタックマシンの 1 本のスタック ⇒ 計数機械の 2 本のカウンタ

∴2 本のスタック ⇒ 4 本のカウンタ
4 本のカウンタ ⇒ 2 本のカウンタ

∴2 本のスタック ⇒ 2 本のカウンタ

∴決定性 2 スタックマシン ⇒ 2 本のカウンタを持つ計数機械
TM ⇒ 決定性 2 スタックマシン

∴TM ⇒ 2 本のカウンタを持つ計数機械

まず，計数機械のカウンタ 2 本で，1 本のスタックを以下のようにしてシミュレートする．
対象のスタックが $k-1$ 種のスタック記号

$$Z_1, Z_2, \cdots, Z_{k-1}$$

を持つとする．スタックの内容が

$$Z_{i_1}, Z_{i_2}, \cdots, Z_{i_m}$$

であるとき，これを k 進法の記述と見て自然数にエンコードする．

$$j = i_m + ki_{m-1} + k^2 im - 2 + \cdots + k^m i_0$$

ただし，スタックが空なら $j = 0$
このようにカウンタ j をスタック記号 $Z_1, Z_2, \cdots, Z_{k-1}$ に対応させると，確かにシミュレートできていることを確認する．

- スタックに Z_r をプッシュするには，カウンタを

  $$j' = kj + r$$

  とすればよい．尚，このような計算が計数機械に可能なことは，3.4 で示した．

- スタックからポップするには，

  $$j' = j/k$$

  とすればよい．

- スタックトップによって決定性 2 スタックマシンの動作を変えるには，

$$j \mod k$$

によって計数機械の FC の状態を変えればよい．

従って，4 本のカウンタを持つ計数機械により，決定性 2 スタックマシンをシミュレートできる．
更に，4 カウンタの計数機械は，2 カウンタの計数機械でシミュレート可能である．
4 本のカウンタが i, j, k, l という値を持つとき，これを

$$I = 2^i \times 3^j \times 5^k \times 7^l$$

という値を持つ 1 本のカウンタで一意に表現できる．例えば，4 本のカウンタのうち，i を 1 増やすには，I を 2 倍すればよい．逆に，k を 1 減らすようなときは，I を 5 で割ればよい．また，l がゼロか否かの判定は，I を 7 で割った剰余を見ればよい．

以上をまとめると， 2 カウンタの計数機械で TM はシミュレート可能である と言える．

3.5 TM のテープ記号の制限

テープ記号を 3 種 $\{0, 1, B\}$ としても，一般の TM をシミュレート可能．元のテープ記号を k 種とすると，$\log_2 k$ 個の 0/1 でエンコードすればよい．

3.6 万能チューリングマシン (Universal Turing Machine)

万能チューリングマシン (Universal Tring Machine) とは，どんな TM でもシミュレートできる TM である．[*6]

これは，

[シミュレート対象の TM の記述] [区切り文字\$] [入力記号列] [ブランク $\cdots$]

という入力をとり，自らの状態を

シミュレート対象の < 状態，ヘッド位置の記号，次状態，書き込む記号，左右 >

の組を表す状態として持つ TM である．

シミュレート対象の TM の状態数や入力記号の種類数には上界はないが，有限である．従って，シミュレータ TM も有限の状態数と入力記号でシミュレートできる．

3.6.1 多テープ TM での実現

入力の他に，作業テープを 2 本使う．内訳は，シミュレート対象の TM の動作関数で 1 本，シミュレート対象の TM の状態で 1 本．

入力テープの内容と状態テープの内容を用いて，動作関数テープ (先の入力) を探索し，見つかった動作を実行すればよい．

[*6] エミュレータの理論の基礎になってます．

第4章

帰納的関数論(Recursive function theory)

帰納的関数は,

$$\mathbb{N}^n \to \mathbb{N}$$

の写像であり，この章で述べていくような特徴がある．これは，次章の「計算可能性」を議論するための道具として用いられる．

4.1 Peano の公理

Peano の公理は，自然数を定義する公理である．

- 0 は自然数
- 自然数の次は自然数
- 以上で定義されるものだけが自然数

このように自然数が定義される．現代的には，自然数とは Peano の公理を満たすものとされるらしい．

4.2 原始帰納的関数 (Primitive recursive function)

原始帰納的関数 は，以下のいずれかで定義される．

ゼロ関数 (zero function)

$$zero(x) = 0$$

Peano 公理から 0 の存在は言えるので，このような関数は存在する．

後者関数 (successor function)

$$succ(x) = x + 1$$

Peano の公理の 1 つ，「自然数の次は自然数」を反映した関数.

射影関数 (projection function)

$$proj_k(x_1, \cdots, x_k, \cdots, x_n) = x_k \quad (k \in \{1, \cdots, n\})$$

これは Peano の公理とは関係なく，「引数の中から 1 つを選ぶ関数」を定義しただけ.

関数合成

$$h(x_1, \cdots, x_n) = g(f_1(x_1, \cdots, x_n), \cdots, f_n(x_1, \cdots, x_n))$$

ただし，g, f_k は原始帰納的関数.

原始帰納法 (primitive recursion)

$$h(x_1 \cdots, x_n, 0) = f(x_1, \cdots, x_n)$$
$$h(x_1, \cdots, x_n, y+1) = g(x_1, \cdots, x_n, y, h(x_1, \cdots, x_n, y))^{*1}$$

ただし，f, g は原始帰納的関数．h の再帰呼出しは，最後の引数が必ず小さくなるため，最後の引数が必ず 0 になり，それは 1 行目の定義から計算できる．計算は有限ステップ.

4.3 原始帰納的関数の例

定数関数

任意の定数関数は，有限回の $succ$ と 0 の合成で可能.

例 4.3.4: '2' の定義

$$two(x) = succ(one(x))$$
$$one(x) = succ(zero(x))$$

以降，定数が突然出てきても，それは定数関数で定義されたものと思いましょう.

引数の順序の変更

例 4.3.5: 2 引数の順序の変更

$$f(x, y) = g(proj_2(x, y)), proj_1(x, y))$$
$$= g(y, x)$$

*1 ここで $y+1$ のような加算が出てくるが，加算は 4.3 で定義される.

加算

$$add(x,0) = proj_1(x) = x$$
$$add(x,y+1) = succ(add(x,y)) = \cdots = x + (y+1) = x + y) + 1$$

乗算

$$mult(x,0) = 0$$
$$mult(x,y+1) = add(x, mult(x,y)) = \cdots = x \times (y+1) = x \times y + x$$

前者関数 (predcessor)

$$\begin{aligned} pred(0) &= 0 \\ pred(x+1) &= proj_1(x, pred(x)) \quad \text{(原始帰納法と射影関数を用いた)} \\ &= x \end{aligned}$$

減算

ここでは，自然数の枠組みに収まるように減算を定義する．

$$sub(x,0) = x$$
$$sub(x,y+1) = pred(sub(x,y)) = \cdots = x - (y+1) = (x-y) - 1$$

通常用いる関数のほとんどは原始帰納的である．

4.4 原始帰納的でない関数

原始帰納的でない関数の例として，Ackermann 関数が有名．

$$Ack(0,y) = y + 1$$
$$Ack(x+1,0) = Ack(x,1)$$
$$Ack(x+1,y+1) = Ack(x, Ack(x+1,y))$$[*2]

定義の適用のたびに x か y の少なくとも一方は小さくなるので，この計算は有限ステップで終わる．

しかし，これと同じことをする関数は，原始帰納関数の枠組みで作ることができない．

表 4.1 から見て取れるように，Ackermann 関数の値は，第 1 引数の増加に伴って急速に大きくなる．実は，原始帰納関数には以下のような定理が成り立つ．[*3]

定理 4.4.1

どんな原始的帰納関数 g についても，ある定数 c に対して，

$$g(x_1, \cdots, x_n) < Ack(c, \max(x_1, \cdots, x_n))$$

が成り立つ．

[*2] このような関数は 多重帰納法 という．これは原始帰納関数の枠組みにはない方法．

[*3] これの証明は扱ってません．

表 4.1　Ackermann 関数を具体的に見た表

$x \diagdown y$	0	1	2	3	4	5	6
0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	5	7	9	11	13	15
3	5	13	29	61	125	253	509
4	13	65533	$\cdots$	$\cdots$	$\cdots$	$\cdots$	

$\therefore$ Ackermann 関数は原始帰納的に定義できない.

4.5　原始帰納的述語

真偽値を返す原始的関数を，原始帰納的述語 という.

