可読性の高いアルゴリズムの記述法

大場克彦　　湯浅太一
京都医療技術短大　　豊橋技術科学大学　情報工学系

プログラムの可読性を高め、アルゴリズムの理解を容易にする一方法として、入出力を明確にしたブロックの階層構造で、プログラムを構成することを提案する。まず、データの流れを明確にするブロック化の基準について考察し、プログラミング言語に望まれる機能を提案する。具体的にＰＡＳＣＡＬにこれらの機能拡張を行い、本提案を具体例に適用してアルゴリズムの理解のしやすさを評価する。

An Algorithm Description Method to Increase the Readability

Katsuhiko OHBA† Taiichi YUASA*

† Kyoto College of Medical Technology
* Department of Information and Computer Sciences, Toyohashi University of Technology

As a method to increase the readability of programs and to make it easy to understand algorithms, we propose to construct a program with a hierarchy of blocks whose inputs and outputs are clearly described. First, we discuss those rules that make the data flow of a block clear, and then we propose the constructs desirable for a programing language to satisfy these rules. We extend PASCAL to realize our proposal. By applying our proposal to an example program, we show that the proposal will make it easy to understand the algorithm.

1　はじめに

プログラムのライフサイクルの中で、開発に要する労力よりも保守に要する労力の方が大きいことが指摘されている［1］。プログラムの保守を行うためには、そのアルゴリズムを理解しなければならないが、これに多くの時間を要する。保守で最終的に頼りにされるのはプログラムテキストであるが、これを解読してアルゴリズムを理解することは容易でないためである。

本稿では、プログラムの可読性を高め、アルゴリズムの理解を容易にする一方法として、入出力を明確にしたブロックの階層構造で、プログラムを構成することを提案する。プログラムが、入出力を明確にしたブロックから構成されていれば、そのアルゴリズムを理解するために役立つのは当然である。さらに、機能の抽象化と入出力データの抽象化を対応させてブロック化することにより、アルゴリズムの理解はより容易になる。上位のブロックから下位のブロックへ、順番に各ブロック内のデータの流れをたどることにより、プログラム全体のデータの流れが把握できるようになるためである。

以下では、まず、データの流れを明確にするためのブロック化の基準について考察する。次にこのブロック化の基準を適用するため、プログラミング言語にどのような機能があればよいかを提案する。具体例としてＰＡＳＣＡＬを用いる。続いてプログラムの記述と理解の手順を示し、最後に具体例の記述を行ってアルゴリズムの理解しやすさを評価する。

2　ブロック化の基準

本稿では、ブロックの 階層構造でプログラムを構成する。このようにしたとき、データの流れを明確にする方法を検討する。まず、ブロックの入力と出力を明確にしたブロック文を設け、子ブロックに関して必要なデータに関する情報は、ブロック文から得られるようにする。次に、ブロックの入力を出力に変換する過程を、そのブロック内だけで理解できるようにブロック化する。これを実現するためには、一定のブロック化の条件が必要である。このブロック化の基準を下に示す。基準１、２は、ブロックの出力、入力に関する基準である。基準３は、そのブロック内の文が使用するデータは、そのブロック内で与えるべきことを規定している。基準４は、そのブロック内の文が出力したデータは、そのブロック内で利用すべきことを規定している。この基準に従ってブロック化を行ったとき、ブロックの入力を出力に変換する過程をブロック内で理解することを可能にしたい。ところが、ブロックを使って段階的詳細化の過程を表すと、親ブロックでは抽象化されたデータを扱い、子ブロックでは具体化されたデータを扱う場合が生じる。本稿では、データの抽象化と具体化の関係を、構造を有するデータを使って表す。ブロックの入出力が構造を有するデータの場合には、ブロック内の文ではその要素を参照したり、その要素に代入する場合が生じる。この場合には入出力データに関する基準１、基準２が、このままでは成立しない。しかし、構造を有するデータの場合には、その要素の値が変われば、構造を有するデータそのものの値が新しい値になる［2］ので、出力データに関しては補足１が成立する。また、構造型の要素が参照されるということは、大きくみれば構造を有するデータそのものが参照されていることになるので、補足２が成立する。補足１、補足２を適用することにより、基準１、基準２が成立するようになる。

