Jour. Ind. Inst. Sc. 61 (A), May 1979, pp. 123-133
© Printed in India

Development of a structured program for conversion to prenex normal form

N. CHAKRAPANI*, S. V. RANGASWAMY* AND V. G. TIKEKAR**
Indian Institute of Science, Bangalore 560 012

Received on February 9, 1979; Revised on May 24, 1979

Abstract

The method of structured programming or program development using top-down, stepwise refinement technique, provides a systematic approach for the development of programs of considerable complexity. The aim of this paper is to present the philosophy of structured programming through a case study of a non-numeric programming task. The problem of converting a well-formed formula in first-order logic into prenex normal form is considered. The program has been coded in the programming language Pascal and implemented on a Dec-10 system. The program has about 500 lines of code and comprises 11 procedures.

Key words : Structured programming, non-numeric computation, stepwise refinement, Pascal.

1. Introduction

Structured programming or program development using top-down stepwise refinement is gaining importance in the programming profession, as it provides a systematic approach to the development of fairly complex programs in an elegant and efficient manner. Problems currently considered for computerisation are fairly complex and many computer implementations require modifications due to improvements conceived in the original problem statement. This requires that programs be developed in a hierarchical manner, so that the changes conceived in the original problem definition can be realized by changing only a relevant segment in a large program without altering its general structure. This paper presents the design of a moderately complex program using stepwise refinement. Following the statement of the problem, a top-down design of an algorithm, for solving the problem using the stepwise refinement approach, is presented in detail,

* School of Automation.
** Department of Applied Mathematics.

to bring out the fact that the program developed using this algorithm, is well-structured from the point of view of ease of comprehension of the program and its logic.

2. Structured programming

Programming of early digital computers was mainly restricted to the use of machine language; even simple problems resulted in reasonably large programs. Subsequent to the introduction of high level languages, the digital computers were utilized extensively in diversified fields of activity. Programming continued to be an art to a large extent, and every programmer had his own ' bag of tricks ' in writing programs. There was a need for the programs to be extremely efficient, in that they utilize minimum execution time and main memory of the computer ; saving of bits and microseconds was treated as great virtues.

With the advents in hardware technology the basic hardware of a computer became extremely fast. This, in turn, enabled users to consider problems of greater complexity for processing on computers. Development of such complex programs posed problems of their reliability. The need for systematic procedures to enable the development of reliable software was felt and one such technique for achieving this goal, due to Dijkstra, is called structured programming[1].

Structured programming aims at developing a computational algorithm for a given problem by breaking down the problem into subproblems, the complexity of each subproblem being much less than that of the original problem. This method of program development is also known as the method of top-down stepwise refinement, since in this approach one starts from the problem specification, splits the problem into subproblems, until the details involved in a given subproblem are of a manageable size and complexity. A recursive definition of stepwise refinement[2, 3] is the following :

Solve (problem) :

1. Obtain a precise statement of the specifications to be met.
2. Formulate a simple, iterative solution naming as subproblems, any trouble spots encountered along the way.
3. While any subproblem remains unsolved

Solve (subproblem):

The method of top-down design is of great help as it permits one to defer the consideration of the details of solving the subproblems until the major steps involved in the solution of the problem are clearly identified. It also has the advantage that each time an error in any part is found or an improved method of solving that part is discovered, one can refine that particular part of a computational algorithm without having to

abandon the entire program developed so far. This method of program development demands writing down the various levels of details of a specific problem as one goes through the development process. In the literature, certain constructs have been identified, which can be conveniently used to represent this top-down design at every level, so that the final write-up corresponds to a program in a programming language. Boehm and Jacopini[4] have identified three language constructs, *viz.*, sequencing, conditional, and repetitive statements that are sufficient for designing any program. Hence these three constructs are normally utilized to represent the various details involved in a particular level in the stepwise refinement design. Programming languages like Pascal have statements for each one of the above-mentioned constructs and hence, the final level in the stepwise refinement approach lends itself easily to be coded as a program in such languages.

