数理解析研究所講究録
第 586 巻 1986 年 35-43

# Programming Language Designs to Support Programming Methodologies

Hayashi, Tsunetoshi*

北海道大学大型計算機センター 林 恒俊

## 1. Introduction

It becomes not unusual that a large number of large scale softwares have been developed and are in continuous use for a long period of time. To maintain such a large scale software over its whole life cycle, it is indispensable for its principles in solving problems and their description in programs to be easily understood by programmers other than the author, and by the author himself long after the development as well. Proper programming methodologies should be used to make programs understandable and readable. Structured programming, top-down programming, stepwise refinement, and bottom-up programming (software components) are some of such programming methodologies which give programmers support to write understandable programs. The analytical, or deductive, approach like top-down programming seems to have more descriptive power than the synthetic, or inductive, approach such as bottom-up programming. In the top-down approach, a program is developed in the same way as a programmer solves the problem by expanding it into subproblems. Then the subproblems are further divided over again until a partial program will emerge by itself as a solution to each subproblem. This approach inherently coincide with the natural working of human mind, so a program written in this way often becomes very understandable.

Using such a methodology, however, is not a sufficient condition for a program to become understandable. There are lots of garbage programs without goto statement. A programmer should obey the strict discipline of the programming methodology used and direct his way of thinking along its paradigm while writing his program. There should be some software tool, or programming language, to support, or to enforce, such disciplines.

It seems, however, there is slight discordance between the programming language designs, especially modern ones, and such top-down programming methodologies. The modern design unwittingly supports the bottom-up approach rather than top-down one. This fact sometimes discourages a programmer to follow the analytical, or deductive, discipline. Therefore we should reconsider the current design tendency of a programming language, and/or devise some means for coping with its defect.

On the other hand, in the bottom-up approach, several functional program modules are prepared in advance, and they are used as building blocks with which a complete program is to be constructed. It is rather difficult to understand the total logic of a program by induction on what its constituent modules are doing. It is difficult as well to prepare requisite

* Associate Professor, Hokkaido University Computing Center, Kita 11 Nishi 5, Kitaku, Sapporo, Hokkaido 060, Japan

functional program modules in advance, since we cannot know what module is necessary until we are forced to use that module.

In addition, a program must have its precise documentation along with its source code. The source code alone is insufficient for a reader to become acquainted with the logic of the program, whatever detailed comments are dispersed among the source code. Comments are just comments and cannot be compared with a readable document. It is rather indispensable than desirable that a descriptive program document should be written hand in hand with the source code development. And the tool should also encourage a programmer to write documentation as well as the source code simultaneously.

In this paper, we would like to give considerations on the design principles of a programming language, or software tool, which inherently support programming methodologies and documentation.

## 2. Programming Methodology vs. Programming Language

As stated above, the modern programming language design does not necessarily support to write readable programs. The reasons are: (i) it does not help a programmer to write a program in top-down manner; (ii) the contextual distance between a definition and its reference sometimes grows very far, for example, a few tens of pages in a large program; (iii) it does not to help recording the problem solving steps. These are discussed in the following.

There is a slight discrepancy among the natural textual orders of pieces of program elements defined by a programming language syntax and the top-down programming methodology. The program elements in this context are names of variables, procedures, labels, and so on.

Programming languages, especially modern ones, request that the the definition of a name should have been completed when that name is to be accessed. For example, in Pascal [JW76], the names must be declared in the order **label**, **constant**, **type**, **var** followed by **procedure** and **function**. A variable to be accessed in the program body must be declared explicitly in the **var** part. Even in the declaration part, a name representing a symbolic constant in the **type** part must appear in the **constant** part. For some special case of mutual recursion, which cannot meet above principle, Pascal provides special programming devices such as pointers ($\uparrow$) with deferred type and **forward** specification. This is also the case for C language as well as Ada [Ad83], where the declaration should precede reference contextually. Ada also provides special programming devices: the specification declaration and body definition; and incomplete type declaration.

This ordering is very natural for compiling a source program. The definition and specification of variables and subprograms are perfectly completed when they get referred to later in their scope in the source code text. It simplifies the compiler design as well, as the whole source program need not be kept in the memory all the time.

This ordering also helps to write a program in the bottom-up approach. The natural ordering of program elements written in this approach is "declaration to reference." Subprograms are written as functional modules prior to their invocation. Variables should be declared in advance at the beginning of a program.

