

情報基礎教育のためのアルゴリズム記述形式の提案

唐澤 博

karasawa@esi.yamanashi.ac.jp

山梨大学 工学部

〒400 甲府市武田 4-3-11

高校や大学で実施されている情報基礎教育でアルゴリズムを学習するための道具立ては、フローダイアグラムかプログラミング言語を適用している例が多い。しかしアルゴリズムが複雑になるとフローダイアグラムでは見通しが悪くなり、プログラミング言語ではアルゴリズムの本質を離れた詳細の記述を多く必要とされたり、他言語による記述が読みとれないといった学習の妨げが存在する。こういった問題に対処するために、高校から大学までのアルゴリズム学習に一貫して適用可能な教具として、フローダイアグラムの代替程度のシンプルさをもつ疑似コード記述形式を提案する。

A Proposal of Algorithm Description Format for Fundamental Education of Computer Science

Hiroshi Karasawa

Faculty of Engineering, Yamanashi University

4-3-11 Takeda, Kofu, Yamanashi, 400 Japan

It is general that a flow diagram or any of programing language is applied to learning on algorithm in high schools and universities. But the more complex algorithm to learn is, the less readable the flow diagram is. By a programing language, a student is required to describe syntactic details rather than the essence of algorithm. Or he cannot read a algorithmic description in a foreign programing language he has not learnt. As the solution for such obstacles, the author proposes algorithm discription format which is so simple pseudo code as to replace a flow diagram.

1 はじめに

アルゴリズムの記述にはプログラミング言語よりも抽象度が高くかつ単純なのが向いている。その意味では、プログラミング言語よりフローチャートの記述レベル向きである。

現在の学習課程の主流が、いずれかのプログラミング言語を習得して後にアルゴリズム教育に進んでいる。しかし、プログラミング言語では粒度が細か過ぎてアルゴリズム学習には不向きである。

大学の情報系におけるアルゴリズム教育では、フローチャートではなく PASCAL 言語風の疑似コードを適用することが行なわれている。これはアルゴリズムがある程度複雑になってくるとダイアグラム表示ではかえって繁雑になるため記号系で記述したほうが見通しがよいということが理由の一つにある。しかし、そのような疑似コードは、そのままコンピュータ上で実行してその正しさを検証することができないという問題がある。検証するにはいずれかのプログラミング言語に翻訳する作業が要求される。筆者の授業において見られることは、この翻訳作業が純粋なアルゴリズム学習に対するハードルとなっている。

フローチャート・レベルの記述粒度の疑似コードで、かつこれを検証する評価系(プログラム)が存在することが、効果的なアルゴリズム学習に必要なものであるという必要性から、筆者はアルゴリズム学習専用記述形式を設計した。この記述形式を、以降 ADF (Algorithm Description Format) と呼ぶことにする。

ADF のような記述教具の必要性は、高校における情報基礎教育にも言えることであり、大学へと連なる学習課程の整合性を考慮する上でも重要であると考ええる。

職業科における情報教育をみると、既に永い実績をもつ商業科、工業科では、基礎段階と実務指向段階の2段階の科目立てになっており、前者は BASIC から COBOL へ、後者は BASIC から FORTRAN へというステップで学習しているところが多い。一方、新学習指導要領に基づき導入された、水産、農業、家庭、看護の各学科の情報関係基礎科目においては、情報教育はユーザ的な内容にとどまり、市販されている教科書を調査した範囲では、みな BASIC を題材にしている。この、商業、工業の基礎段階での学習に BASIC ではなく ADF を用いたアル

ゴリズムの学習を実施すること、他の職業科でも、BASIC と抱き合わせて使用しているフローチャートを ADF に代替すること、そして普通科の数学に導入されている情報関連の学習内容のうち「コンピュータと算法」の、算法の学習には特定の言語によらず ADF を適用して実施すること、以上の提案を行なうものである。

高校段階におけるあらゆる学科の情報教育に ADF のような共通語が導入されるメリットは大きい。ソフトウェア教育の上級課程では、アルゴリズムの学習が主体になってくることが多いが、高校段階でさまざまなプログラミング言語でこれ思考する習慣がついている場合と、統一された表記で思考訓練がされている場合とでは、その円滑さに差がでる。さらに入試問題でも、学科ごとに異なった言語を考慮するような繁雑な必要性もなくなる。

