ソフトウェア工学 22－11
（1982. 2. 10）

プログラミング言語の改良

－プログラムの文書化の立場から－

An Improvement of the Programming Language from a Point of View of Better Documentation.

高嶋 孝明　　大岩 元
Takaaki TAKASHIMA　　Hajime OHIWA

（豊橋技術科学大学）

1．まえがき

プログラミング言語は、人間が計算機にやってもらいたいことを正確に表現するための道具である。それは、機械語の時代から始まって常に Man-Machineインタフエースに主眼をおいて開発・研究が行われてきた。すなわち、人間にとっていかに楽に、かつ厳密に仕事の手順を指示できるか、ということが最大の関心事であった。

しかし、プログラムが巨大化・複雑化している現代においては、唯一の信頼できる文書（ドキュメント）としての役割が重要になりつつある。本論文では、この観点からソースプログラムの文書としての役割を強化させるために、プログラミング言語に冗長性を持たせることを提案して「Redundant Control Facility」と名づけ、Ratforに対する実施例を報告する。

2．本質的処理と非本質的処理

プログラムをトップダウンに作成する場合、まず最初は作りたいプログラムを最も一般的な形で表現し、次にその各構成要素を動作の記述に段々に展開していく。この詳細化の初期段階においては、プログラムはアルゴリズムの構成上不可欠な要素のみから成っている。ところが、ある段階を越えると、そのプログラムが実際に働くために必要な処理が付け加えられる。このアルゴリズム構成上不可欠な要素を「本質的処理（Essential）」、アルゴリズム実現のために付加された処理を「非本質的処理（Non-Essential）」と呼ぶことにする。同様の概念を Hanata らは「Essential」「Incidental」と呼んでいる[1]。

例えば、配列の添字が範囲を越えた時の処理、0で割らないかどうかのチエックなどは、初期の概念設計のレベルでは考慮されない。アルゴリズムは、これらの起らない理想の環境下を想定している。しかし、詳細化が進めば現実の問題として、この環境からはずれた際の処理や、環境を作り出すための前処理や後処理が必要となる。これらが、我々のいう非本質的処理である。（図／参照）　また、この処理が実質的な意味でプログラムの信頼性の鍵を握っている、と言っても過言ではない。

エラー処理も非本質的処理の一部であるが、それが全てではない。例えば、予期しないところでの EOFの読込みは明らかにエラー処理である。しかし、その時点で EOFをも期待している入力に対する EOFの処理をはたしてエラー処理と呼べるであろうか。また、テキストフアイルの処理において、行構造を意識せずに単に文字の集まりとして扱う局面で、プログラミング言語が行単位の入力しか許していなくて、一行分の処理を終えた後に必要となる次の行の読込み処理をエラー処理と呼ぶのは不適当である。しかし、それらは決して本質的処理ではない。これらの処理、すなわち非本質的処理は、プログラム設計の初期には考慮されないし、また考慮されていたとしても、考えずに済むのならそうしておきたい性格のものである。しかしそれは、最終的なプログラムでは非常に重要な役割を果すものである。本質的処理と非本質的処理が組合さって、プログラムは初めて完全なものとなる。