On the other hand, the natural ordering defined by the top-down approach is contrary to the above one. According to this approach, a higher, abstract level part of program comes first, then its lower, more concrete parts come next contextually, and so on. In other words, if we assume these parts are written as subprograms, reference to them precedes their declarations, as shown in (1).

```
program
    ...
{ abstracted action is invoked }
    the_action;
    ...
{ here abstracted action is further expanded }                    (1)
    procedure the_action;
        ...
    end
    ...
end
```

The specification for variables should be also determined with respect to the requirement emerged while expanding the problem. This implies the variable declaration should follow, or be parallel to their reference in a partial program. This ordering does not agree with the current programming language. A programmer is always required to make conscious effort to think in the other way than the paradigm a programming language defines while writing a program. In a sense, rather classic languages, Algol 60, PL/I, and even FORTRAN are superior to modern languages in this respect. In PL/I and FORTRAN, the declarations and the other statements can be intermixed freely. There is no restriction on the order of subprogram definitions in Algol 60 either.

To ease the task to read a program, it is desirable that variables should be declared contextually near to the place where they are most actively referred to as much as possible. The programming language design cannot meet this condition either, especially for so-called global variables. Variables common to several subprograms is to be declared globally, that is, outside to those procedures. They are put at the beginning of a program. Then, several intervening subprograms tend to follow for a few pages. And, finally subprograms referring to those variables appear. A reader must turn pages several times when he is reading that part of a program.

To understand the problem solving logic of a program, it is indispensable to know how it is developed in each step of top-down programming. A programming language also fails to meet for this condition in another point. A final program written in the top-down approach can hardly record the developing steps in itself. The only thing we can do is that the abstraction done in one step is reserved in the name of a subprogram with explanation in comments. This tends to develop rather large number of subprograms, which makes bad run-time efficiency. And explanation in comments tends to be of poor quality. A programming language should provide some means to record such steps, other than subprogram abstraction mechanism.

```
program
    var
{ global variables are declared here }
    global: ···
    . . .
{ interveneing procedures several pages long }
    . . .
    procedure ···                                         (2)
        . . .
{ a global variable is accessed here }
        global← ··· ;
        . . .
    end
    . . .
end
```

## 3. Program Document vs. Comments

In reading a program, there is some difficulty in the program representation itself. There are several representations of a program written in some program language. Some of them are, for example, hardware representation and publishing representation. The hardware representation is machine readable and used for storing programs in a computer. The publishing representation is for printing source codes in a publication. Here, keywords, names, symbols and operators are printed in different font styles and are mutually distinguishable. This convention makes a program source code often very readable. The conversion from hardware represenation into publishing format could be done automatically by a sophisticated pretty-printing (pretty-typesetting?) program. It is desirable that a language processor should provide such facility, rather than using a separate software tool.

Although a program source code itself has some descriptive ability, it is insufficient to give an acount of how the program is actually working. Informal explanation sometimes excels formal description. Therefore, almost all languages are provided with additional means for explanation—comments—within their basic syntax. Comments are usually ignored by the processor, still they are parts of a source code and are subject to the source program syntax in general. In some language, a particular combination of characters is not permitted in a comment, as it might terminate the comment in which it appears. Comments cannot be unrestricted, full, free texts, they can be used only for simple explanation.

A programming language should provide a facility to write a (possibly fully typeset) document as well as source code description. Both of these should be done hand by hand simultaneously. It might be better to write a document with pieces of programs interspered than a program with comments interspersed. A language processor can be made to accept such a document and extract the source program codes.

## 4. The Case of WEB

WEB language [Kn83, Kn84] developed by Knuth can be seen as an example which can fill the gap between the current programming language design and programming methodology. The concept involved in WEB is called literated programming. WEB is founded on Pascal language and TEX typsetting program, and used chiefly for writing TEX itself. The WEB basic function provides facilities to write a program in top-down manner; to record refinement steps; and to generate a document along with a program source code. There are also some bells and whistles added to the basic function.

It is said that Pascal lacks sufficient functionality for large scale systems programming. The primary purpose of WEB is to complement weak points and to enhance program portability of the basic language, Pascal. WEB is very successful with respect to its purpose, since a good number of large scale programs are written in WEB, which have been transported to many kinds of computers. The source code of TEX itself is to be published [Kn85]. This shows the effectiveness of WEB as an information communicating medium.