ADF は、Wirth [1] によって周到に設計された教育用言語である PASCAL 言語をシンプルにしたような仕様をもちながら、一般的なプログラミング言語に共通した核を主体とした記述体系である。このため、ADF による思考訓練は、そのままスムーズに特定のプログラミング言語による記述作業に適用できると思われる。

2 情報基礎教育におけるニーズ

● アルゴリズム学習の重要性

アルゴリズムの学習目標は、与えられたデータを目標の形に加工するための手順を基本的な操作の組合せとして表現する点にあり、正確な結果と効率的な手順を保証する定型アルゴリズムを習得することが中心となる。この意味で、多分に数学的な色彩を有する。「基礎」としては、直面する問題を分析してその解法をアルゴリズムに表現する訓練をする方が本質的である。プログラミングという作業は多分に専門的であり、誰もが要求されることではないが、直面する問題を分析してその解法をアルゴリズムに表現することは、情報系業務でない多くの現場で将来的に要求される技能になろう。

● アルゴリズムを説明、議論する共通の基盤

大学の入学試験においては、高校での習得言語が様々である問題作成が困難になる。高校教育の段階で統一した記述形式のもとに学

習が行なわれていれば、入学試験のみならず入学後のアルゴリズム教育もやり易くなる。

- フローダイアグラムに替わる記述道具
アルゴリズムを直接的に記述する道具としては、フローチャートが主流で、ボックス・ダイアグラムや PAD 図、HIPO 図、PDL のような疑似コード等が使われる場合がある。比較的単純なロジックの場合は図による記述が直視性の点で優れているが、若干複雑になってくると図が繁雑になり、記号体系の方が見通しがよい。単純からやや複雑なロジックまでをカバーする記述道具としては、非常に単純化された PDL 系が適していると思われる。
- 基本から段階的に学習できる道具立て
情報基礎教育のための記述形式に要求されることは、一般のプログラミング言語と異なり、記述効率やメモリ使用効率といった点よりも基本の習得が優先される。基本原則を習得し易い配慮がなされており、その基本を発展させて、プログラミング言語に採用されているような記述へ導いていくような方針の設計になっている必要がある。
- 学生の記述したロジックの検証
学習場面で学生が ADF を用いて記述したロジックが期待する結果を生じるか検証可能な手段が提供されるべきである。ADF を説明の道具として使うだけでなく教具として役立たせるには、検証系を伴う必要がある。検証系とは、ADF を解釈実行するシステムのことである。

3 アルゴリズム記述の基本要素

アルゴリズム記述のありかたを議論するために、ソフトウェア工学の成果を振り返ってみる。

3.1 構造化

構造化プログラミングは情報基礎教育においても重要視されており、ロジックを明瞭に表示するための適切な構文を用いることや、goto 文使用の害がそこでの論点になっている。ソフトウェア工学の分野では 1960 年代に盛んに議論がなされ、いまでは一応の決着が見られている。

Dijkstra [2] によって goto 文を追放した構造化プログラミングが提唱されたが、その後の研究によっ

て、(1)goto 文を使わないと、ソフトウェア・スイッチ用の論理変数を新たに使用するとか、多重ネストからの大域脱出可能な多段 exit 構造をもつ記述系を適用しない限り、記述できないロジックが存在する、(2)goto 文を使用した方が良み易くなる場合もある、といった性質が明らかにされ、goto 文についての結論は、goto 文を用いて読み易いプログラムを書くこともできれば、goto 文を用いないで読み難いプログラムを書くこともできるというところへ落ち着いた [3]。

構造化や goto-less の議論は、良み易さの向上に対する工夫に由来するものであって goto 文がロジックの文脈独立に有害であるという意見は本末転倒である。goto 文の適用上の留意点としては、それなるべく使用しないようにし、使用する場合でも、大局的な実行の流れに沿う分岐は逆らう分岐よりも害がないとされている。

3.2 階層化

階層的プログラミングはロジックを読み易いものにするもう一つの手段である。ここでは処理の記述を top-down に行ない、段階的詳細化の手順を踏むことと関連付けられる。ひとつのモジュールの記述はせいぜい A4 で 1 ページ程度までにとどめるよう推奨される。この手法は認知科学的には、人間の管理量の限界をカバーするものと解釈できる。この観点では、上位モジュールの記述は下位モジュール内の詳細のチャンキングとみなせよう。また、これをもって階層的プログラミングの読み易さの理由付けとすることもできるだろう。