4.6　帰納的関数 (Recursive function)

原始帰納的関数の定義形式に加えて，最小解関数 を許す．最小解関数とは，原始帰納的述語 $p(n, x_1, \cdots, x_k)$ が真となる最小の n を値とする関数.

帰納的関数は，TM と同等の計算能力を持つ.

第5章

計算可能性 (Computability)

計算可能性とは，「特定の問題に解を与える手続き (プログラム) は作れるか?」という問に応える理論.

問題が完全に固定ならば，計算可能性は自明である.[*1]

「将棋は先手必勝か」というような問に対してはどうか．これも問題が固定であり，計算可能であると言える．この問に対する答えは，[Yes/No] の 2 通りしかない．従って，Yes を出力するプログラムと，No を出力するプログラムを作成すれば，そのどちらかは「特定の問題に解を与える手続き (プログラム)」である.

更に，入力の種類が有限ならば，これも計算可能である．何故ならば，出力は入力によって決まるので，入出力の対応は可算無限種 (enumerable)．従って，それらの入出力 1 ペアずつに対応するプログラムを上記のように作成することができ，結局「問題が完全に固定」ということができる.

ところが，入力が無限種の場合はそうはいかない，正解か否の判定は，有限の手続きで終わらない.

従って， 無限種の入力を持つ問題だけに計算可能性を考える意味がある.

5.1 計算可能でない問題

まず計算可能性を考える前に，本当に解けない問題があることを証明しよう.

定理 5.1.1

ある問題に対して，それを解くことができるプログラムが必ずしも存在しない.

証明 5.1.1 定理 5.1.1 の証明

このことは，表 5.1 によって証明できる．(対角線論法) [*2]

解けない問題があることはわかった．ここで興味があるのは，「記述できる問題仕様で計算できないものがあるか」，もっと言えば，「意味のある仕様の問題で計算できないものがあるか」ということである.

以下では，この問に対する答えを導くべく，いくつかの理論を学んでいく.

5.2 帰納的可算集合 (Recursively enumerable set)

帰納的可算集合 (Recursively enumerable set) は，次のよう定義できる.

[*1] printf(答え); で OK

[*2] この証明は追えませんでした... 分かる方がいましたらご連絡でもおねがいします.

表 5.1　ある問題に対して，それを解くことができるプログラムが必ずしも存在しないことを証明する表

プログラム＼入力	0	1	2	3	4	5	⋯
0	3	7	9	4	1	8	⋯
1	5	0	4	9	2	3	⋯
2	2	1	3	5	6	4	⋯
3	5	1	9	6	1	3	⋯
⋯							

帰納的可算集合 ⇔ 帰納的関数の値域になっている集合[*3]

帰納的可算集合は，要素を順に数え上げる (enumerate) ことができる．[*4]

帰納的可算集合を 計算可能 であるという．何故ならば，帰納的可算集合の要素であれば，順に数え上げていくことで，有限の手間で確かに要素であることがわかるから．[*5][*6]

5.3　ゲーデル関数 (Goedel function)

関数 $G : \mathbb{N}^m \to \mathbb{N}$ が 1 対 1 関数の時，すなわち，

$$G(x_1, \cdots, x_m) = G(y_1, \cdots, y_m) \Leftrightarrow x_1 = y_1,\ x_2 = y_2,\ \cdots,\ x_m = y_m$$

であるとき，G は $x_1, x_2, \cdots, x_m$ をエンコードする関数とみなせる．

更にこのような G に対して，

$$G_i(G(x_1, \cdots, x_m)) = x_i$$

なる関数 $G_1, \cdots, G_m$ があれば，これらでデコードすることができる．

G および $G_1, \cdots, G_m$ が原始帰納的関数であるとき，G を ゲーデル関数 ，$G_1, \cdots, G_m$ を ゲーデル逆関数 と呼び，$G(x_1, \cdots, x_m)$ を $x_1, \cdots, x_m$ の ゲーデル数 と呼ぶ．

例 5.3.6:　ゲーデル関数の例

$$G(x_1, \cdots, x_m) = \Pi_{k=1}^{m} p_k^{x_k}$$

ただし p_k は k 番目の素数．この場合，G_i は素因数分解によって求めることができる．

ゲーデル関数を使って自然数の並びをエンコードして扱える．更に，並びの長さを任意長に拡張することも可能．従って，任意の記号列を自然数にエンコードして扱うことができる．

[*3] 帰納的関数と TM は同じ能力を持っているので，TM で受理できる言語は帰納的可算である．
[*4] 帰納関数が順序付けられる性質のものだったので，その出力も入力と同じ順番に並べられる．
[*5] 要素でないことが有限の手間でわかるとは言っていない．
[*6] 逆に，帰納的でない可算集合では，ある値がその集合の中にあるのか照合する方法を与えられない．

5.4 プログラムの停止性判定の不可能性

プログラムをエンコードしたものと，それに対する入力データを入力して，そのプログラムがその入力データを取り込んだとき，実行を終えることができるか否か．これを判断するプログラムが，停止性判定をするプログラムである．[*7]

これがあると超便利であるが，残念ながらそれは存在しない．このことを，いくつかの手法で証明する．

5.4.1 プログラムを用いた証明

証明 5.4.1 プログラムの停止性を判断するプログラムが存在しないことの証明

背理法による．

まず，停止性判断をするプログラム $P(p,d)$ の存在を仮定する．ただし，p は対象プログラムをエンコードしたもの，d は対象プログラムに与える入力データ．

P からは，サブルーチンとして $stops(p,d)$ を切り分けることができるはず (真偽値を帰り値として持つ).

このもし本当に P が存在すれば，そのサブルーチン $stops(p,d)$ を使って，

$$if\ (stops(p,d))\ then\ 無限ループ\ else\ 停止$$

というプログラムを作ることができる．このプログラムをエンコードしたものを p_0 とする．p_0 は p,d を引数に取るので，$p_0(p,d)$ と表記できる．このプログラムを $p_0(p_0,p_0)$ と呼び出すことを考えよう ($d = p_0$ とは，p_0 をエンコードしたものを入力データに使うということ)．これは，実際

$$if\ (stops(p_0,p_0))\ then\ 無限ループ\ else\ 停止$$

というプログラム p_1 を実行することに対応する ($p_1 = p_0(p_0,p_0)$).

いま，p_1 の if-statement の test 文，$stops(p_0,p_0)$ の返り値について場合分けする．

真を返す

$stops(p_0,d_0)$ が真を返すとは，「プログラム p_0 にデータ p_0 を与えると，p_0 の実行は止まる」ということ．

ところで，$p_1 = p_0(p_0,p_0)$ を呼び出すことを先程から考えているが，この呼出しも，「プログラム p_0 にデータ p_0 を与える」ことを考えている．

しかし，この呼出しによってできたプログラム p_1 をよく見ると，$stops(p_0,d_0)$ が真を返すときは，無限ループに陥るような仕様になっている．従って，矛盾が生じている．

偽を返す

同様に，矛盾.

どちらの場合でも矛盾するので，背理法の仮定が誤りであったことが言える．

[*7] TM に対する万能 TM みたいなものですね.

5.4.2 TM による証明

証明 5.4.2 プログラムの停止性を判断するプログラムが存在しないことの証明 2

次のような機能を持つ万能 TM $X1$ の存在を仮定する．プログラムテープにより表されている TM $X2$ が，入力テープで表されている入力に対して停止するか否かを判定する．

このような万能 TM X_1 が存在できるならば，「X_1 のプログラムと入力を読み込み，そのプログラムが表す TM X_2 がその入力に対して，停止するなら停止しない/停止しないなら停止する」という万能 TM X を作れる．

何故ならば，X_1 は X_2 の停止性を判定できるはずなので，X により X_1 の動作をシミュレートすれば，X は X_2 の停止性を知ることができる．それで，もしも X_2 が停止すると知れば X は無限ループをし，X_2 が停止しないとすれば，X はその時点で停止すればよい．

さて，X を表す万能 TM 用のプログラムと X 自身を表す万能 TM 用のデータを X に入力として与えるとどうなるか?

停止しない
X は X を入力として停止するという判定がなされ，矛盾である．

停止する
同様に矛盾.

よって，背理法の仮定は偽.