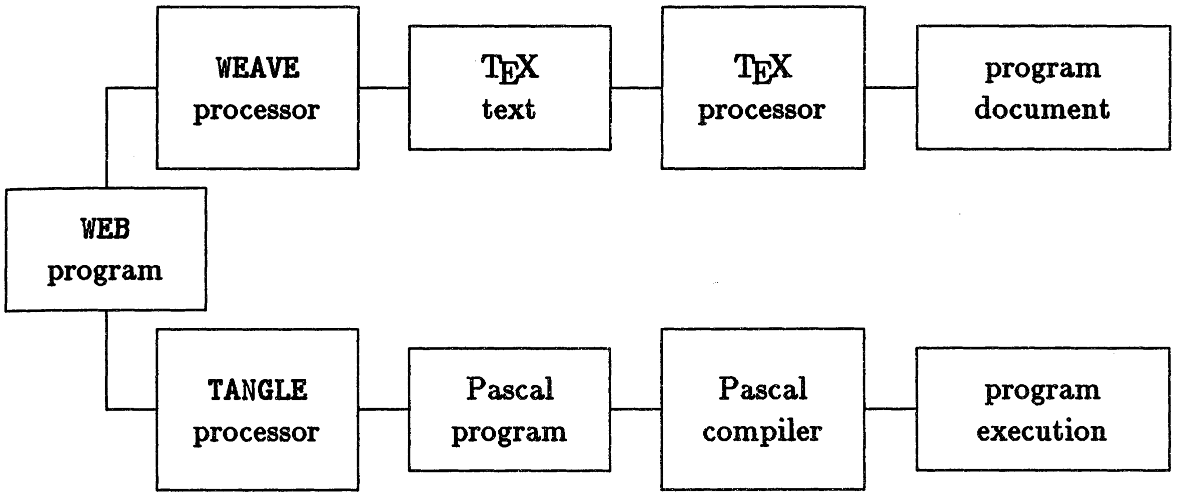

Figure 1. WEB configuration.

The basic design principles of WEB are:

- A WEB source code looks like a document text with special mark-ups rather than a program source.
- The mark-ups discriminates the source text into several kinds of texts, for example, document text, Pascal program text, module name and definition, macro definition, indexing specification, etc.
- A WEB source code is divided to several pieces, or modules in WEB terminolgy, and a module is the unit of programming. A module occupies at most one page when it is typeset.
- A module consists of at most three parts: a document text part; a macro definition part; and a Pascal program part, in this order.
- The Pascal program part may be given a name, which is an arbitrary text. Statements in the Pascal program part may refer to this name. This name serves as a pseudo-instruction. The definition and reference to module names can be in any order.
- There are two WEB processors, one to translate a WEB source text into a Pascal program, and the other to convert it to TEX input text. (See fig. 1.) The former, called TANGLE,

generates a Pascal program by replacing the module names with its definitions. Modules with the same name are collected and concatenated together. The latter, called WEAVE, generates a TEX source text, which will be then typeset as the program document.

Using the module names, we can write a WEB source code in the top-down manner, and record the developing steps. At the abstract level, we can use a name instead of Pascal statements, and then a module with that name is developed later. Global variables can be declared at several places by using the name attached to the module containing **var** statement, as shown in (3). The generated Pascal program looks almost the same as (2).

```
⟨ global variables ⟩ ≡
{ a module to define global variable }
    var ···
...
⟨ outer module ⟩ ≡
{ another module in which module ⟨ the_action ⟩ is invoked }
procedure ···
    ...
    ⟨ the_action ⟩;
    ...
end;                                                              (3)
...
⟨ the action ⟩ ≡
{ here ⟨ the_action ⟩ is defined }
    ...
{ a global variable is accessed }
    global← ··· ;
    ...
⟨ global variables ⟩ +≡
{ global variables are to be declared }
    global: ···
    ...
```

There are some shortcomings or defects in WEB language. First, one must juggle with three mutually much different languages in writing a WEB source code, namely, WEB language, TEX source language, and Pascal. Good understanding of these languages are required. The number of languages involved should be small as much as possible. Second, the coupling between a WEB source and generated program is rather loose. This is because the WEB processors are implemented as preprocessors. There is no way to reflect the source module organization to the generated program at run-time. The program must be executed independent to the source WEB program.

## 5. Literated Programming Language Design Principles

A programming language supporting top-down approach and documentation should be designed to the following criteria. The language is called LLP (Language for Literated Programming) tentatively. The LLP can establish itself as a new language with its own processors and environment. It need not depend on some existing language.

**Segmentation** The LLP source program is divided into several segments. Each segment represents a refinement step of the top-down programming activities. Not the syntactic structure of programming languages but the internal logic of a program should determine the segmentation, and then refinement process. In other words, it must be made that, in the LLP, the syntactic and logical aspects of programming should be separately dealt with. A segment may include program document text, partial program itself, or both. A segment should not be too long and better not span over one page when formatted.

**No mark-ups** The mark-ups are not to be used for designating the document part and program part of a segment. Instead, some means like shift/escape sequence may be used for this purpose. Or else, compiler directives like '#' lines in C language may be employed. This relieves a programmer from remembering superfluous subtleties. The point is that a programmer should be able to handle the both text and program parts in the programming language framework.

**Flexible intrasegment structure** The program part and text part in a segment may be interleaved in arbitrary order. The WEB segment structure is rather rigid: a segment must be constituted from the text part, macro definition, and program part in this order, where some parts may be omitted. This turns out to be rather inconvenient as related pieces of programs and variables declaration must be put in separate segments.