階層化に必要な記述系への要請は、モジュール単位で記述し易く、上位モジュールほど概然的な表現で処理単位を記述できることなどが考えられる。

4 ADF の提案

ADF は、以下のような方針のもとに設計された。

1. 最小の仕様は、フローチャートを代替する程度の制御構造系とする。
2. 初級単元から上級単元まで連続的に適用できるよう、仕様を基本と拡張に分けて定義する。
3. 教育用に設計された PASCAL に近い仕様とし、PASCAL よりもさらに基本的な構造を提供する。

4. ユーザが導入する変数名、手続き名、関数名は、任意の長さの自然言語表記を許して読み易さを向上させる。
5. goto に相当する記述は、前述の義論をガイドンスとして適用することを認める。
6. ロジック組み立て教育の道具としても適用可能なように、検証系プログラムの作成を考慮して、式の記述形式も規定しておく。
7. 関数と手続きの違いを意識させるような構文にする。関数は必ず値を返し、手続きは値を返さず副作用を起こすことを理解させる。
8. 記述は読み易さの点から1実行1行以上とし、マルチステートメントを避ける。
9. 基本的な変数の構造としては配列までを定義する。配列は繰り返し操作の対象にしばしばされることがその根拠である。
10. アルゴリズムの記述とは、変数群を目的の値になるよう操作する手順を示すものであることを認識させる意味で、操作対象の変数群をまず明示的に列挙し、これに続いて変数操作の過程が記述されるものとする。これは、一般的なプログラミング言語でも同様になっている。
11. プログラミング言語による記述で初期のハードルとなる変数の型宣言は省略する。データは数値、文字列、論理値のみの区別を設ける。
12. 一般的なアルゴリズムの説明でよく使われる集合表現も直接的な表現を可能にする。

4.1 制御構造

ADF の制御構造の3基本要素を示す。

表1 基本制御構造

種別	表記
連結	$S_1/S_2/\dots/S_n$
条件分岐	if α then β_1 [else β_2] if#
繰り返し	loop θ loop#

連結は改行 / で終る任意の実行文 S の列挙で、記述の上方から下方に向かって実行される。条件分岐の α は条件式、 β は連結と同様の実行文の列挙である。繰り返しの θ は繰り返し実行すべき実行文の列挙である。ここでの実行文には基本制御構造自身もなり得る。繰り返しの θ に以下の2種類の特別な実行文を含めることができる。

表2 繰り返しに関わる実行文

種別	表記
繰り返しの続行	nextloop
繰り返しの脱出	exitloop

nextloop が実行されると、この時点で loop の最初に分岐して次の繰り返しに移る。exitloop が実行されると、この時点で loop から脱出する。これらを条件分岐の実行文に用いることで、while 文や repeat 文や for 文に相当する条件付き繰り返しの基本的な表現を実現する。PASCAL 言語に例をとると以下のようなになる。

```

• while  $\alpha$  do  $\theta$ 
  loop
    if not  $\alpha$  then exitloop
     $\theta$ 
  loop#

• repeat  $\theta$  until  $\alpha$ 
  loop
     $\theta$ 
    if  $\alpha$  then exitloop
  loop#

• for  $i:=n$  to  $m$  do  $\theta$ 
   $i \leftarrow n$ 
  loop
    if  $i>m$  then exitloop
     $\theta$ 
     $i \leftarrow i+1$ 
  loop#

```

ADF の方が記述は一般に冗長になるが、ADF は基本を学習する道具としての位置付けから、原理が明確に示される点に重きが置かれる。このような繰り返し構造は、while, repeat 等よりも一般的な構造として Knuth [4] により示された。

while 文, repeat 文, for 文といった構造は、上記のような学習を経てから段階的に導入されるべきマクロ的構造と考える。現状のように、これらの相互の相違や共通性の認識を素通りして、半ば丸暗記的に学習させることから基本が理解されないという問題から回避できる。