With a view to present the philosophy of structured programming, a case study of a non-numeric programming task is presented in section 4, indicating the various stages of program development.

3. Some definitions in first-order logic

First-order logic is a formal language useful for symbolizing logical arguments in mathematics[5–8]. The sentences in this language are called well-formed formulas (*wffs*). The symbols from which *wffs* are constructed are listed below :

(*a*) Truth symbols: T and F.

Connectives: $\sim$ (not), $\supset$ (implication), $\wedge$ (and), $\vee$ (or).

Punctuation marks: ((left parenthesis),) (right parenthesis), , (comma);

(*b*) Quantifiers: $\forall$ (universal quantifier) and $\exists$ (existential quantifier);

(c_1) A denumerable set of symbols called individual variables;

(c_2) A denumerable set of symbols (not in c_1) called individual parameters; and

(*d*) For each positive integer n, a denumerable set of symbols called n-ary predicates.

1. *Atomic formula* (*atf*) :

An $(n+1)$-tuple $Pc_1, \ldots, c_n$ where P is an n-ary predicate, and $c_1, \cdots, c_n$ are any individual symbols (variables or parameters), is called an atomic formula.

2. *Well-formed formula* (*wff*) :

A *wff* is recursively defined as follows :

(*a*) Each *atf* is a *wff*.

(*b*) If A is a *wff*, then so is $\sim A$,

(*c*) If A and B are *wffs*, then so are

$(A \wedge B)$, $(A \vee B)$ and $(A \supset B)$.

(*d*) If A is a *wff* and x is a variable, then

$(\exists x)\, A$ and $(\forall x)A$ are *wffs*.

3. *Prenex normal form* (*pnf*) :

A *wff* is said to be in *pnf*, if it is of the form

$$(q_1 x_1) \ldots (q_n x_n)\, (M)$$

where each q_i is one of the quantifier symbols '$\exists$', '$\forall$' and $x_i \neq x_j$ for $i \neq j$ and M is a quantifier-free formula.

Any *wff* can be converted into its *pnf* equivalent. The proof is based on the following *pnf* basic equivalences. In these equivalences, $\phi(x)$ is any formula, ψ is a formula, y is a variable which has no free occurrence in $\phi(x)$ or ψ, and $\phi(y)$ is the result of substituting y for all free occurrences of x in $\phi(x)$.

$$\sim (\forall x)\phi(x) \simeq (\exists x) \sim \phi(x)$$

$$\sim (\exists x)\phi(x) \simeq (\forall x) \sim \phi(x)$$

$$(\forall x)\phi(x) \wedge \psi \simeq (\forall x)\,[\phi(x) \wedge \psi]$$

$$\psi \wedge (\forall x)\phi(x) \simeq (\forall x)\,[\psi \wedge \phi(x)]$$

$$(\exists x)\phi(x) \wedge \psi \simeq (\exists y)\ [\phi(y) \wedge \psi]$$

$$\psi \wedge (\exists x)\phi(x) \simeq (\exists y)\,[\psi \wedge \phi(y)]$$

$$(\forall x)\phi(x) \vee \psi \simeq (\forall y)\ [\phi(y) \vee \psi]$$

$$\psi \vee (\forall x)\phi(x) \simeq (\forall y)\ [\psi \vee \phi(y)]$$

$$(\exists x)\phi(x) \vee \psi \simeq (\exists y)\,[\phi(y) \vee (\psi)]$$

$$\psi \vee (\exists x)\phi(x) \simeq (\exists y)\ [\psi \vee \phi(y)]$$

$$\psi \supset (\exists x)\phi(x) \simeq (\exists y)\,[\psi \supset \phi(y)]$$

$$(\exists x)\phi(x) \supset \psi \simeq (\forall y)\ [\phi(y) \supset \psi]$$

$$\psi \supset (\forall x)\phi(x) \simeq (\forall y)\ [\psi \supset \phi(y)]$$

$$(\forall x)\phi(x) \supset \psi \simeq (\exists y)\ [\phi(y) \supset \psi].$$

These equivalences enable us to move interior quantifiers to the front of a formula. Prior to the use of basic *pnf* equivalences, the *wff* is rewritten with a view to eliminate redundant quantifiers, *e.g.*, $(qx)\ A$ is equivalent to A if x has no free occurrence in A,

and by renaming variables in order all quantifiers in *wff* will have different variables and no variable is both bound and free.