5.4.3 対角線論法による証明

[*8]

証明 5.4.3 プログラムの停止性を判断するプログラムが存在しないことの証明 3

[*9]

述語の集合 P を，

$$P = \{ \text{自然数に対する 1 引数の帰納的述語} \}$$

とする．P は可算無限集合で数え上げられる．

P の要素 P_k に対して，P_j をエンコードした c_j を入力として与えた時の結果を表にまとめると，例えば表 5.2 を得る．

いま，停止性を判定する述語 $stops(p, c)$ の存在を仮定する．このとき述語 $diag(x)$ を

$$diag(c) = \text{[*10]}$$

とする．$diag(c_j)$ が返す値は，

$p_j(c_j)$ が停止しないとき
表中の (j, j) 要素は $\perp$ だから，$stops(c_j, c_j)$ は偽なので，$diag(c_j)$ は False.

[*8] この証明は無茶苦茶です．寝てました．ごめんなさい．ここ以外はばっちり目覚めていたのでご安心を.

表 5.2 (⊥ は停止しないことを表す)

述語＼入力	c_0	c_1	c_2	c_3	$\cdots$
P_0	T	F	⊥	F	$\cdots$
P_1	F	T	T	⊥	$\cdots$
P_2	T	F	⊥	T	$\cdots$
$\cdots$					

$p_j(c_j)$ が停止するとき
$stops(c_j, c_j)$ なので，$diag(c_j)$ は $\neg p_j(c_j)$

diag はこの表にあるか? いずれの場合も $diag(c_j)$ は $p_j(c_j)$ とは異なる値になっている．
従って *diag* はこの表にはない．つまり，*stops* の存在を仮定したのが誤りである．

5.5 様々な停止性判定

いま，停止性判定をするプログラムは作れないと証明したばかりだが，実は「一般のプログラムの停止性を判定するプログラム」は作れないとしか言っていない．

特定の制約条件を与えると，プログラムの停止性判定をするプログラムが作れる．ここでは，そのうちの 1 つの例について指摘する．

例 5.5.7: 「プログラムが n ステップ以内に停止するか」を判定するプログラム

停止することを意図して作ったプログラムは，実際に実行することでわかる．ここでは，エミュレータ[*11]を作って n ステップ実行すればよい．このエミュレータは，

$$stops\_in\_steps(p, d, n)$$

のような形式を持つ．ただし，p は判定対象のプログラム，d は p に与える入力，n はステップ数で，*stops_in_steps* は真偽値を返り値に持つ．

この例で出てきた $stops\_in\_steps(p, d, n)$ 関数は，次の節でも出てくるので注意．

5.6 「停止性が計算不能である」理論の利用

証明 5.4.1 で示した,「プログラムの停止性判定をするプログラムは作れない」という定理を利用して，他の問題の計算可能性を論じることがよくある．ここでは，その一つの例を見てみる．

[*10] 一番大事なところ書もらしてました．補完希望.
[*11] 万能 TM を想像しても良いですが，ここでは

停止判定対象のプログラム = 私たちが組むようなプログラム，　エミュレータ = プログラムを走らせる *OS*

とでも考えると，現実に則した理解ができると思います.

証明 5.6.1 「プログラムがいつでもゼロを返す」の判定は計算不可能

「プログラムがいつでもゼロを返す」かどうかを判定するようなプログラムは作成できない．すなわち，

$$always\_zero(p)$$

のような関数は作成できないということを背理法で示す．

stops_in_steps 関数の存在に関しては，証明 5.4.1 の脚注で触れた．いま，プログラム p とデータ d に対して

$$stops\_in\_steps\_pd(n) = stops\_in\_steps(p, d, n)$$

を構成することができる (*stops_in_steps* 関数に対する部分評価, partial evaluation)[*12]．

もし $always\_zero(p)$ 関数が存在可能 (計算可能) だと仮定するなら，$stops(p, d)$ 関数を

$$stops(p, d) = !always\_zero(stops\_in\_steps\_pd(n))$$

のように構成することができる．しかし，$stops(p, d)$ の計算不可能性は，証明 5.4.1 にて示した．従って矛盾であり，背理法の仮定が誤りであったと言える．

[*12] プログラムの世界ではカリー化ともいいます．

第6章

λ計算 (Lambda calculus)

λ 計算 (lambda calculus) [*1] とは，帰納的関数を数と無関係に定義する枠組みである．

6.1 λ抽象化 (Lambda abstraction)

計算の手続きに名前をつけることができる．例えば，

$$a \to 1 + 1$$

など．ここで更に

$$b \to 1 + 1$$

とすると，これらの手続きは名前は違えど，表す手続きの内容は同じである．このような手続きを一緒くたにまとめることを， λ 抽象化 (Lambda abstraction) という．[*2][*3]

6.2 λ式 (Lambda expression)

λ 式 (lambda expression) は，

$$\lambda x.\text{定義}$$

と表記する．[*4] $\lambda x.$定義 は，

「ひとつの引数を取って定義の内容を返す関数」を表す式

例 6.2.8: λ 式の例

$$f(x) = 3x + 2 \quad g(x) = 3x + 2 \quad \cdots$$

[*1] λ 計算を実装したプログラミング言語としては Lisp がとても有名です．この授業の勉強で使えとは当然言いませんが，[?] を読んでみると (プログラム的に) 見える世界が変わってきます．個人的イチオシ．

[*2] λ 抽象化については授業であまり説明がなく，この辺は自分の理解で書きました．もしかしたら間違ってるかもしれないので注意．

[*3] λ 抽象化を利用すれば，いちいち名前をつけずに手続きを定義できます．これはプログラムでは無名関数という形で出てきて，Lisp，Javascript，その他最近の言語では無名関数が使えます．無名関数の概念は，次のような感じ (*do_anything_to_x* を呼び出す際の第 1 引数が無名関数になっています．)

$$do\_anything\_to\_x(\{x * x\}, x); \quad \cdots\cdots\cdots\cdots\cdots\cdots \textcircled{6.1}$$

[*4] $\lambda x.$定義 の x は変数名

などの，「1 変数をとり，その変数を 3 倍した数に 2 を足した数を返す 関数 」は，全て

$$\lambda x.(3x + 2)$$

という 式 で表せる．

6.3 高階関数 (Higher-order function)

高階関数 (Higher-order function) とか，関数を引数や結果とする関数である．[*5]

例 6.3.9: ある関数を 2 回適用する関数を高階関数として記述する

$$g(x) = f(f(x))$$

この g と同じ意味を持つ，

$$twice(f)$$

という関数を作りたい．これを λ 式として書くと，

$$twice = \lambda f.(\lambda x.fx)$$

となる．[*6]

6.4 カリー化 (Currying)

カリー化 (Currying) とは，複数引数の関数を単一引数で表現することをいう．

例 6.4.10: 簡単なカリー化

$$f(x, y, z) = x + y \times z$$

という 3 変数関数を，

$$f_{x,y}(z) = x + y \times z$$

という風に 1 引数関数にすることもできる (x, y を固定した)．これがカリー化．

この $f_{x,y}$ は，λ 式では

$$\lambda x.(\lambda y.(\lambda z.(x + y \times z)))$$

と表記できる．

[*5] 脚注に書いた *do_anything_to_x* (式 6.1) も一種の高階関数です．
[*6] λ 式の表記法は勉強しておきましょう． λ 式で解答するような問題が試験にも出題される模様です．

6.5 λ式について覚えておきたいこと

6.5.1 λ式の略記法

λ式の基本形はλ変数名.定義 であるが，「概念的に複数引数を取るλ式」は，次のように表記できる．

$$\lambda x_1 x_2 \cdots x_n.\text{定義} \equiv \lambda x_1.(\lambda x_2(\cdots.(\lambda x_n.\text{定義})))^{*7}$$

例 6.5.11: 略記法の簡単な例

$$\lambda xyz.(x + y \times z) \equiv \lambda x.(\lambda y.(\lambda z.(x + y \times z)))$$

6.5.2 関数の適用順

関数を連続して適用するとき，その適用順は次のとおりである．

$$M_1 M_2 M_3 \cdots M_n \equiv (((\cdots(M_1 M_2)M_3)\cdots)M_n)$$

ただし，この適用順を明示的に変更したいときは，括弧を使う．

例 6.5.12: 関数の適用順の例

$$fffx \equiv (((ff)f)x)^{*8}$$

高校で習った「合成関数」を再現したいときは，

$$f(f(fx))$$

という風に括弧でくくります．

6.5.3 基本的な関数

$$I \equiv \lambda x.x \quad ;Identity$$

$$K \equiv \lambda xy.x \quad ;\text{定数値を返す関数}$$

6.5.4 引数への適用

[*9]λ式の引数に値を適用するときは，

$$(\lambda\text{引数.定義})\ \text{適用する値}$$

[*7] カリー化の逆適用ですね．
[*8] 「関数に対して関数を入力しているので変に思うかもしれませんが，それが高階関数というものです．」
[*9] 勝手に付け加えました．

と書く.