補足１、２は、機能の抽象化と入出力の抽象化を対応させる手段を与える。ブロック化の基準に、補足１、２を付け加えることにより、機能の抽象化と入出力データの抽象化を対応させてブロック化した場合にも、ブロックの入力を出力に変換する過程を、ブロック内で理解することが可能になる。

基準１：ブロックの出力データは、ブロック内の文の出力データの集合の中にある。

基準２：ブロックの入力データ全てがブロック内のいずれかの文の入力データとして使われている。

基準３：ブロック内の文の入力データは、次のいずれかの基準を満足している。

- そのブロックの入力データの中にある。
- 同じブロック内の文で、かつ、その文の前に実行される文の出力データの集合の中にある。

基準４：ブロック内の文の出力データは、次のいずれかの条件を満足している。

- そのブロックの出力データになっている。
- 同じブロック内の文で、かつ、その文の後に実行される文の入力データの集合の中にある。

補足１：ブロックの出力データが構造型の場合、ブロック内の文がその構成要素に代入をしていればその構造を有するデータは出力となる。この場合は、値がブロック内で更新されるので、入力側にも同じデータ名を書く。

補足２：ブロックの入力データが構造型の場合、ブロック内の文がその構成要素を使用していればその構造を有するデータは使用されている。

3　記述法

前節に示したブロック化の基準を適用しながらプログラミングを行うには、プログラミング言語にどのような機能が必要か検討する。具体例としてＰＡＳＣＡＬを用いる。

プログラムのアルゴリズムを理解するためには、プログラムのデータの流れを理解する必要がある。データの流れを理解するためには、プログラム本体や手続きを、さらに小さな構成要素（ブロック）に分割して考えると理解しやすくなる。ブロックの入力と出力が明確にされていて、ブロックの入力から出力に変換される過程がブロック内だけで理解できれば、それだけでデータの流れは理解しやすくなる。さらに、ブロック間のデータの流れが明確にされていれば、全体のデータの流れは、より理解しやすくなる。従って、プログラミング言語には、データの流れを明確にする機能が望まれる。

本稿では、上に述べた機能を実現するため、まず、入力と出力を明記したブロックという、手続きより小さいプログラムの構成要を設ける。次に、プログラム本体や手続きをブロックの階層構造で表すように構文を定める。ところが、プログラム本体や手続きをブロックの階層構造で表すと、ブロックが入れ子になる。ブロックを入れ子にすると、現在考察の対象となっている文と文の間に、直接関係のない文が入るので、データの流れが把握しにくくなる。そこで、ブロック内では、子ブロックをブロック文を用いて表し、子ブロックの内容の記述は、別のブロックで行うように構文を定める。階層構造内のブロックの位置を明確に示すため、ブロックに、ブロック番号をつける。

ブロックの入力を出力に変換する過程が、ブロック内だけで理解できるように記述してあることを保証するため、ブロック化の基準を用いて、記述された内容を検査する機能を付け加える。

ブロックの中で、データの流れが明確でなくなるのは、手続き呼び出しのときである。手続き呼び出しが現れたときにも、データの流れが明確になるように機能を拡張する。具体的には、手続き呼び出し文と手続き頭部で入力データと出力データを区別して記述するように構文を改める。また、ブロックと同じ理由で、手続き宣言も入れ子にしない。

以上の検討結果をもとにＰＡＳＣＡＬの仕様を拡張した部分を下に示す。なお、名前は日本語（母国語）が使えるものとする。構文は拡張バッカス記法で示す。ここで｛α｝はαをを０個以上並べたものを示し、｛α｝'はαを０個か１個並べたものを示す。