4. An algorithm for conversion of a *wff* to *pnf*

The algorithm for converting a *wff* into *pnf* is presented in the sequel as a sequence of steps.

Step 1 *Eliminate in X all redundant quantifiers* :

If a quantified variable does not appear within the scope of the quantifier in the *wff*, then such quantifiers will be deleted.

Step 2 *Rename variables* :

If the *wff* contains variables which are quantified more than once, then rewrite the *wff* so that every quantified variable is distinct.

Step 3 *Push quantifiers to the left of negation* :

If a quantifier appears to the right of a negation operator, then change the quantifier by its counterpart (replace existential quantifier by universal quantifier and *vice versa*) and move the same to the left of the negation operator.

Step 4 *Push quantifiers to the left of* Λ, V *and* $\supset$ *and extend scope of quantifier* :

If a quantifier appears to the right of a conjunction, disjunction, or implication, move the quantifier to the left, so that the scope of the quantifier covers both the operands of the given operator.

Step 5 *Extend scope of quantifier*:

If a quantifier has the left hand operand of a conjunction or disjunction for its scope, extend the scope of the quantifier to both the operands.

Step 6 *Replace quantifier and extend its scope*:

If a quantifier appears to the left of an implication operator, then replace the quantifier by its counterpart and extend its scope to both the operands of implication.

Step 7 *Repetition of steps for obtaining pnf*:

Repeat steps 3 through 6, until no more applications of these steps are possible. This final form will be the *pnf* version of the given formula.

The algorithm stated above has been presented, as a sequence of steps and it will become apparent that this helps the program development by stepwise refinement to a very great extent.

5. Illustrative examples

We illustrate the process of applying the algorithm for a given *wff* to arrive at the *pn* version. Three examples are presented in the sequel. The application of the algorithm for a *wff* results in a sequence of formulas, the last one being the required *pnf* version. The number in parentheses against each *wff* in the sequence refers to the corresponding step of the algorithm used in the rewriting process.

5.1. *Example* 1

wff : $(\exists x)\, Fx \supset (\sim (\exists y)\, Gy \supset (\forall t)(\exists y)\, H(x, y)$

Sequence :

$$(\exists x)\, Fx \supset (\sim (\exists y)\, Gy \supset (\exists y)\, H(x, y)) \quad (1)$$

$$(\exists w)\, Fw \supset (\sim (\exists y)\, Gy \supset (\exists z)\, H(x, z)) \quad (2)$$

$$(\exists w)\, Fw \supset ((\forall y) \sim Gy \supset (\exists z)\, H(x, z)) \quad (3)$$

$$(\exists w)\, Fw \supset ((\exists z)\{(\forall y) \sim Gy \supset H(x, z)\}) \quad (4)$$

$$(\forall w)\,[Fw \supset (\exists z)\{(\forall y) \sim Gy \supset H(x, z)\}] \quad (6)$$

$$(\forall w)(\exists z)\,[Fw \supset \{(\forall y) \sim Gy \supset H(x, z)\}] \quad (4)$$

$$(\forall w)(\exists z)\,[Fw \supset (\exists y)\{\sim Gy \supset H(x, z)\}] \quad (6)$$

$$(\forall w)(\exists z)(\exists y)\,[Fw \supset \{\sim Gy \supset H(x, z)\}] \quad (4)$$

5.2. *Example* 2

wff : $(\exists x)\, P(x, z) \supset (\forall z)\,[(\exists y)\, P(x, z) \supset (\forall x)(\exists y)\, P(x, y)]$