例 6.5.13: 引数への値適用の例

$$(\lambda x.x + 3)5 = 5 + 3$$
$$= 8$$

6.6 β 変換 (β reduction)

式 M の中の全ての自由な [*10]x を式 N で置き換えた結果を,

$$M[x := N]$$

と書く.

例 6.6.14: 略記法の例

$$\lambda x.(x + (\lambda x.x)5)$$

について,

$$M = x + (\lambda x.x)5$$

とし,

$$\lambda x.(x + (\lambda x.x)5) = \lambda x.M$$

とすると,

$$M[x := 3] = 3 + (\lambda x.x)5 = 3 + 5 = 8$$

この略記法を用いて, 以下のように β 変換 (β reduction) が定義される. [*11]

引数渡し

$$(\lambda x.M)N \underset{\beta}{\longrightarrow} M[x := N]$$

$M \underset{\beta}{\longrightarrow} N$ ならば,

定義の中での変換

$$\lambda x.M \underset{\beta}{\longrightarrow} \lambda x.N$$

[*10] λ 式の引数になっている引数は, λ 式によって束縛されています. つまり, 定義の部分の変数を自由な変数と言います. ところで, 引数は λ 式に束縛されることを利用して, λ 式によってローカル変数が作れます. Lisp の let ですね. 胸熱.

[*11] α 変換もある. それは,

$$\lambda x.x \underset{\alpha}{\longrightarrow} \lambda y.y$$

のように定義される.

引数の変換

$$PM \underset{\beta}{\longrightarrow} PN$$

関数の変換

$$MP \underset{\beta}{\longrightarrow} NP$$

- β 変換の対象となる式を β 基 (β-redex) という．
- 式 P から有限回の β 変換で，

$$P = P_1 \underset{\beta}{\longrightarrow} P_2 \underset{\beta}{\longrightarrow} \cdots \underset{\beta}{\longrightarrow} P_n = Q$$

となるとき，この列 $P_1 \cdots P_n$ を β 変換列といい，

$$P \underset{\beta}{\longrightarrow} Q$$

と表す．

6.7 Church-Rosser の定理

Church-Rosser の定理 とは，次のものである．

定理 6.7.1
同じ式から β 変換列で導かれた二つの式からは，必ず β 変換列によって同じ式を導くことができる．

なお，この性質を合流性 (Confluence) と呼ぶこともある．

図 6.1 は，合流性を表すものである．図中の矢印は全て β 変換を表す．例えば，2 つの黄色い式から，赤い式に「合流」していることが見て取れる．

6.8 正規形 (Normal form)

β 基を含まない λ 式，すなわち，それ以上 β 変換できない式を 正規形 (normal form) であるという．

例 6.8.15: λ 式の正規形への変換

$$(\lambda x.x)(\lambda y.y) \underset{\beta}{\longrightarrow} \lambda y.y$$

この右辺は正規形である．

正規形にならない λ 式もある．

例 6.8.16: 正規形にならない λ 式

$X = \lambda x.(xx)x$ のとき，

$$XX \underset{\beta}{\longrightarrow} (XX)X \underset{\beta}{\longrightarrow} ((XX)X)X \underset{\beta}{\longrightarrow} \cdots$$

どんどん式が長くなっていく方に変換されていく．

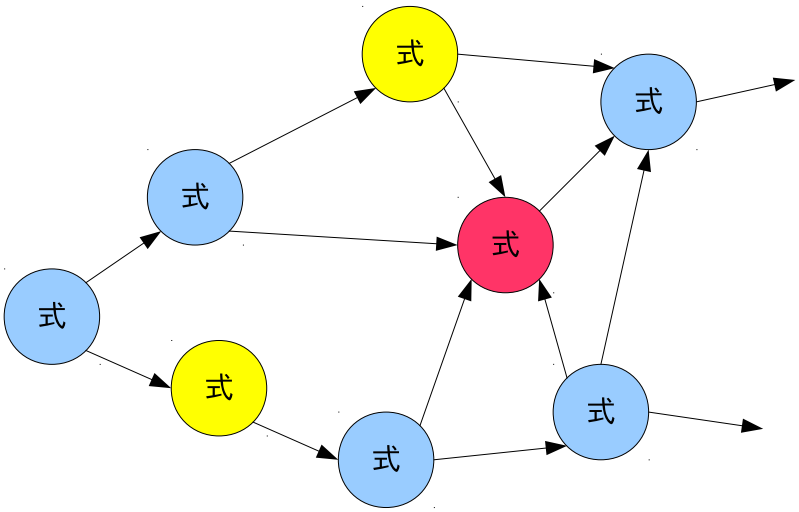


図 6.1　β 変換列の合流性

6.9　λ 式による論理の表現

以下，$[\alpha]$ で概念 α を表す λ 式を意味するものとする．

$$[true] \equiv \lambda xy.x \equiv T$$
$$[false] \equiv \lambda xy.y \equiv F$$

このように定義すれば，式 P の真偽によって，

$$PMN \underset{\beta}{\longrightarrow} \begin{cases} M : P \text{ が真} \\ N : P \text{ が偽} \end{cases}$$

と，if-then-else を表せる．更に，

$$[not] \equiv \lambda pxy.pyx^{*12}$$
$$[and] \equiv \lambda pq.pqp^{*13}$$
$$[or] \equiv \lambda pq.ppq^{*14}$$

*12 p, q は真偽値．引数の，「もし p が真なら x を，偽なら y を返すような式」というのを逆さまにして，「もし p が真なら y を，偽なら x を返すような式」としている．
*13 もし p が真なら q の値．さもなくば p の値，すなわち偽．
*14 もし p が真なら p(真)，さもなくば q(q の真偽に委ねられる)．

6.10 λ 式による自然数の表現

[*15]λ 式による自然数の表現 [*16] は色々知られているが，ここでは Church numeral [*17]と呼ばれるものを扱う．その表記法では，自然数 n を，次のような二引数の高階関数として表現する．

第一引数 何らかの関数 f

第二引数 その関数に渡す値 x

結果 f を x に n 回適用したもの

すなわち，関数の適用回数で自然数を表していると言える．実際に，

$$
\begin{aligned}
[0] &\equiv \lambda fx.x \quad [*18] \\
[1] &\equiv \lambda fx.fx \\
[2] &\equiv \lambda fx.f(fx) \\
&\cdots \\
[n] &\equiv \lambda fx.f(f(\cdots(fx)\cdots)) \quad \text{ただし } f \text{ は } n \text{ 回適用.}
\end{aligned}
$$

と表現できる．この表現を取ると，自然数 n に対して次のような式が定義できる．

$$
\begin{aligned}
[iszero] &\equiv \lambda n.n(\lambda x.F)T \quad [*19] \\
[succ] &\equiv \lambda nfx.f(nfx) \quad [*20] \\
[plus] &\equiv \lambda mnfx.mf(nfx) \quad [*21]
\end{aligned}
$$

6.11 λ 式によるデータ構造

λ 式を使えば，あらゆるデータ構造が表現できる．まず，λ 式による pair の表現を見てみる．pair さえあれば，まず list が「最初の要素と残り」という形で再帰的に定義でき，list があればあらゆるデータ構造が表現可能である．[*22]

[*15] 数を式で表すのですから，偉大な抽象化です．ロマンを感じましょう．

[*16] 現代的には，自然数は集合によって定義するのが普通．

$$
\begin{aligned}
[0] &\equiv \{\} \\
[succ(x)] &\equiv \{x\} \cup x
\end{aligned}
$$

0 を含み $succ$ について閉じている集合は存在 (公理)．自然数は，「そのような集合の共通部分」と定義．

[*17] 「数」は number だが，「数を表す表記法」は numeral という．また，「数字」は digit.