後述するように、より高度なアルゴリズム記述では記述全体を簡潔にして可読性を向上させる目的で、前述のマクロ的構造を ADF の仕様に拡張的に

採り入れている。ここで重要なことは、もっとも基本的な事柄からの出発を可能にすることである。

なお、# 記号は「～の終り」の意味をもつ。ADFを読み上げるときは、たとえば loop# は「ループの終り」あるいは loop end と読む。

4.2 実行文

実行文は表3に示された8種類である。ただし前節で導入した繰り返し制御に関わる2種類の実行文はここに含まれていない。

表3 基本実行文

種別	表記
代入	変数←式
キー入力	input 変数の並び
画面表示	output 式の並び
画面改行	newline
手続き呼びだし	\$ 手続き名 式の並び \$
関数呼びだし	\$ 関数名 (式の並び) \$
呼び出しの戻り	return(式)
分岐	goto ラベル

画面改行は明示するものとし、暗黙の改行操作を排除する。手続き呼びだし、関数呼びだしの両端を\$記号で挟む記法は、フローチャートの呼び出しの箱のイメージからのアナロジーである。goto文の分岐先のラベルは、操作の頭部に任意につき予約語以外の任意の文字列の末尾が半角のコロンで終わっているものとする。

4.3 式

式は図1のように定義される。式定義中の演算子は表4で定義される。図の記号で|は選択を、{...}は選択範囲を、[...]は省略可能を示す。

式 ::=	単純式 { = < > <= >= } 単純式
単純式 ::=	[{ + - }] 項 単純式 { + - or } 項
項 ::=	因子 因子 { * / % and ^ } 因子
因子 ::=	符号なし数 文字列 変数 (式) not 因子 関数呼び出し true false

図1 式の定義

数は整数型、実数型といった区別をしない。文字列は1文字定数と文字列定数の区別をしない。数値と文字列との計算はできないため、これらの区別まで簡略化することはしない。

演算子を表4に示す。表中の順位とは結合順位である。これら演算子は、一部を除いてPASCAL言語の定義から援用した。

表4 演算子の定義

順位	演算子と項の型	結果型	意味
1	not 論理型	論理型	論理否定
2	論理型 and 論理型 数値型 * 数値型 数値型 % 数値型 数値型 / 数値型 数値型 ^ 数値型	論理型 数値型 数値型 数値型 数値型	論理積 算術積 整数商 算術商 べき乗
3	論理型 or 論理型 数値型 + 数値型 文字列 + 文字列 数値型 - 数値型	論理型 数値型 文字列 数値型	論理和 算術和 文字列の連結 算術差
4	数値型 = 数値型 文字列 = 文字列 数値型 <> 数値型 文字列 <> 文字列 数値型 <= 数値型 文字列 <= 文字列 数値型 >= 数値型 文字列 >= 文字列 数値型 < 数値型 文字列 < 文字列 数値型 > 数値型 文字列 > 文字列	論理型 論理型 論理型 論理型 論理型 論理型 論理型 論理型 論理型 論理型 論理型 論理型	数値一致 文字列一致 数値不一致 文字列不一致 数値(以下) 文字列(以下) 数値(以上) 文字列(以上) 数値(未満) 文字列(未満) 数値(超える) 文字列(超える)

4.3.1 絶対値と文字列長

| 記号で挟んだ記法 | 式 | について、式の結果が

- (1) 数値型なら絶対値
- (2) 文字列なら文字列長を与えるとする。

4.4 モジュール表記

一つのまとまったアルゴリズムの記述は、3.2節の議論に従う。互いに関連したモジュール群の列挙からなる。モジュールの基本構造は、以下のように定義される。

```
begin モジュール名 引数並び
    変数宣言
    実行文
end
```

このとき引数並びは、モジュールが手続き定義の場合、

モジュール名 変数,..., 変数

であり、モジュール名 の次に空白を置いて変数を並べる。モジュールが関数定義である場合は、

モジュール名 (変数,..., 変数)

と表す。これらは呼びだし側の形に合わせてある。また、関数の評価は必ず return 文の実行をもって終了する必要がある、これが関数値を返す。

モジュール名を持たない特別なモジュールを必ず1つ含むようにする。これはトップ・モジュールと解釈される。

4.5 注釈

可読性の向上のために注釈を丁寧に入れることが推奨されるので、ADF もそのための表記を提供する。注釈は、{...} の形式とし、... 部分には { および } を含まない任意の文字列を記述する。

4.6 変数の宣言

各モジュールの最初で、そのモジュール内における局所変数を以下のような形式で列挙する。

var 変数名,..., 変数名

変数のスコープの概念により、ここに列挙されていない変数は、このモジュールを呼び出した側で宣言されているものが継承される。呼び出した側と同名変数はローカル変数が優先される。ローカル変数は当該モジュールの実行開始時に生成され、実行終了時に消滅する。