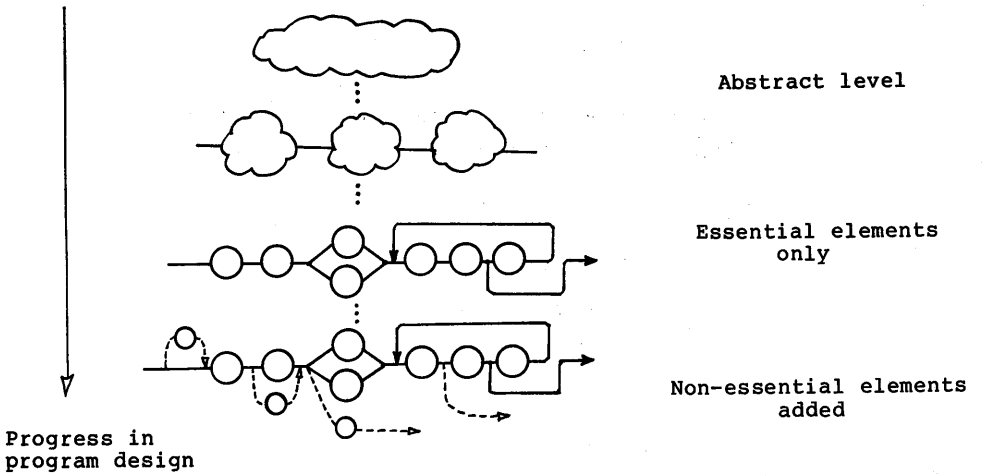

図 1　プログラムの段階的詳細化の過程

3．コミュニケーションのためのプログラミング言語

ここでは、これまでの話を逆にして、すでに書かれたプログラムを理解することを考える。プログラムを理解することが、単にそのプログラムの命令によって計算機がどのように働いているかを知ることであるのならば、詩を意味を考えずに朗読するのと同じである。それを「理解した」とは言わない。アルゴリズムを、その目的という観点から整理して意味をつかみとることが、プログラムの理解である。そのためには、まず第一にプログラムから非本質的処理をはぎ取って本質的処理を明確にさせることが必要である。そうして、アルゴリズムの全体像をつかみ、その意味を理解した後に非本質的処理の意味を一つずつ理解する。これではじめてプログラムを理解したことになる。言い換えれば、プログラマがたどった詳細化の道すじを再びたどることが理解であり、それができなければならない。

プログラミング言語が人間から計算機への一方通行のコミュニケーションの道具であれば、このことは必ずしも要求されない。しかし、人間と人間のコミュニケーションの道具としてプログラミング言語を考えるとき、これは非常に重要な要素である。

4．本質的処理と非本質処理の書き分け

これまでに述べてきたことから、プログラムを理解しやすくする　ーすなわち文書としての役割を向上させるー　ためにまず最初に要求されるのは、非本質的処理のはぎ取りが容易に行えることである、という点に気付いた。本質的処理と非本質的処理がソースプログラム上で明確に書き分けてあれば、その理解は容易になる。しかし、従来のプログラミング言語では、注釈に頼らずにこの書き分けを行うことはほとんど不可能である。

書き分けの難しさは、これまでにもしばしば指摘されている。例えば Kernighanらは

> 「誤りの処理は読みやすさとは矛盾する。だが、この処理をなしですますことはできない。最良の言語を使ってさえ、ものごとの主要な流れをわかりにくくする。というのは、誤りの処理そのものがプログラム本来の仕事をあらわす構造とは別の構造をもち込むからである。」

と述べている[2]。

上記の引用で、「誤りの処理」を「非本質的処理」と置き換えたものが、我々が主張しようとするところである。

それでは、この難しさの原因はどこにあるのだろうか。 Kernighanらのいう構造とは、プログラムの制御構造ではなく、論理構造を意味している。本来の仕事（本質的処理）に用いられている制御構造は、誤りの処理（非本質的処理）に制御を移すためにも用いられる。従来のプログラミング言語はこれらの区別の行っていない、というのは、計算機にとってはどちらも、なすべき仕事は同一だからである。したがって、お互い、全く異なった意図をもって使用されるにもかかわらず、同じ表現で書かざるを得ない。

これらのことから、プログラムを読みにくくする原因の一つは、コーデイングの段階で本質的処理と非本質的処理が混じりあってしまい、あとから分離できなくなることにある、との結論に達した。すぐれたプログラマは、はじめからそのことをよく考えて設計する。また HCPチャートを用いたプログラム設計でもこれらを書き分けていく[1]。ところが、いざコーデイングとなると、それらに対して一つの表記法しか用意されていない。また、異なった意図をもって使用されるべきものにただ一つの表記法しかないことは、本来、非本質的処理として明確に書き分けるものを、無意識のうちに、プログラムのそこら中に無造作にほうり込ませてしまい、読みやすさや信頼性を著しく低下させる結果となる。（図2 参照）

5．本質的処理と非本質的処理の分離書法

以上の考察から、本質的処理と非本質的処理を書き分けるためには、プログラムの流れの中でこれら二つの接点となる分岐型の制御構造に、二つの異なった表記法を持たせればよいことが明らかになる。また同様のことが、繰返し処理から抜け出る制御構造に対しても、正常出口、異常出口の形で言える。

すなわち、「プログラミング言語に冗長性を持たせ、同一の制御構造でありながら使用する意図の異なるものには、異なった表記法を設ける。」というのが我々の提案であり、これを「Redundant Control Facility」と名づける。

Ratforでの実施例

我々はこの観点から Ratfor に対して言語改良を行った[3]。 Ratfor を用いた理由は、処理系が小規模で容易に手が加えられ、また実質的な意味で我々が最も多用しているプログラミング言語だからである。