**Segment naming convention** Either a segment or each piece of program texts in a segment may be given adequate name. The name can be any arbitrary text as in WEB. This name serves as a pseudo-code in the referring occurrence, and can be used as a lexical item in a piece of program text. There should be some means to abbreviate a name when it is appeared second time or later.

**Name definition and reference** The order of definition and reference of a name is arbitrary, moreover, a same name could be defined at several places. This means consecutive pieces of program text are scattered among several places.

**Macro definition** It is desirable the LLP provides macro programming function. However, this may be realized by the named segment convention. It is much useful that a named segment could accept parameters which will be substituted for the formal arguments in the piece of program text. In this way, the module definition and macro definition in WEB can be unified.

**Overall consideration** The source program including both text and program parts should be programming language-oriented rather than text-oriented. This is required for giving concise and precise explanation in the program document. This point is important for the batch-oriented implementation, where the formatted document and the source program text might be much different.

The programming language of the program part can be arbitrary as stated before. It may be a completely original one designed to be best suited for the literated programming. It can be a generic algorithmic language, from which a program in any language can be generated. The LLP "TANGLE" processor may handle such conversion. This point leaves much freedom for the LLP design.

## 6. Implementation as Programming Environment

There are several conceivable ways to implement the LLP. For example, WEB employs conventional preprocessor implementation in the batch-oriented processing mode. WEB has two processors, TANGLE and WEAVE, their functions are document preparation and program preprocessing respectively. Both of them run on the same WEB source, but at the different time. Therefore the program and its document are obtained separately. The TANGLE rearranges the source code by replacing segment names with their definitions, and generates a program which can be processed by a compiler. The WEAVE does virtually nothing but inserts directives for pretty-printing and generates the index and cross-reference. The implementation in this way makes the coupling between the source program preparation, generated program, and the program document rather loose. It is very important that we can settle only with the single source program preparation by strengthening the coupling.

We had better devise a means to strengthen the coupling to make the LLP a good programming tool. The means may be (i) improving the LLP compiler in the batch mode processing; and (ii) implementing the LLP in the interactive mode processing as a programming environment.

In the batch mode processing, a single compiler can be made to handle both parts of the LLP source input. The unified compiler, which may have processors such as given in the above as its phases, can be made to accept the source program and generate a program document and an object at the same time. In this way we may get along with the LLP source program only.

The LLP will work best in the interactive processing mode, where only the program document need be prepared. Sufficient check will be done over the source program by combining the editting and syntactic processing. Neither mark-ups nor special LLP own syntax are required. The document will be printed as is shown on the screen. The program can run in the interpretive/interactive execution, debugging and run-time profiling can be made reflecting the source segment structure. A program may be generated as the final result, which can be preserved for later batch mode compiling.

In this case, a problem is that only one page of the program document can be displayed on the screen. This will become obstructive when developing a large scale program, where the number of related segments might grow large and they might span over several pages. The semi-automated multi-segment access through the multiple window mechanism will solve the problem. Related segments can be accessed and displayed on the separate windows automatically by referring to the symbol cross-reference and index database when editting a segment. Such database can be created and maintained by the interactive LLP processor. In a sense, the LLP should be implemented as a programming environment in the interactive mode processing.

## 7. Conclusion

So far, we have discussed the relationship between programming language design and programming methodology, from the point of view of program understanding. The current design of main-stream programming languages does not tend to support the programming methodologies adequate for writing understandable programs. In addition, current programming languages fail to support programming as documentation proccess.

We should devise some means, which we called the LLP, to fill the gap between the programming language and programming methodology. We presented the design criteria of the LLP, which are extracted from the experience in using the WEB language. Finally some LLP implementation techniques are considered and their merits and demerits are discussed. An interactive one implemented as a programming environment will be most desirable in this context.

## References

[JW76] K. Jensen and N. Wirth, *PASCAL User Manual and Report in Lecture Notes in Computer Science* **18**, Springer-Verlag (1976).

[Ad83] *The Programming Language Ada Reference Manual in Lecture Notes in Computer Science* **155**, Springer-Verlag (1983).

[Kn83] D. E. Knuth, *The WEB System of Structured Documentation*, CS–980, Stanford University Computer Science Department (1983).

[Kn84] D. E. Knuth, "Literate Programming," *Computer Journal* **27** (1984), 97–111.

[Kn85] D. E. Knuth, *TEX: The Program*, Addison-Wesley (1985).

[Ha85a] T. Hayashi, "Literate Programming Considered as a Means of Stepwise Refinement," *WGSF* 13–6 (1985), (in Japanese).

[Ha85b] T. Hayashi, "A Structured Documentation Language for FORTRAN," *The 31st IPSJ Meeting* 8F–7 (1985).