[*18] [0] の定義は，[false] の定義を満たすことに注意．すなわち，[0] は F である．これは，すぐ下の [iszero] の定義で必要になる考えである．

[*19] n が 0，すなわち n が F を表すなら，適用は 0 回だから T を返す．0 でなければ，1 回以上適用するので F を返す

[*20] f をもう一度適用すれば，n の次の数

[*21] $m+n$ を表すには，x に f を $m+n$ 回適用すればよい．ここでは，f を n 回 x に適用した結果に，更に f を m 回適用している．

[*22] 「list があればあらゆるデータ構造が表現可能」っていうのはひょっとしたら嘘かもしれません (でも確かできたはず)．ご指摘大歓迎です．

$$[pair] \equiv \lambda fsb.bfs^{*23}$$
$$[first] \equiv \lambda p.pT$$
$$[second] \equiv \lambda p.pF$$

少々分かりにくいが，次の例でこれが実際に pair を表せていることを確かめる．

例 6.11.1: データ構造の λ 式による表現の例

$$[pair(A,B)] = (\lambda fsb.bfs)\,[A]\,[B]$$

$$\begin{aligned}[first(pair(A,B))] &= (\lambda p.pT)\{(\lambda fsb.bfs)\,[A]\,[B]\} \\ &= (\lambda fsb.bfs)\,[A]\,[B]\,T \\ &= T\,[A]\,[B] \\ &= [A]\end{aligned}$$

$$\begin{aligned}[second(pair(A,B))] &= (\lambda p.pF)\{(\lambda fsb.bfs)\,[A]\,[B]\} \\ &= (\lambda fsb.bfs)\,[A]\,[B]\,F \\ &= F\,[A]\,[B] \\ &= [B]\end{aligned}$$

6.12 再帰呼出し (Recursive call)

ここまでで，λ 式で何でも表現できるような気がしてきたが，再帰呼出し (recursive call) は中々難しい．何故ならば，λ 式では関数に名前をつけないので，関数の中からストレートに「自分」を呼出す再帰呼出しはできない．

しかしながら，下の例で示す fixed-point operator というものを λ 式で表現することにより，可能である．

例 6.12.1: 階乗

$$f(0) = 1, f(n+1) = n \times f(n)$$

で階乗は再帰的に定義できる．これを λ 式で表そうとしてみる．まず，一旦引数にする．

$$g = \lambda fn.[n = 0 \text{ なら } 1, \text{さもなくば } n \times f(n)]$$

このような関数 g は，λ 式として定義可能．この g に f を引数として渡したものが，定義したい f．

このように， $f = g(f)$ となるような f を，g の不動点 (fixed point) という．

従って，不動点を返すような関数が λ 式で書ければよい．実際これは書けて，

$$Y \equiv \lambda f.((\lambda x.f(xx))(\lambda x.f(xx)))^{*24}$$

*21 f は first，s は second，b は boolean.

この Y が不動点を返すことを確かめてみると，

$$\begin{aligned} Yg &= (\lambda f.((\lambda x.f(xx))(\lambda x.f(xx))))g \\ &= (\lambda x.g(xx))(\lambda x.g(xx)) \\ &= \underline{g(\lambda x.g(xx))(\lambda x.g(xx))} \\ &= g(\underline{g((\lambda x.g(xx))(\lambda x.g(xx)))}) \\ &= g(Yg) \end{aligned}$$

となり，確かに．

これを用いて階乗を表すと，

$$\begin{aligned} [fact] &= g(Yg) \\ [fact(5)] &= g(Yg)5 \\ &= [5 = 0 \text{ なら } 1，\text{さもなくば } 5 \times (Yg)4] \\ &= 5 \times (Yg)4 \end{aligned}$$

6.13 λ 計算と，帰納的関数や TM は同じ計算能力

証明は省いてました．

6.14 同値性の決定不可能性 (Undecidability of equivalence)

2 つの λ 式が同値か否かを決定するアルゴリズムは存在しない．従って，[*25]正規形の存在は決定不能．

[*24] これには名前がついていて，Y combinator, fixed-point operator, paradoxical operator とか呼ぶ．

[*25] 正直分かりません

第 7 章

組合せ論理 (Combinatory Logic)

組合せ論理 (Combinatory Logic) は，計算モデルの一つであり，関数抽象化をせずに,[*1] 構造の変換だけで計算する体系をもつ.

7.1 組合せ項 (Combinatory Term)

組合せ項 (Combinatory term) は，以下のいずれかである.

- 変数
- プリミティブ: S, K, I combinator
- 組 $(E_1\, E_2)$

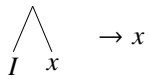
7.2 演算規則

7.2.1 プリミティブの役割

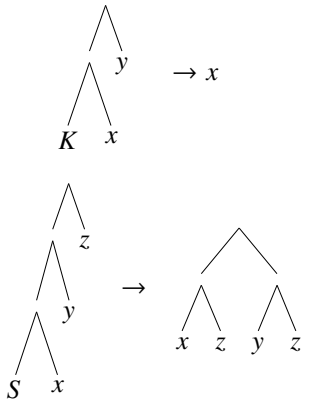
プリミティブ S, K, I は， 簡約化 (reduction) を行う．それぞれの行う演算は，以下の通りである.

- $(I\, x) \Rightarrow x$
- $((S\ x)\, y) \Rightarrow x$
- $(((S\ x)\, y)\, z) \Rightarrow ((x\, z)(y\, z))$

この規則は，二分木構造で見ると多少覚えやすい.



[*1] λ 計算と対照的



7.2.2 左方優先

左方優先性を持つ.

$$(A\,B\,C\,D) \to (((A\,B)C)D)$$

7.3 I なんていらないわ

$I = SKK$ なので，I がなくても体系が構築できる.[*2]実際,

$$\begin{aligned}((SKK)x) &\to ((Kx)(Kx))^{*3}\\ &\to (K\,x\,(K\,x))\\ &\to x\end{aligned}$$

なお，I を作るのは SKK だけでなく，例えば $I = SKS$ でもある.

7.4 組合せ論理は λ 計算と等価

組合せ論理は，以下の点で λ 計算と等価である.

- どんな λ 式に対しても等価な組合せ項がある.
- どんな組合せ項に対しても等価な λ 式がある.

実際，次のように λ 式と組合せ論理式を対応させると，任意の λ 式を組合せ論理式に変換でき，またその逆もできることが分かる.

(λ 式 L から組合せ項 c への変換を $T[L] \to c$ という風に表記する．)

- $T[x] \to x$

[*2] Emacs 使いの皆さんにはお馴染みの skk は，ここから名前をもじったそうです.

[*3] 右側の $(K\,x)$ は，K がオペレータを 2 個取らないからといって，別に illigal な訳ではない．あくまでも，K の持つ計算規則によって簡約化できないだけ.

- $T[FX] \to (T[F]\,T[X])$
- $T[\lambda x.E] \to (K\,T[E])$
  ただし，E の中に x が出現しないとき．
- $T[\lambda x.x] \to I$
- $T[\lambda x.\lambda y.E] \to T[\lambda x.T[\lambda y.E]]$ [*4]
- $T[\lambda x.(F\,Y)] \to (S\,T[\lambda x.F]\,T[\lambda x.Y])$

例 7.4.2: 変換規則適用の例

$\lambda x.\lambda y.(y\,x)$ を，組合せ論理式に変換する．

$$
\begin{aligned}
T[\lambda x.\lambda y.(y\,x)] &\to T[\lambda x.T[\lambda y.(y\,x)]] \quad \because 5\\
&\to T[\lambda x.(S\;T[\lambda y.y]\,T[\lambda y.x])] \quad \because 6\\
&\to T[\lambda x.(S\;I\,(K\,T[x]))] \quad \because 3,4\\
&\to T[\lambda x.(S\;I\,(K\,x))] \quad \because 1\\
&\to (S\;T[\lambda x.(S\;I)]\,T[\lambda x.(Kx)]) \quad \because 6\\
&\to (S\;(K\,T[(S\;I)])\,T[\lambda x.(Kx)]) \quad \because 3\\
&\to (S\;(K\,(S\;I))\,(S\;T[\lambda x.K]\,T[\lambda x.x])) \quad \because 1,6\\
&\to (S\,(K(S\;I)))(S\;(K\;K)\,I) \quad \because 3,4
\end{aligned}
$$

逆変換も見てみる．

$$
\begin{aligned}
(S\,(K(S\;I)))(S\,(K\;K)I)\,x\,y &\to ((K(S\;I))\,x)((S\,(K\;K)I)x)\,y\\
&\to (S\;I)(((K\;K)x)(I\,x))\,y\\
&\to (S\;I)(K\,x)\,y\\
&\to (I\,y)((K\,x)\,y)\\
&\to y\,x
\end{aligned}
$$

7.5 真偽値の表現

P を，Pxy が真なら x，偽なら y となるように作る．真を表す記号を T，偽を表す記号を F とすると，それぞれ以下のように組合せ論理で構成できる．

$$
\begin{aligned}
T &= K\\
F &= (K\;I)
\end{aligned}
$$

証明 7.5.1 *T，F が組合せ論理で構成できる証明*

[*4] ここにも適用条件はあるはずですが，先生が忘れてらっしゃいました．

$$(T\,x\,y) = (K\,x\,y) = x$$
$$(F\,x\,y) = (K\,I\,x\,y) = (I\,y) = y$$

7.6 組合せ論理の性質

λ 式と組合せ論理は等価なので，組合せ論理は当然 λ 式と同様な性質を持つ.