①＜手続き宣言＞：：＝ｐｒｏｃｅｄｕｒｅ
　　＜主ブロック＞｛；＜ブロック＞｝　ｅｎｄ－ｐｒｏｃｅｄｕｒｅ
②＜主ブロック＞：：＝＜手続き頭部＞＜定義・宣言部＞＜複合文＞
③＜手続き頭部＞：：＝＜名前＞（＜仮パラメタ並び＞）－＞（＜仮パラメタ並び＞）
④＜仮パラメタ並び＞：：＝｛＜仮パラメタ＞｝'｛、＜仮パラメタ＞｝
⑤＜ブロック＞：：＝＜ブロック番号＞＜ブロック頭部＞＜定義・宣言部＞＜複合文＞
⑥＜ブロック番号＞：：＝＃＜番号＞
⑦＜番号＞：：＝＜数字列＞｛.＜数字列＞｝
⑧＜ブロック頭部＞：：＝＜名前＞（＜変数並び＞）－＞（＜変数並び＞）
⑨＜変数並び＞：：＝｛＜変数＞｝'｛、＜変数＞｝
⑩＜ブロック文＞：：＝＃＜名前＞（＜変数並び＞）－＞（＜変数並び＞）
⑪＜手続き呼び出し文＞：：＝＜名前＞（＜実パラメタ並び＞）－＞（＜実パラメタ並び＞）
⑫＜実パラメタ並び＞：：＝｛＜実パラメタ＞｝'｛、＜実パラメタ＞｝
⑬＜定義・宣言部＞：：＝
　｛＜ラベル宣言部＞｝｛＜定数定義部＞｝｛＜型定義部＞｝｛＜変数宣言部＞｝

4　プログラムの記述と理解の手順

4．１　記述の手順

本記述法では、構造化プログラミング［３］の段階的詳細化法［４］における、１つの詳細化の過程を１つのブロックで表す。まず、与えられた仕様からプログラムの入力と出力を明確にし、これをプログラム頭部に書く。次に仕様の条件を整理し、入力から出力を作り出す手順を大まかに記述する。必要ならば中間的なデータを導入する。これが主ブロックとなる。次にブロック内のブロック文を詳細化する。ブロック頭部はブロック文をコピーする。また、ブロック内で局所的に使用するデータは、そのブロック内で宣言する。ブロックで宣言したデータの有効範囲はそのブロックを頂点とする部分木である。詳細化されていないブロック文がなくなるまで、詳細化を繰り返す。詳細化をやめる条件は、ブロック文が代入文か手続き呼び出し文に置き換えられるときである。詳細化の各段階で、データの流れがブロック化の基準を満足していることを確かめる。段階的詳細化の結果、プログラムはブロックの木で表さ

れる。木の中に同じ機能を表す部分木があればこれを手続きにする。また、木が大きくなりすぎるときには、まとまった機能を表す部分木を手続きに置き換える。以上に述べた記述方法は手続きや関数にも適用できる。

4．2　理解の手順

本手法によるプログラムの理解の手順について検討する。本手法により記述されたプログラムは、ブロックの木を構成している。親のブロックの中では、子ブロックをブロック文として抽象化している。ブロックのデータの流れを考えるときは、子ブロックの詳細なデータの流れを知る必要はなく、ブロック文の入出力だけを考えればよい。これはブロック化の基準により、ブロック文の入出力として記述されたデータ以外は、子ブロックとの間にデータの受渡しがないことが保証されているためである。従って、主ブロックから順番に個々のブロックのデータの流れをたどることにより、プログラムを理解することができる。

5　評価

具体的な問題を記述しアルゴリズムの理解しやすさを評価する。例として、8クイーンの1つの解を求める問題の中で、再帰的に呼び出される手続きを記述する。アルゴリズムは、文献［4］［3］を参考にした。この手続きは、盤上の各々の升目に女王を置けるかどうかという情報（『女王』）と、女王をどの行から置き始めるかという情報（『行』）が与えられ、全ての行に女王が置かれたどうかという情報（『終了』）と盤の各行に置かれた女王の列番号を配列（『解』）に入れて返すもの（図3）である。出力側だけでなく入力側にも『解』があるのは、『解』の値が手続きの中で更新されるためである。『女王』が出力側にもあるのは、手続きの中で『女王』の値が変わるためである。手続きの中で使用される型の型定義と関数の関数宣言を、図1、図2に示す。

```
type 状態＝record
                列上に無い：array［1..8］of Boolean；
                上対角に無い：array［2..16］of Boolean；
                下対角に無い：array［－7..7］of Boolean；
            end；
     番号＝0..8；
```

