Structured languages

by LEON PRESSER

University of California
Santa Barbara, California

A design process is a highly creative undertaking that has been generally guided by personal experiences and gross rules of design. This fact has been made explicit and at present the essence of the process of design is under investigation. This important problem is receiving attention in Computer Science. In particular, disciplined program design and development is the center of much discussion.

Conventionally a (software) design and development process commences with requirements analysis and proceeds through design, coding, testing, and finally the product moves into an operational environment and subsequent maintenance. The problem of error isolation and correction (i.e., debugging) is ubiquitous and the overall process is characterized by multiple iterations through various parts of the sequence of steps outlined above. In addition, appropriate documentation is generated throughout the project. The evaluation of this overall process is a complex issue that requires that 'evaluation' be a key objective from the very inception of the project through at least the beginning period of its operational lifetime. Currently, evaluation of design and development methodology is starting to receive attention but it may be some time before any substantial results are obtained. However, it is generally believed that coding itself accounts for a relatively small portion of the total effort (~?).

Our objective here is to discuss structured (i.e., disciplined) languages and, with the context provided by the above paragraphs as a base, we can now proceed to highlight some of the important issues.

A structured language is one that, in addition to matching the needs of an intended application area, is designed to satisfy some specific objectives. Popularly, the main objective has been to produce a language that forces code to be written in a manner that prevents errors. This in turn implies clear documentation with a static version (i.e., listing) that resembles closely the dynamic behavior of the code. Another possible language design objective could be to have code written in such a manner that locality is explicit and performance in a virtual environment is improved. Still another goal might be to make testing more explicit. Let us concentrate on the error prevention objective.

Any program can be constructed with the following two control structures (Figure 1):

- **goto** (transfer of control forwards or backwards to any point)
- **predicate** (conditional transfer of control forwards or backwards to one of two possible points)

These two control statements represent a complete set for a programming language. However, the freedom provided by the potential use of the *goto* in an undisciplined fashion defeats the objectives previously outlined; in particular, it conflicts with the desire to have static and dynamic versions that are close. Consequently, the following complete set of control structures is more appropriate for our purposes (Figure 2):

- **sequence** (transfer of control to next point in program sequence)
- **if-then-else** (conditional transfer of control to one of two operations which upon execution transfer control to next point following if-then-else in program sequence)
- **loop** (transfer of control so as to repeat an operation as long as some specified condition, which may be placed anywhere in the loop, is true; when the condition is no longer true control is transferred to next point following the loop in program sequence)

It is important to note that with this trio of control structures transfer of control occurs only in a disciplined manner. That is, with this set transfer of control always proceeds forwards in a strictly sequential fashion, possibly skipping the next operation in the sequence. Transfer of control backwards takes place only within the loop construct. Furthermore, each one of these structures possesses a single entry and a single exit point as shown in Figure 2. It is common to extend the if-then-else construct, which is a two-way branch, to an n-way branch called case.

We have thus far delineated the type of basic control structures that define a structured language. Additional discipline on the overall structure of programs, in order to facilitate readability and compile-time checks, may enforce the following format:

```
<program> ::= <declarations> <executable code>
```

Moreover, it is an objective of structured language design...
to make basic programming language principles explicit. For example, scope and protection issues may lead to the following program format:

\[
\text{<program> ::= <global declarations>}
\]
\[
\text{<local declarations>}
\]
\[
\text{<executable code>}
\]

In addition, a very desirable data definition facility may lead to the following format:

\[
\text{<program> ::= <definitions and declarations>}
\]
\[
\text{<executable code>}
\]
\[
\text{<definitions and declarations> ::= <global section>}
\]
\[
\text{<local section>}
\]
\[
\text{<global section> ::= <global new data structure definitions>}
\]
\[
\text{<global declarations>}
\]
\[
\text{<local section> ::= <local new data structure definitions>}
\]
\[
\text{<local declarations>}
\]

Furthermore, the important issue of attempting to enforce a design discipline through a structured programming language impacts on the final format of programs. This point is discussed in detail by White and Presser. Hence, the overall structure of the programs produced with a structured language is shaped by the aggregate of objectives that determined the language design. In particular, it should be clear that a structured programming language may be used as a tool to enforce discipline that impacts well beyond the bounds of the coding phase. Let us now address the practical problem of obtaining a structured language.

There are three ways to generate a 'structured language', these are:

1. Imitation—Through the use of appropriately placed comments an existing language (e.g., Fortran) may be made to look like a language 'designed' around the three basic control structures previously described.

2. Preprocessing—A structured language may be designed such that programs written in it may be translated into an existing language for subsequent compilation and execution. In some cases this approach may hamper the features possible in the structured language.

3. Design—A structured language is obtained through design and direct implementation.

The last approach mentioned above is the most attractive one, however, as is commonly the case with any new programming language, vested interests are extremely difficult to overcome. On the other hand the importance of the issues under discussion is becoming clearer and the impact may be felt with some strength in the years ahead.

Finally, I would like to end by mentioning a couple of points of interest based on my personal experience in the direct design of a structured language. First, a programming language designed with objectives of structure tends to be almost free of ambiguities; this is due to the explicit treatment of principles and the inherent redundancies. Second, in the past I have used as a metric of the 'smoothness' of a programming language a count of the number of words like 'except', 'but', etc., present in the manual that describes how to use the language. The smoothness of a language appears to be indirectly proportional to this number, and a structured language tends to be quite smooth!

REFERENCES