ADF では変数宣言の際にデータ型の宣言は省かせるが、ADF による学習過程において、データには少なくとも「数値」、「論理値」、「文字列」の異なった3種類の区別があることは学習内容に含める必要がある。

4.7 拡張された仕様

アルゴリズムの基礎的な学習には前節までの道具立てで足りると思われるが、さらに進んだ学習内容のために、「制御構造のマクロ的表記」、「集合表現」を拡張的に定義する。なお、一般に、アルゴリズムの学習単元にファイル操作は現れないため「ファイル操作」に関する ADF の拡張は初期仕様では考慮しない。

4.7.1 制御構造のマクロ的表記

すでに4.1節で議論したように、loop の基本概念が理解された後の、より複雑なアルゴリズム記述の簡潔な表現のために、while、repeat、for の構造を導入する。

表5 拡張された制御構造

種別	表記
条件の前判定	while α do θ while #
条件の後判定	repeat θ exitif α repeat #
回数制御	for $i \leftarrow m$ to n by p do θ for #

「条件の後判定」における exitif は、PASCAL 言語における until に対応するものである。until 条件が、while 文における継続条件と違って脱出条件になっているという点に起因する混乱が授業で見受けられることを考慮して、until よりもより直接的な表現を導入した。

「回数制御」の i は繰り返しカウンタ用の制御変数、 m は初期値、 n は最終値、 p は刻み値を示す。

4.7.2 集合表現

高度なアルゴリズム記述には集合表現がしばしば用いられる。集合が実際に計算機上でどう実現されるかというプリミティブな問題とは独立に、集合そのものを記述要素に認めることは、アルゴリズム記述を非常に理解し易いものにする。

集合表現を実現するためには、変数と定数に集合の型を認める、集合の基本演算、全集合要素に対する操作、などの記述要素が必要になる。

- 集合を表す変数の宣言

set 変数の並び

- 集合を表す定数の表現

[要素,..., 要素]

- 集合の基本演算

表6に集合に関わる演算子を定義する。これらはすべて PASCAL 言語からの援用である。

- 全要素に対する操作

指定された集合の全要素に同じ操作を適用する繰り返し構造を以下の記述で表示する。

forall a **in** A **do** θ **forall**#

A は任意の集合、 a はその要素を示す。 a は

表6 集合に関わる演算子の定義

順位	演算子と項の型	結果型	意味
2	集合型 * 集合型	集合型	積集合 $s \cap t$
3	集合型 + 集合型 集合型 - 集合型	集合型 集合型	和集合 $s \cup t$ 差集合 $s - t$
4	集合型 = 集合型 集合型 <> 集合型 集合型 <= 集合型 集合型 >= 集合型 数値 in 集合型 文字列 in 集合型	論理型 論理型 論理型 論理型 論理型 論理型	相等 $s = t$ 相等 $s \neq t$ 包含関係 $s \subseteq t$ 包含関係 $s \supseteq t$ 要素関係 $a \in s$ 要素関係 $a \in s$

θ 内で値を参照される。この抽象化された構造は、1970年代に ALPHARD 言語 [5] で提案され、また PASCAL 8000 [6] に実装された。

● 要素数

記法 $|式|$ を拡張する。式の結果が集合型であるとき、 $|式|$ は集合の要素数を与えるとする。

5 ADF による記述例

図2は、四則計算の例を ADF で記述したものである。単一のモジュールから成り、入出力を伴うが繰り返しは含まない。初期導入用の単元の一つである。

```
begin
  var A,B,和,差,積,商

  input A,B

  和 <- A+B
  差 <- A-B
  積 <- A*B
  商 <- A/B

  output 和,差,積,商
end
```

図2 四則計算の記述例

図3は、階層化して複数のモジュールで表記した例である。必要な変数はすべて大域にしてあるため引数渡しに伴わない副手続きの呼び出しを行っている。平均値を求めるにはまず総和を求める必要があることがトップ・モジュールに明示されている。