図1　型定義

```
function 女王を置ける（女王：状態；行、列：番号）：Boolean；
  begin
    女王を置ける：＝女王.列上に無い［列］ and 女王.上対角に無い［行＋列］
                                  and 女王.下対角に無い［行－列］
  end；
end_function；
function 最後の女王（行：番号）：Boolean；
  begin 最後の女王：＝（行＝8） end
end_function；
function 置く場所がない（列：番号）：Boolean；
  begin 置く場所がない：＝（列＝8） end
end_function
```

図2　関数宣言

主ブロックは、手続きの入力から出力が作り出される過程を大まかに記述している。機能だけでなくデータも大まかな記述になっている。主ブロックから手続きの骨格がわかる。『終了』あるいは「置く場所がない」がｔｒｕｅになるまで、列を1つずらし女王を置けるか調べ、女王が置ける場合には、女王を置ける場合の処理をするという操作を、繰り返している。手続きの入力をどこで参照しているか、入力をどこで更新しているか、出力をどこで作りだしているか、中間的データをどこで導入しどこで参照しているかなどが、簡単にわかる。

```
procedure 女王を置く（var 女王：状態；var 解：女王位置；行：番号）
  ->（var 終了：Boolean；var 解：女王位置；var 女王：状態）；
 var 列：番号；
 begin
   列：=0； 終了：=false；
   repeat
     #列を1つずらす（列）->（列）
     if 女王を置ける（女王、行、列） then
       #女王を置ける場合の処理（女王、解、行、列）->（終了、解、女王）
   until 終了 or 置く場所がない（列）
 end；
 #1 列を1つずらす（列）->（列）；
   begin 列：=列+1 end；
 #2 女王を置ける場合の処理（女王、解、行、列）->（終了、解、女王）；
   begin
     #女王を1つ置く（女王、解、行、列）->（解、女王）；
     if 最後の女王（行） then 終了：=true
     else begin
       #残りの女王を置く（女王、解、行）->（終了、解、女王）；
       if not 終了 then
         #置いた女王を取り除く（女王、解、行、列）->（女王、解）；
     end
   end；
 #2．1 女王を1つ置く（女王、解、行、列）->（解、女王）；
   begin 解［行］：=列；
         女王.列上に無い［行］：=false；
         女王.上対角に無い［行+列］：=false；
         女王.下対角に無い［行−列］：=false
   end；
 #2．2 残りの女王を置く（女王、解、行）->（終了、解、女王）；
   begin 女王を置く（女王、行+1）->（終了、解、女王） end；
 #2．3 置いた女王を取り除く（女王、解、行、列）->（女王、解）；
   begin 女王.列上に無い［行］：=true；
         女王.上対角に無い［行+列］：=true；
         女王.下対角に無い［行−列］：=true；
         解［行］：=0
   end；
end_procedure；
```

図3 記述例

主ブロックの中のブロック文を詳細化したのが#1と#2である。さらに、#2の中のブロック文を、#2．1、#2．2、#2．3で詳細化している。#2．1と#2．3は、入力データを更新して出力データを作りだしている。どのように更新しているかをブロック内で記述しているが、機能の詳細化に対応して扱うデータも詳細化されている。なお、代入文で構造を有するデータの要素に代入を行う場合には、構造型データの値が更新されるので、基準3により構造型データも入力として考える必要がある。

上の例では、主ブロックから順番にブロックの階層関係をたどって、個々のブロックのデータの流れを知ることにより、プログラム全体のデータの流れを知ることができる。これは、段階的詳細化の過程を入出力を明確にしたブロックの階層構造で表し、個々のブロックを、ブロック文で抽象化しているためである。また、主ブロック以外のブロックの、データの流れを把握することも簡単である。これは、ブロックの入出力が明記されていること、ブロック内の代入参照関係が明確にされていること、ブロック文による抽象化などの効果によるものである。さらに、データの詳細化と機能の詳細化を対応させて記述することができるので、なぜそのように詳細化するのかということが理解できる。これらの特長が