Sequence:

$$(\exists x)\, P(x, z) \supset (\forall z)\,[P(x, z) \supset (\forall x)(\exists y)\, P(x, y)] \quad (1)$$

$$(\exists x)\, P(x, t) \supset (\forall z)\,[P(x, z) \supset (\forall w)(\exists y)\, P(w, y)] \quad (2)$$

$$(\forall z)\,[(\exists x)\, P(x, t) \supset [P(x, z) \supset (\forall w)(\exists y)\, P(w, y)]] \quad (4)$$

$$(\forall z)(\forall x)\,[P(x, t) \supset [P(x, z) \supset (\forall w)(\exists y)\, P(w, y)]] \quad (6)$$

$$(\forall z)(\forall x)\,[P(x, t) \supset (\forall w)(\exists y)\,[P(x, z) \supset P(w, y)]] \quad (4)$$

$$(\forall z)(\forall x)(\forall w)(\exists y)\,[P(x, t) \supset [P(x, z) \supset P(w, y)]] \quad (4)$$

5.3. *Example* 3

wff : $[(\exists x)\, P(x) \vee (\exists x)\, Q(x)] \supset (\exists x)(P(x) \vee Q(x))$

Sequence:

$$[(\exists x)\, P(x) \vee (\exists y)\, Q(y)] \supset (\exists v)(P(v) \vee Q(v)) \quad (2)$$

$$\{(\exists y)\,[(\exists x)\,P(x) \vee Q(y)]\} \supset (\exists v)\,(P(v) \vee Q(v)) \quad (4)$$

$$\{(\exists y)\,(\exists x)\,[P(x) \vee Q(y)]\} \supset (\exists v)\,(P(v) \vee Q(v)) \quad (5)$$

$$(\forall y)\,\{(\exists x)\,[P(x) \vee Q(y)]\} \supset (\exists v)\,(P(v) \vee Q(v)) \quad (6)$$

$$(\forall y)\,(\forall x)\,[P(x) \vee Q(y)] \supset (\exists v)\,(P(v) \vee Q(v))] \quad (6)$$

$$(\forall y)\,(\forall x)\,(\exists v)\,[(P(x) \vee Q(y)) \supset (P(v) \vee Q(v))]. \quad (4)$$

6. Program development

In the sequel, using the structured programming approach, a program for converting a *wff* in first-order logic into *pnf* is described; this uses the algorithm presented in Section 4. The various stages of program development have been identified as levels. Level 1 is just the problem definition.

Level 1 Develop a program for converting a *wff* in first-order logic into prenex normal form.

The computation involves examining the formula term by term, and then rewriting the terms by resorting to a relevant step of the algorithm stated in the earlier section. This implies the scanning of the given formula a number of times depending upon the number of terms in it. It, therefore, becomes essential that the input in conventional infix notation be rewritten into a suitable form before processing starts. This suggests the logical sequence of steps stated as level 2.

Level 2 2.1. Accept the given formula and perform necessary preprocessing.

2.2. Rewrite the formula into *pnf* and print out the *pnf* version of the formula.

These two subproblems can now be studied in greater detail individually. The preprocessing for input has to be chosen, so that the subsequent computations can be efficiently realized on a computer. We will refine action 2.2 further, so that we get a better appreciation of the computations that are involved.

The given algorithm initially removes redundant quantifiers and renames variables. Level 3 suggests the sequence of steps for the same.

Level 3 3.1. Remove any redundant quantifiers from the formula.

3.2. Rename a quantified variable, if it is quantified more than once.

3.3. Obtain the *pnf* version of the formula and output the same.

The refinement of each one of these actions in level 3 follows. In the sequel, the various Pascal constructs are utilized for depicting control flow.

```
Level 4 {Removal of redundant quantifiers}
        begin
          for every quantifier do
          while (within the scope of the quantifier) do
          begin
            if (no occurrence of quantified variable)
            then remove that quantifier from the formula
          end
        end

Level 5 {Renaming}
        begin
          for every quantified variable do
          if (name = another quantified variable or name = a free variable)
          then
            begin
            rewrite name by a new variable name;
            while (within the scope of the quantifier) do
              begin
              for every occurrence of the name
              do rewrite the name by the same new name
              end
            end
        end

Level 6 (Obtain pnf version)
        begin
        6.1.  while move possible do
              begin
                for every quantifier do
                begin
```

if (quantifier's scope is the right of negation)

then begin

change quantifier to its counterpart; move it to the left of negation extending its scope to include negation.