{条件:データの終りは-1で示し、データ数は1個以上}

```
begin
  var 学生数,総和,平均値
```

```

$ データの入力と総和の計算 $
$ 平均値の計算 $
$ 平均値の出力 $

end

begin   データの入力と総和の計算

  var   データ

  学生数 <- 0
  総和 <- 0
  loop
    input データ
    if データ < 0
      then exitloop
    if#
    学生数 <- 学生数 + 1
    総和 <- 総和 + データ
  loop#
end

begin   平均値の計算
  平均値 <- 総和 / 学生数
end

begin   平均値の出力
  output "平均値 = "
  output 平均値
  newline
end
```

図3 平均値の計算の記述例

図4は、配列と関数定義を用いた例で、スタックの概念を説明している。トップ・モジュールは、下位モジュールの呼び出しの例示となっている。

```
begin
  var スタック [100], スタック先頭位置

  スタック先頭位置 <- 0
  $push 20$
  $push 6$
  $pop$
  if not $empty$
    then output $top$
  if#
  newline
end

begin push 値
  スタック先頭 <- スタック先頭 + 1
  スタック [スタック先頭] <- 値
end
```

```

begin pop
  スタック先頭 ← スタック先頭 -1
end

begin empty
  return(スタック先頭 = < 0)
end

begin top
  return(スタック [スタック先頭])
end

```

図4 スタック構造

図5は、上級レベルの単元内容である。4.7節で拡張した仕様を適用した、より簡潔な表現の例になっている。最短ハミルトン閉路探索アルゴリズムはグラフ探索の一例であり、グラフが頂点集合と辺集合とから定義されることから、集合型のよい使用例になっている。これに合わせて、set宣言、forall文が適用されている。またfor文の使用、再帰呼び出しの例にもなっている。

```

begin
  var 頂点, 訪問レベル [10], レベル, 隣接行列 [10,10]
  var 頂点数
  set 頂点集合

  頂点数 ← 10
  {隣接行列にはデータが読み込まれているものとする}
  forall 頂点 in 頂点集合 do
    訪問レベル [頂点] ← 0
  forall#
    レベル ← 0
    $最短ハミルトン閉路探索 (1)$
end

begin 最短ハミルトン閉路探索 (頂点)
  var 隣接頂点

  レベル ← レベル +1
  訪問レベル [頂点] ← レベル
  for 隣接頂点 ← 1 to 頂点数 do
    if (隣接行列 [頂点, 隣接頂点] <> 9999) &
      訪問レベル [隣接頂点] = 0
    then
      $最短ハミルトン閉路探索 (隣接頂点)$
    if#
  for#
    訪問レベル [頂点] ← 0
    レベル ← レベル -1
end

```

図5 最短ハミルトン閉路探索 (巡回セールスマン問題)

6 課題

- 現在の高校、大学における情報基礎教育を見直し、ADFを適用した教育の効果、適否を長期的に評価する。

- ADFの検証系を開発してこれを教育現場へ無償で提供する

BASICが広く適用されていることやPASCALがあまり普及しなかったことは、現場での利用可能性に関係している。ADFをアルゴリズム教育の教具とするために必要な検証系プログラムを作成し、実際に教育現場に導入されている機種、および基本ソフトの種類だけ提供する必要があると考えている。

- ADF利用者グループを作り、適用時のノウハウ交換や、問題点の抽出を行なっていく。

7 おわりに

導入から上級単元までのアルゴリズム教育に適用可能な、アルゴリズム記述専用の疑似コード的な記述形式ADFを提案した。ADFを高校や大学の情報基礎教育に適用することによって共通したアルゴリズム表記が可能となることを期待する。

なお、本稿で例示したアルゴリズムは、文献[7]を参考にした。

参考文献

- [1] Wirth, N.: The programming language Pascal, Acta Informatica 1, pp.35-63 (1971).
- [2] Dijkstra, E.W.: Go to statement considered harmful, CACM 11, pp.147-148 (1968).
- [3] 鳥居, 杉藤 (他): プログラム作成技術の現状に関する調査報告 [I], 電子技術総合研究所, Vol.185, pp.120-133 (1975).
- [4] Knuth, D.E., and Floyd, R.W.: Notes on avoiding 'go to' statements, Inf. Proc. Letters 1, pp.23-31 (1971).
- [5] Wulf, W.A.: ALPHARD: Toward a language to support structured programs, CMU-CS Report (1974).
- [6] T.Hikita, K.Ishihara: An extended PASCAL and its implementation using a trunk, Report of the Computer Centre, Univ. of Tokyo, pp.23-51 (1976).
- [7] 平田富夫: アルゴリズムとデータ構造, 森北出版 (1990).