アルゴリズムを理解しやすくしている。

6　他の研究との関係

入出力を明確にしたブロックの階層構造でアルゴリズムを表現し、ブロック間のデータの流れに一定の制約を設けることは、１９７０年代にＨＯＳ［５］で行っている。ＨＯＳはブロック間のデータの受渡し間違いをなくすことにより誤りのないプログラムを作ろうとしている。そのため構造化プログラミングの制御構造を使用することを禁止しているので、記述が冗長になる。ＨＯＳの記述の冗長さを改善するために、プログラム言語の制御構造を使用することを許し、ブロック間のデータの流れを規制する新たな制約条件を設ける試み［６］がある。これらの研究は、プログラムの誤りをなくすことを目的にしている点が本研究と異なる。本研究では、アルゴリズムの可読性を高めるという観点から［６］の制約条件に検討を加え、ブロックの入出力が構造を有するデータの場合についても適用できるブロック化の基準にまとめた。また、この基準を有効に適用するために、プログラミング言語の機能を拡張すべき点を明確にしていることも［５］［６］と違う点である。

7　おわりに

プログラムの可読性を高め、アルゴリズムの理解を容易にするために、入出力を明確にしたブロックの階層構造でプログラムを構成することを提案した。まず、データの流れを明確にするブロック化の基準について考察し、プログラミング言語に望まれる機能を提案した。具体的にＰＡＳＣＡＬにこれらの機能拡張を行い、具体的な例に適用した。

本記述法によれば、主ブロックから順番にブロックの階層関係をたどって、個々のブロックのデータの流れを知ることにより、プログラム全体のデータの流れを知ることができる。また、個々のブロックにおいてもデータの流れを簡単に把握できる。さらに、データの詳細化と機能の詳細化を対応させて記述することができるので、なぜそのように詳細化するのかということが理解できる。これらの特長により、アルゴリズムの理解が容易になることを示した。

謝辞　本研究を始める機会を作って下さった豊橋技術科学大学の北川孟教授に感謝します。また、熱心に討論に参加し貴重なご意見を頂いた豊橋技術科学大学湯浅研究室の貴島寿郎氏及び同研究室の学生諸氏に感謝します。

参考文献

［１］情報処理学会編：情報処理ハンドブック，ｐ．１９５６，オーム社，東京（１９８９）
［２］Ｈｏａｒｅ，Ｃ．Ａ．Ｒ．（川合慧訳）：データ構造化序論，構造化プログラミング，ｐ．２４９，サイエンス社，東京（１９７５）
［３］Ｄｉｊｋｓｔｒａ，Ｅ．Ｗ．（野下浩平訳）：構造化プログラミング論，構造化プログラミング，ｐ．２４９，サイエンス社，東京（１９７５）
［４］Ｗｉｒｔｈ，Ｎ：Ｐｒｏｇｒａｍ Ｄｅｖｅｌｏｐｍｅｎｔ ｂｙ Ｓｔｅｐｗｉｓｅ Ｒｅｆｉｎｅｍｅｎｔ，ＣＡＣＭ，Ｖｏｌ．１４，Ｎｏ．４，ｐｐ．２２１－２２７（１９７１）
［５］Ｍａｒｇａｒｅｔ，Ｈ．ａｎｄ Ｚｅｌｄｉｎ，Ｓ．：Ｈｉｇｈｅｒ Ｏｒｄｅｒ Ｓｏｆｔｗａｒｅ－Ａ Ｍｅｔｈｄｏｌｏｇｙ ｆｏｒ Ｄｅｆｉｎｉｎｇ Ｓｏｆｔｗａｒｅ，ＩＥＥＥ ＴＲＡＮＳＡＣＴＩＯＮ ＯＮ ＳＯＦＴＷＡＲＥ ＥＮＧＩＮＥＥＲＩＮＧ，Ｖｏｌ．ＳＥ－２，Ｎｏ．１，ｐｐ．９－３２（１９７６）
［６］大場克彦，金戸孝夫：プログラム論理図の形式的表現，ソフトウエア工学研究会８０－２０，ｐｐ．１５１－１５８（１９９１）