- これ以上簡約化できない項があれば，それを 正規形 と呼ぶ.
- 正規形の存在は決定不能.
- 項のどの部分から簡約化しても，更に簡約化を進めると同じ項を得られる (合流性). 図 6.1 を参照.

例 7.6.3: 正規形を持たない組合せ論理式

$$\begin{aligned}(S\ I\ I\ (S\ I\ I)) &\to (I(S\ I\ I)(I\ (S\ I\ I))) \\ &\to (S\ I\ I\ (S\ I\ I))\end{aligned}$$

この項は正規形を持たない.

第8章

計算量理論

計算量理論 (Computational Complexity Theory) は，特定のアルゴリズムの性能解析手法である．

8.1 漸近的計算量 (Asymptotic complexity)

漸近的計算量 (Asymptotic complexity) とは，処理対象のデータ量が増えたときに，計算量はデータ量のどのような関数になるのかを扱う指標である．

∵データ量小さいときは，どうせ計算量も小さいので気にする必要はない．

問題サイズ (問題となるサイズ．例えば，処理対象のデータ量そのもの) を n とする．また，計算量を $T(n)$ とする．通常 $T(n)$ は単調増加．

8.1.1 Ω 記法 (Ω-notation)

Ω 記法 ($\Omega - notation$) は，計算量の漸近的下界を示す．あるアルゴリズムが $\Omega(f(n))$ という計算量を持つとは，

$$\exists c \exists n_0 \forall n \geq n_0 \quad (T(n) \geq cf(n)) \qquad (c \text{ は定数})$$

が成立するということである．(参考: 図 8.1)

8.1.2 O 記法 (big-O notation)

O 記法 (big-O notation) は，計算量の上界を示す．$O(f(n))$ の計算量とは，

$$\exists c \exists n_0 \forall n \geq n_0 \quad (T(n) \leq cf(n))$$

が成立するということである．

8.1.3 Θ 記法 (Θ-notation)

Θ 記法 (Θ-notation) により，f と g の上界と下界の関数形が一致する (上界と下界が定数系の違い) とき，すなわち

$$f(n) = O(g(n)) \quad \text{かつ} \quad f(n) = \Omega(g(n))$$

のとき，

$$f(n) = \Theta(g(n))$$

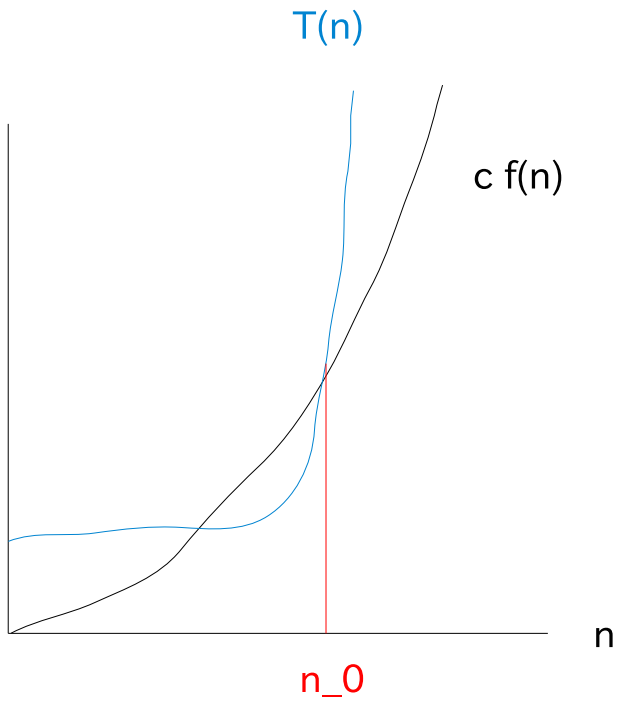


図 8.1　Ω 記法の式のイメージ

と表記する.

それぞれの c, n_0 は別々に考えて良い.

8.2　入力データと計算量

単純でないアルゴリズムの計算量指標は，入力データの量だけでなく，質によって計算量が異なる場合がある.

例 8.2.4:　クイックソート

クイックソートは，ピボットの位置によって，計算量が最悪で $T(n) = cn^2$ になってしまう.

8.2.1 最悪の場合の計算量 (Worst-case complexity)

最悪の場合の計算量 (Worst-case complexity) は，「どんなデータに対しても，最悪でこの計算量を保証しますよ」というようなものである．計算資源に絶対的制約がある際には不可欠である．

例 8.2.5: リアルタイム制御

原子炉や証券取引を制御するプログラムを考えてみると，「この計算は普通 1ms で終わりますけど，ひょっとしたら 1s 掛かる時もあります．まぁ滅多にそんなことは起きませんけどね.」は通用しない．最悪の場合の計算量を見積る必要がある．

8.2.2 平均計算量 (Average-case complexity)

平均計算量 (Average-case complexity) は，様々な入力データに要する計算量の平均である．この「様々な入力データ」には，分布の仮定が必要．その仮定に対してのみ平均計算量が出せる．

8.2.3 償却計算量 (Amortized complexity)

償却計算量 (Amortized complexity) は，複数操作[*1]の系列の総計算量．8.5 に詳細を記述する．

8.3 計算量クラス

8.3.1 クラス P(Polymial time)

決定性チューリングマシン (やそれと等価なもの) によって，問題サイズの多項式関数で表すことのできる計算量のアルゴリズムが存在するような問題のクラスのことを， クラス P と呼ぶ．

8.3.2 クラス NP(Non-deterministic polynomial time)

非決定性チューリングマシン (やそれと等価なもの) によって，問題サイズの多項式関数で表すことのできる計算量のアルゴリズムが存在するような問題のクラスのことを， クラス NP と呼ぶ．

決定性チューリングマシンは，非決定性チューリングマシンの特別な場合に過ぎないので，

$$NP \subseteq P$$

である．

8.3.3 NP 困難 (NP-hard)

どんな NP の問題でも，多項式時間の計算量でその問題に帰着できるような問題のクラスのことを， NP 困難 な問題のクラスと呼ぶ．

[*1] 複数操作とは，例えば，AVN 木での「木のバランスを方操作・挿入」をまとめたもの．AVN 木は，木のバランスが前提のデータ構造なので，これらをまとめて考える価値がある．

8.3.4 NP 完全 (NP-complete)

NP に属す NP-hard な問題のクラスは， NP 完全 (NP-complete) な問題のクラスである．すなわち，

$$NP - complete = NP \cap NP - hard$$

8.3.5 P ≠ NP ?

P ≠ NP であるという予想が立っているが，これは未解決問題である．

8.3.6 P に入るか否か不明の問題

P ≠ NP を仮定すると，NP-complete なら P には入らない．一方で，

- 素因数分解
- グラフ同型問題

などは，P に入るか否か不明の問題である．

8.3.7 Time-Hierarchy Theorem

どのような時間計算量の問題クラスに対しても，それより複雑な計算量の問題が存在する．

8.3.8 Space-Hierarchy Theorem

(板書なし)

8.4 多倍長カウンタの例

例 8.4.6: 多倍長カウンタ

1 word(通常 32bit) で表現しきれない数値を整数の配列で表現することがある．整数の配列を $a[i]$ とすると，

$$n = \Sigma_{k=0}^{n-1}\{a[k] \times 2^k\}$$

これを用いた，下記のアルゴリズム increment の計算量を考える．

worst-case
: carry はいくらでも出せるので，最悪ではビット数に比例．例えば，0111 から 1000 にカウントアップするときに最悪.

average-case
: 2^n までカウントすると，合計では 2^n のオーダー 1 回あたりの平均計算量は $O(1)$．ちゃんと

計算すると，平均計算量は 2 と求められるらしい．

アルゴリズム increment

```
void increment(int a[])
{
    int k;
    for (k = 0; a[k] == 1; k++)
        a[k] = 0;
    a[k] = 1;
}
```

