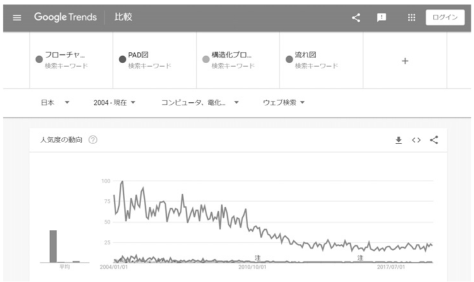
構造化プログラミングとPAD図
プログラム言語論

櫻井彰人

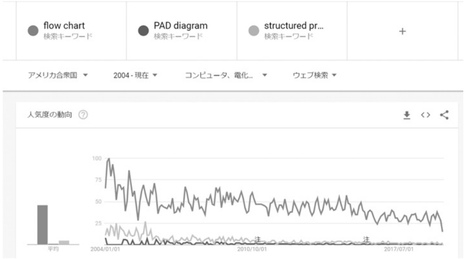
背景

- もうフローチャートを描く時代ではないと思っていたのですが。



背景

- プログラム作成のためとは限らないので注意は必要



構造化プログラミング

プログラム言語を設計する時には、プログラムがどのような計算を行っているかが容易にわかるような言語仕様とするべく努力する必要がある。

> 構造化プログラミング：プログラムをテキストとしてみたとき、その構造はプログラムの動作を理解する手助けとなるべきである。

勿論、構造化したプログラムも、しっかり考えて作れば、構造化されていないプログラムと同等に効率的（速い、メモリ量少ない）であるべきである

E. Dijkstra, Go to statement considered harmful, *Communication of the ACM*, 11(3), 147-148(1968).

Edgar Dijkstra: Go To Statement Considered Harmful

Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing
CR Categories: 4.22, 5.23, 5.24

Editor:

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of **go to** statements in the programs they produce. More recently I discovered why the use of the **go to** statement has such disastrous effects, and I became convinced that the **go to** statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded as the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of **go to** statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the text a "textual index."

When we include conditional clauses (**if** B **then** A), alternative clauses (**if** B **then** $A1$ **else** $A2$), choice clauses as introduced by C. A. R. Hoare (case[i] of($A1, A2, \cdots, An$)), or conditional expressions as introduced by J. McCarthy ($B1 \rightarrow E1, B2 \rightarrow E2, \cdots, Bn \rightarrow En$), the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, **while** B **repeat** A or **repeat** A **until** B). Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of n, its value equals the number of people in the room minus one!

The unbridled use of the **go to** statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the **go to** statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The **go to** statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the **go to** statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the **go to** statement is far from new. I remember having read the explicit recommendation to restrict the use of the **go to** statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than **go to** statements and switches, and it eliminates the need for introducing a large number of labels in the program."

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the **go to** statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

References:
1. Wirth, Niklaus, and Hoare, C. A. R. A contribution to the development of ALGOL. *Comm. ACM 9* (June 1966), 413–432.
2. Böhm, Corrado, and Jacopini, Guiseppe. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. ACM 9* (May 1966), 366–371.

Edsger W. Dijkstra
Technological University
Eindhoven, The Netherlands

Volume 11 / Number 3 / March, 1968 — Communications of the ACM 147

Edgar Dijkstra: Go To Statement Considered Harmful 1

Aus: *Communications of the ACM* 11, 3 (March 1968). 147-148.

Edgar Dijkstra: Go To Statement Considered Harmful

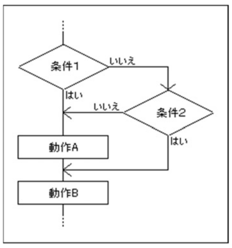
https://homepages.cwi.nl/~storm/teaching/reader/Dijkstra68.pdf

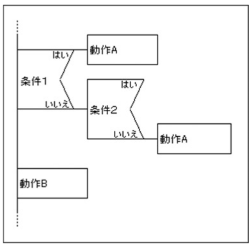
---

流れより構造

- プログラム中の制御の流れを示すのに、流れ図（フローチャート）を使うことが多い。
- しかし、これは、「構造化プログラミング」の精神に反する
- 1980年代に、「構造化プログラミング」に適した、フローチャートに代わる、図示方法が検討された
- PAD図はその中の代表的なものである。
- PAD = Problem Analysis Diagram であるが、あまり気にしなくてよい
- 発案者は、（当時）日立製作所中央研究所の二村良彦