end

else

if (quantifier's scope is the right operand of conjunction, disjunction or implication)

then move the quantifier to the left so that its scope includes both the operands of the operator

else

if (quantifier's scope is left operand of conjunction or disjunction)

then extend scope of quantifier to both the operands of the operator

else

if (quantifier's scope is left operand of implication)

then change the quantifier to its counterpart and extend its scope to include both the operands of implication.

end

end

6.2. Print out the *pnf* version.

end.

Having identified the details of computation needed for the conversion, we can decide upon our internal representation for the formula. As the computations require scanning of the formula in units of terms, the postfix notation has been identified as the most suitable representation. The preprocessing action therefore corresponds to rewriting the formula into postfix notation before main computation starts. This leads to the refinement of action 2.1 in level 2 as follows:

Level 7 {Preprocessing}

7.1. Read in the formula and check for possible input errors.

7.2. Convert the formula into postfix notation.

As the computations are now performed on the postfix form of the given formula, the final result will also be available in postfix notation. Hence we have to rewrite this into conventional infix notation. This results in the next level of refinement of action 6.2.

Level 8 {Postprocessing}

8.1. Convert the *pnf* in postfix notation into infix notation.

8.2. Print out the infix notation of the *pnf* version.

It is to be noted that any of the actions in the last four levels can be detailed further if one so desires. As the subsequent levels of refinement are easy to realize, we leave the stepwise refinement of program development at this stage.

7. Program implementation

The program implementing the above-mentioned algorithm, using this stepwise refinement approach, has been coded in Pascal[9]. Input and Output are coded as three procedures. Each one of the steps in the levels 4 through 8 have been coded as one or more procedures. Levels 4 and 5 have been coded as one procedure each; level 6 has resulted in three procedures. Levels 7 and 8 have been together coded as three procedures. The formula and its postfix form have been represented as sequences and the one-way linked list structure has been utilized to hold the *pnf* version. The complete program contains about 500 Pascal statements.

8. Remarks and conclusions

The stepwise refinement technique has been elucidated with the help of a non-numeric example, instead of the common presentations of numerical examples. But for the adoption of the philosophy of structured programming and the technique of top-down program design, the final program would have been quite large and messy, making it difficult to comprehend or modify the program.

9. Acknowledgements

The authors are grateful to Prof. R. Narasimhan, Director, NCSDCT, Tata Institute of Fundamental Research, Bombay, for providing the necessary computational facilities.

References

1. DIJKSTRA, E. W. *A Short Introduction to the Art of Programming*, TR EWD 316, The Netherlands (Indian reprint published by the Computer Society of India, 1977).

2. KIEBURTZ, R. B. *Structured Programming and Problem Solving with ALGOL-W*, Prentice-Hall, 1975.

3. BOEHM, C. AND JACOPINI, G. Flow-diagrams, Turing Machines and Languages with only Two Formation Rules, *CACM*, 1968, 9 (5).

4. WIRTH, N. Program Development by Stepwise Refinement, *CACM*, 1971 **14**, 221–227.

5. SMULLYAN, R. M. *First-Order Logic*, Springer–Verlag, 1968.

6. Kleene, S. C. *Mathematical Logic*, John Wiley, 1968.

7. MENDELSON, E. *Introduction to Mathematical Logic*, Van-Nostrand, 1964.

8. MANNA, Z. *Mathematical Theory of Computation*, McGraw-Hill, 1974.

9. JENSEN, K. AND WIRTH, N. *PASCAL — User Manual and Report, Lecture Notes in Computer Science* 18, Springer-Verlag, 1974.