確かに最悪の場合ビット数に比例した計算量だが，最悪の場合はカウントアップを初めてからかなり後にしか起きない． 平均計算量では，各状態での操作の手間について何も保証できていない ので，このようなミスマッチが生じる．

最初の方の操作の手間は，ビット数 n に依存しないことを表現できる枠組みが欲しい．それが下記の償却計算量である．

8.5 償却計算量 (Amortized complexity)

「償却」に込められた意味は，「実質的に意味のある値」みたいな感じ．減価償却も同じ用法. [*2]
この考えのもとで，先程のアルゴリズム increment を見直してみる．

- たくさん繰り上げが生じると，その後しばらくは繰り上げが出にくい．
- たくさんの繰り上げを「投資」と思えば，繰り上げにより生じる下位のビット 0 の連続は「資産」．コストはこの資産を利用するときに払う．

また，「引当金」[*3]と同じ意味での引当ての考え方で見てみる．

- 実際の「コスト」は $0 \to 1$, $1 \to 0$ が単位．
- 1 のビットを作ると将来 carry を生む原因になるので，$0 \to 1$ の反転のコストを 2 と考える．実際のコスト 1 に引当分の 1 を加えたという計算．
- $1 \to 0$ は引当済み．つまり，コストは $0 \to 1$ になったときに計上済みなので，$1 \to 0$ のコストは 0 と考える．

これによると，カウントアップ時に行う操作のコストは，

- [$1 \to 0$ のコスト] × [キャリービット数] = 0
- [$0 \to 1$ のコスト] × [1 ビットだけ] = 2

[*2] ただし，減価償却は前払いであるのに対し，カウントアップの例は後払い (値が 0 に近いうちはあまりコストが掛からない)．この点で，先生は「あまり良い訳ではない」と仰ってました．どうでもい (ry
[*3] 気になったらググってください

表 8.1　カウントアップの計算量

操作		実際の手間	コスト累計	引当 (償却) 計算量	残
$0 \to 1$	0001	1	1	2	1
$1 \to 2$	0010	2	3	4	1
$2 \to 3$	0011	1	4	6	2
$3 \to 4$	0100	3	7	8	1
$4 \to 5$	0101	1	8	10	2
$5 \to 6$	0110	2	10	12	2
$6 \to 7$	0111	1	11	14	3
$7 \to 8$	1000	4	15	16	1

(毎回のカウントアップで，$0 \to 1$ になるビットはいつもひとつであることに注意．) 従って，カウントアップのコストは常に 2 である．

例 8.5.7: 配列の拡張

配列は，予めサイズを決めるのが難しい．従って，最初は小さなサイズで確保して，必要に応じて拡張していくことが多い．しかし，配列では連続領域が必要なので，拡張時に連続領域が足りなくなったら，別の連続領域をとれる場所に現在の中身をコピーしなければならない．この計算量を考えてみる．

サイズが足りなくなったら倍に拡張するとする．拡張して 2^n 要素とした後，次の拡張までにまた 2^{n-1} 要素を追加できる．この間に，次回の拡張に要する手間を引当てていると考えられる．

定数 a, c を使うと，拡張時の割付けの手間は $2^{n+1}a$，コピーの手間は $2^n c$ なので，引き当てるべき計算量は，

$$\frac{2^{n+1}a + 2^n c}{2^{n-1}} = 4a + 2c$$

である．つまり，配列は拡張時には確かに計算量は大きくなるが，1 回の操作あたりの手間は定数オーダであると言える．

第9章

データの複雑さ (Data Complexity)

データの複雑さを表す尺度として，コルモゴロフ複雑性 (Kolmogorov complexity) がある．これは，記号長を生成するのに必要なプログラムの最小長である．

例えば，’101010101...10 (1 万桁)’ は，大きなデータであるが，単純なループで書けてしまうので，データの複雑さは小さいと言える．

9.1 定式化

プログラム p をある計算機 m 上で動作させた出力を $m(p)$ とすると，ある記号列 s に対して $m(p) = s$ となる最短のプログラム p の長さは，

$$K_m(s) = \min_{p|m(p)=s} (length(p))$$

と表せる．これがコルモゴロフ複雑性である．

9.2 性質

- s に規則性があれば，s を定数として含むプログラムより短いプログラムで生成可能．
- 2 つの異なる言語でも，一方で他方のエミュレータを作れるなら，複雑性はエミュレータのプログラム長 (定数) しか異ならない．

この性質は，データ圧縮の可能性に関係している．つまり，コルモゴロフ複雑性が低いデータは高い圧縮率で圧縮できる．

ただし，

> **定理 9.2.1**
>
> コルモゴロフ複雑性は計算不能．

である (計算不能とは，どのような入力に対しても出力を出すようなアルゴリズムは存在しないということであった)．

証明 9.2.1 コルモゴロフ複雑性は計算不能であることの証明

どのようなデータ s に対しても $K(s)$ を導くようなアルゴリズムの存在を仮定して，矛盾を導く．

$K(s)$ が計算できるのだから，プログラム長 l のプログラムでは生成できない様な最短の記号列を求めることができる．それは，記号列を短い方から系統的に生成して，$K(s)$ が l を超える最初の記号列を返すことにより実現できるはずである．

このようなプログラムを $C(l)$ とし，その長さを L_C とするとする．この関数を引数 l_0 で呼出した結果を返すプログラムを考える．この関数を引数 l_0 で呼出した結果を返すようなプログラムを考えると，その長さは

$$L_c + \log_2(l_0) + \alpha$$

となる．この長さのうち，$\log_2(l_0)$ 以外は定数である．従って，l_0 を大きくしていくことにより，いつかは

$$l_0 > L_C + \log_2(l_0) + \alpha$$

となるはずである．このプログラムは $C(l_0)$，即ち l_0 の長さのプログラムでは生成できない記号列を返すはず．それなのに長さが l_0 未満であるため，矛盾である．

第 10 章

並行計算のモデル

コンピュータで並行というと，次の 2 つの意味を指す場合がある．

Concurrency
: 論理的並列性を表す．前後関係が定義されていない 2 つの計算についての言葉．論理的正当性に関係する

Parallelism
: 物理的並列性を表す．実際に並列に動作しているかどうか．性能に関係する．

Concurrency が保証されなければ Parallelism があっても意味がないことに注意．本章では，前者の Concurrency について議論する．

10.1 ペトリネット (Petri Net)

ペトリネット (Petri Net) は，並行システムの古典的モデル．並行システムを有向二部グラフで表現する．Carl Adams Petri の提唱である．

10.1.1 構成要素

トークン (token)
: 処理のアクティビティを表す．例えば，プロセスやスレッド．ペトリネットの中をトークンが動き回る．

場所 (place)
: 処理の状態を表す．

遷移 (transition)
: 場所から場所へ移る際の中継点．

辺

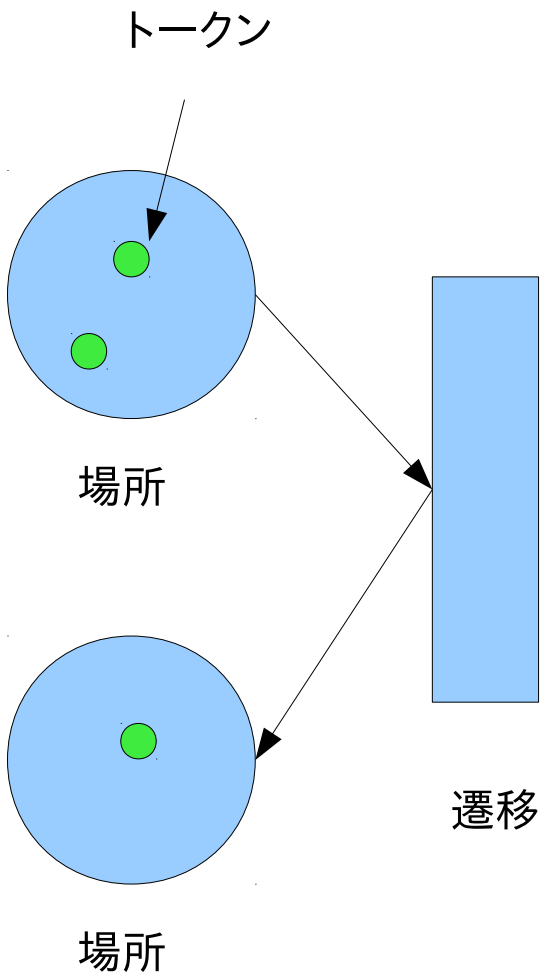


図 10.1　ペトリネットの構成要素

「場所 → 遷移」，または「遷移 → 場所」を表す繋ぎ目．「場所 → 場所」，或いは「遷移 → 遷移」がないのが有向” 二部” グラフである所以．

10.1.2　動作規則

- ある遷移に入る全ての辺の視点にトークンがあれば，その遷移が発動可能になる．(join) (図 10.2)
- 遷移が発動すると，遷移から出る辺の終点の場所全てにトークンがひとつ増える．遷移に入る辺の視点のトークンはひとつ減る．(fork) (図 10.3)
- 可能な遷移が複数ある時，どの遷移が発動するかは非決定的．(図 10.4)