比較





http://www.happy2-island.com/beginner/invitation02/capter99900.shtml

---

プログラムの構成と構成単位

- 基本的な文 – 命令型言語では、代入文

- 連接 – 文を順につなげる
- 分岐 – ある条件が成立するときのみ実行される文を作る
- くり返し – 同じ文を繰り返させる

連接ー順番に実行する

命令型言語においては、順番に実行すべき文は、（一般に、左から右に、上から下に）順に並べて記す。Fortran では上から下に、Algol 60 では左から右、上から下に、セミコロンで区切って並べる

（例） Algol60では
y :=2* x + 1; z := 3*y + 2;
x := y + z;

複合文, ブロックともいう

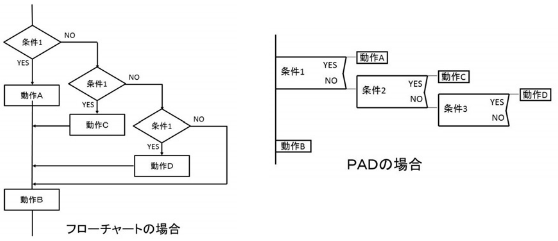
- Algol60では, 文の連鎖を begin と end で区切ると複合文（ブロック）となる
- (注) 厳密には宣言文があるブロックとないブロックは分けて考えるが、本スライドでは区別しないことにする
- (例) **begin** w := x; x := y; y := w **end**

- Algol60 では、文が現れるべきところには、どこでも、複合文（ブロック）が現れて良い。複合文（ブロック） は、1入力1出力である
- (注) 実行の効率性のため（無駄な実行を行わないため）、ブロックを途中脱出する方法（goto文以外）が、現在の言語では提供されている
- (注) 例外という、実行が中断される全く別の枠組みもある

条件文

- 条件「文」というより条件「構造」の方が正しかろう
  - 文と式から「条件文」を作り上げているから
- ある条件が成立したときに限りある文を実行する。従って、 if … then … が基本。
  - 勿論、これだけでは使いにくいので、if … then … else … が使われる。
  - さらに、if 構造の末尾を明確にするために、現在では、if … then … else … end とすることが多い

条件文



フローチャートの場合

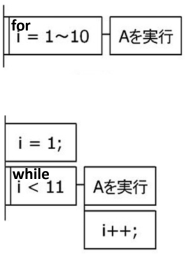
PADの場合

http://www.ced.is.utsunomiya-u.ac.jp/lecture/2013/prog/p1/kadai4/kadai4_2.html

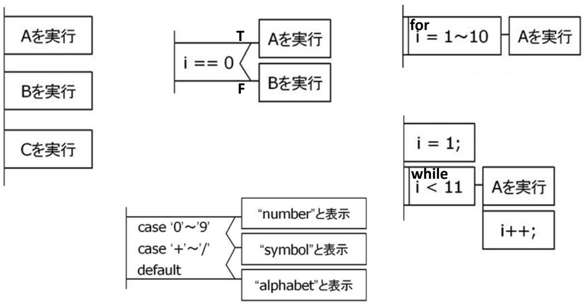
くり返し

- くり返しは、(命令型言語の場合)繰り返す文と繰り返しを継続するか否かを判断する部分からなっている
  - コンピュータのプログラムと日常生活のプログラム(繰り返しがない)との大きな乖離点である
- 機械語には、対応する(原始的な)機能はない
- Fortranでは、DOループという形で for ループが導入された
- Algol60では、while … do … が導入された

くり返し



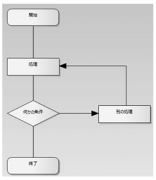
まとめ: PAD図の基本

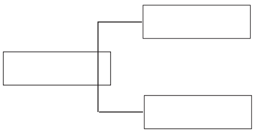


http://web.cc.yamaguchi-u.ac.jp/~fukuyo/prog2/302.html

Break と n+1/2

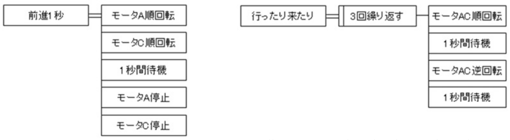
- 配列中にある要素を検索するプログラムでは、発見したら、くり返しを終了したい
  - while の条件中に「未発見」という条件を入れる
  - 発見したら、ループを終了する → break
- n+1/2 と呼ばれるループがある





PAD図と設計

- PAD図は(流れ図と同様に)下流工程(プログラミング)の設計に向いている
  - 構造化プログラミングそのものであるため、流れ図より遥かによい。
  - 教育にもよい



http://itpro.nikkeibp.co.jp/article/MAG/20100517/348034/?ST=develop&P=2

- 上流設計では、UMLに準拠したアクティビティ図かステートチャートあたりがよい