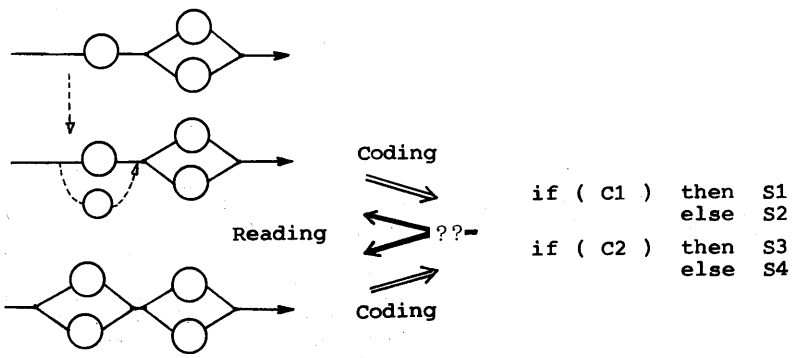

図2　表記法が一つしかないことの問題点

I. 分岐制御

本質的処理としての分岐には case 文を導入した。図3に構文グラフを示す。 Ratfor には分岐制御として if··else if ··else があある。つまり2分岐だけが許されている。しかし、我々は「本質的処理における分岐は多分岐が基本である」と考えるため、多分岐の扱える case 文を導入したのである。これは、case のラベルに条件式を書くことから Pascal などのcase より一般的になっている。

また、並列処理を考慮し、各条件式は互いに排他的、すなわち評価の順によらないものとする。 (実際には、Fortran に変換されるために並列処理を考えることは意味がないが、将来のことを考えて。) 条件式の順が問題となる時は、

```
case  (C1) S1  (C2) S2  ···· else S
```

のように else を用いる。もちろん else case の連鎖を用いることも可能である。従来の if 文は、この case 文の条件式が一つだけの特殊な場合である。基本的にはこの場合も case を用いることにしているが、 if の使用を否定するものではない。また case における else の省略は、 if における省略と同様に扱う。

非本質的処理への分岐には if を when と置き換えて使う。構文は if と同一である。また case 文の else節として非本質的処理の来る場合があるため、この時は else を otherwiseに置き換えて使う。

II. 繰返し処理からの抜け出し

従来の Ratfor の break文は、break の意味合いが「制御をこわす」という方が強いため、これを非本質的処理に用い、その代りに exit を本質的処理の出口に用いる。

図4に、これらの書き分けの例を示す。

以上の我々の定義からすれば、

```
case  ( c )  break
when  ( c )  exit
```

などといった使い方は誤りとなる。

これらの言語改良は、Ratforの処理系のマクロ置換と構文解析部にわずかに手を加えることで、容易に実現できる。

すなわち、exit、otherwise、when はマクロ置換で break、else、if に置換え、また case は if に置換えた後 case が閉じない間に現れた (で始まる文を else if (に置換えればよい。その後は、従来の処理を施して Fortran に変換する。