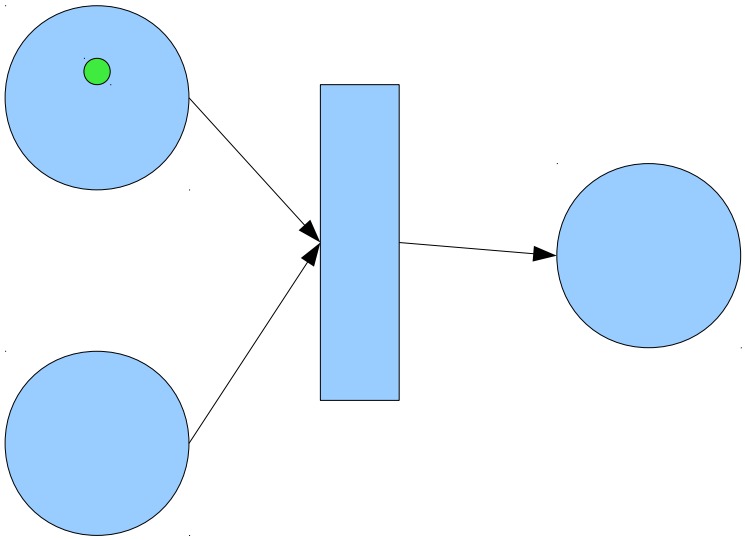


図 10.2　待ち合わせ (join)

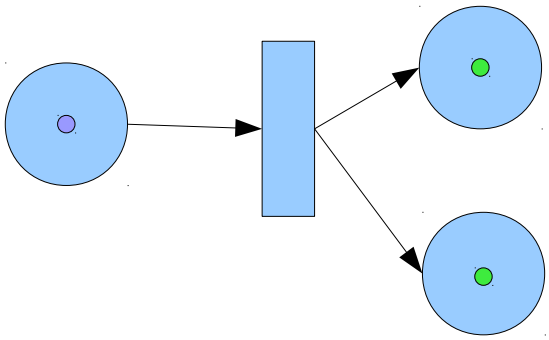


図 10.3　分岐 (fork)

10.1.3　並行版の有限オートマトンとしての見方

ペトリネットは，有限状態オートマトンの並行版としても見られる．逐次の state machine は，次の特徴を持つペトリネットとモデル化できる．

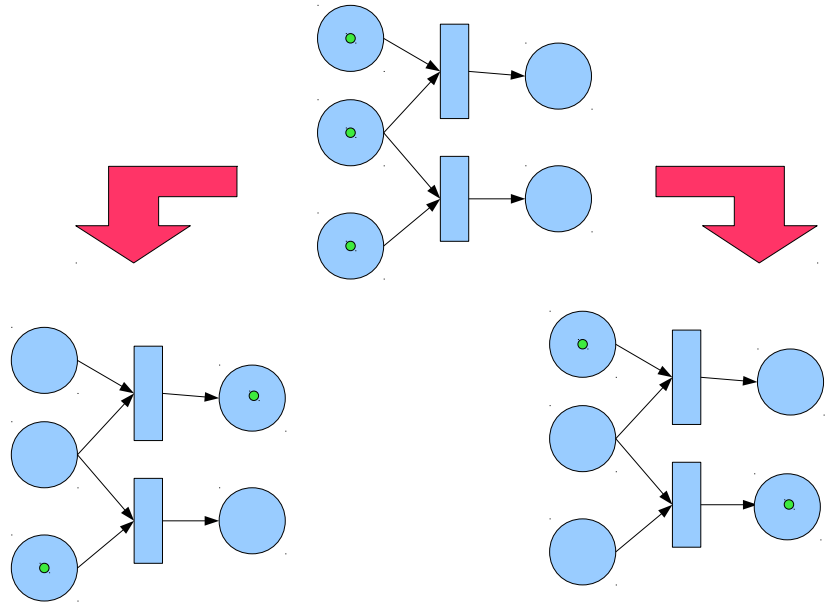


図 10.4　非決定性

- 全ての遷移が単一の入出力
- トークンが増減しない
- 初期状態でトークンはひとつ

10.2　プロセス計算 (Process Calculus)

[*1]並行計算を抽象化して代数的に扱う枠組みの総称．独立したプロセスがメッセージを交換して通信するようなモデル．プロセス計算は，プロセスの基本動作と組合せ方 (通信) によって表現される．

10.2.1　並列合成 (Parallel composition)

$$P|Q$$

で，プロセス P, Q は並行動作していることを表す．

10.2.2　通信と同期 (Communication and Synchronization)

通信路を経由．通信できる相手は，通信路 (channel) がつながっている先．

[*1] この節については結構説明が浅かったです．残念．

10.2.3 メッセージ通信・受信

$$\bar{c} < x > \quad c < x >$$

オブジェクト指向のモデルだったりするらしい.

10.2.4 同期通信路 (Synchronous channel)

メッセージが受信されるまで送信は終わらない. [*2]

10.2.5 非同期通信路 (Asynchronous channel)

送信はすぐ終わり，受信は後で可能. [*3]

10.2.6 逐次合成 (Sequential composition)

メッセージ送受の順序関係．例えば,「受信してから送信する」など.

10.2.7 簡約化規則 (Reduction rules)

(記述なし)

10.3 パイ計算 (π calculus)

プロセス計算の代表的枠組み．並行版の λ 計算． [*4]

10.3.1 基本要素

$P|Q$
: 並行動作

$c(x).P$
: c から x を受信してから P を実行

$\bar{c} < x > .P$
: c に x を送信してから P を実行．$c?(x)$ や $c!(x)$ という表記もある.

[*2] C とかの関数呼出しは同期通信です.
[*3] ABCL とかいう言語は非同期通信を実装した最初の言語だそうです．http://en.wikipedia.org/wiki/Actor-Based_Concurrent_Language
[*4] 萌えますね

$!P$

P の無限の複製．つまり，P というプロセスを無限に fork する (できる) ことを表す．

0

空のプロセス (nil process [*5])

$(\nu c)P$

通信路 c を生成して，それを使って P を実行．$newc.P$ という表記も．

例 10.3.8: パイ計算の実例

$$(\nu z)((\nu x)(\bar{x}<z>.0|x(y).\bar{y}<x>.x(w).0)|z(v).\bar{v}(v).0)$$

以下で，緑字を (1-1)，赤字を (1-2)，青字を (2) と表記する．
この式で表される動作の概略は，次のようなものである．

- 通信路 x を作り，二つのプロセス (1-1)，(1-2) を動かす．
  (1-1) x に z を送る

  (1-2) x から y を受信して，y に x を送り，x から w を受信
- (2) が通信路 z から v を受信して，そこに v 自身を送る

より詳細に見てみる．

- (1-1) から送った z を (1-2) は y に受信．
  $$(\nu z)((\nu x)(0|\bar{z}<x>.x(w).0)|z(v).\bar{v}<v>.0)$$
- (1-2) が z に x を送り，(2) は v に受信．
  $$(\nu z)((\nu x)(0|x(w).0)|\bar{x}<x>.0)$$
- (2) が x に x を送り，(1-2) が受信
  $$(\nu z)((\nu x)(0|0)|0)$$
- 終了

10.3.2 特徴

パイ計算には，次のような重要な特徴がある．

- 通信路を通信できる．
- それ自体でチューリング完全．つまり，パイ計算だけであらゆる計算を表現可能．

[*5] nil って nihil の略から生まれたんですって．関係ないけど豆知識

例 10.3.9: 真偽値

$$True \quad c(t).c(e).c(x).\bar{t} < x >$$
$$False \quad c(t).c(e).c(x).\bar{e} < x >$$

2 つの通信路とデータ x を受け取って，True は第 1 の通信路に，False は第 2 のデータ x を送る．