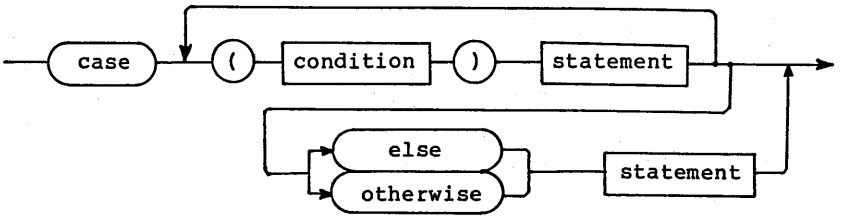

図 3　CASE文の構文グラフ

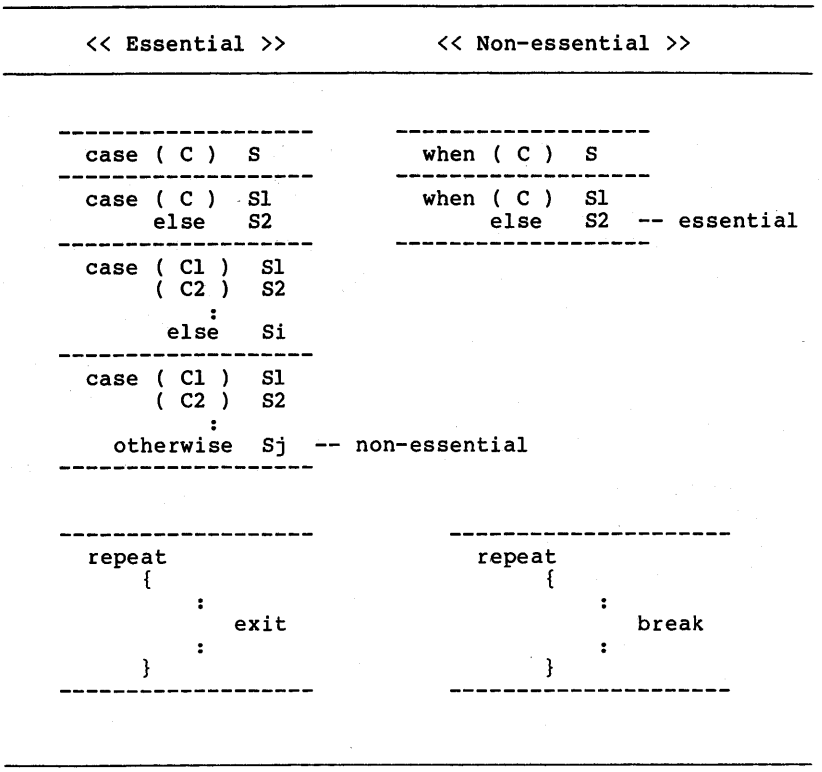

図 4　本質的処理と非本質的処理の書き分け

6. Ada における例外処理との比較

プログラミング言語 Adaでは、エラー処理や例外的な条件に対する処理が定められている[4]。本節では、我々の提案と Adaにおける例外処理とを比較する。

Ada には、プログラム単位の最後に例外処理をまとめて記述する・言語系が定義する例外がある・例外を発生し、それを伝幡できる、などの機能がある。例外処理を本体とは分離して書くことは、我々の提案する本質的処理と非本質的処理との書き分けと同様の効果をもつ。また、言語が定義する規定の例外は、モニタによって自動的に調べられて制御が例外処理に移され、その伝播も行えることは、我々のものよりすぐれている。

しかし一方、利用者が定義した例外は、その処理内容は本体から離して記述されるが、例外のチエックや例外の発生はプログラマ自身が記述して、プログラム本体にうめ込まなければならない。これが本質的処理と混在してしまえば、たとえ例外処理本体が分離されていたとしても、プログラムは読みにくいものになる。我々の提案は、例外を発生するか否かの判断・分岐の段階から書き分けを行っていこうとするものであり、その点では、Ada よりもすぐれている。

また Adaでは、例外が発生すると、プログラム単位の残りの部分が例外処理に置き換ってしまう。例外処理をエラー処理と考えればこれも適当であるが、我々はエラー処理だけが非本質的処理とは考えていない。したがって、この処理の置き換えはかえって不便なものとなる。

7. おわりに

プログラムの段階的詳細化の過程を分析することから「本質的処理（Essential）」と「非本質的処理（Non-Essential）」の考えを導入した。そして、これら二つの処理がコーディングの段階で混在してしまうことがプログラムを読みにくくすることを明らかにし、これらを書き分ける書法としてプログラミング言語に冗長性を取り入れることを提案し「Redundant Control Facility」と名づけた。

これに対して、「本質的処理と非本質的処理の区別をどこでするのか？」、「はたして、そのような書き分けが本当にできるのか？」といった疑問がでてくるかもしれない。我々は、これは作成者が主観的に決めればよい問題であると考えている。また、このような冗長性を取入れたことで、作成時にどちらで書くかを迷うようであれば、それは作成者に反省をうながす作用をもつものであり、より完成度の高いプログラムを生むことになるであろう。

我々は、プログラムづらを良くし、文書としての機能を向上させようとしているが、これは比較的少数意見のようである[2]。ここで提案したプログラミング言語の改良はちっぽけなものであるが、そこから得られる効果は非常に大きなものが期待される。ソフトウエアの品質管理が問題となりつつある現在、文書化の立場から改めてプログラミング言語を見直せば、さらに多くの問題が浮きぼりにされ、今後の言語設計や改良に対して、何らかの示唆を与えるものと思われる。

参考文献

[1] S.Hanata, T.Sato, M.Inada : Documentation Technology for Packing Hierachical Function, Data, and Control Structures. IEEE COMPCON Fall '81 (1981), PP.284-290.

[2] B.W.Kernighan, P.J.Plauger, 木村訳：ソフトウエア作法 .共立出版 (1981) PP.78-79, P485.

[3] 高嶋, 大岩：ドキュメント性能を向上させたRatforについて, 情報処理学会 第23回全国大会 3H-8 (1981)

[4] US Department of Defence : Reference Manual for the Ada Programming Language. Springer-Verlag, New York, (1980).