What Makes a Good Programming Language ?

Uday P. Khedker
Technical Report TR-97-upk-1
(October 1997)

Department of Computer Science
University of Pune
Pune 411007

What Makes a Good Programming Language ?

Uday P. Khedker
(uday@cs.unipune.ernet.in)
Department of Computer Science, University of Pune.

Abstract

A useful programming language need not possess the qualities that may make it a good programming language. Experience suggests that a programming language is good if it helps us write programs which are easy to read, easy to understand, and easy to modify. These requirements – which can only be expressed intuitively rather that defined formally – arise directly in response to the first two of the three fundamental requirements of software : reliability, maintainability, and efficiency. In this chapter, we look at the qualities that help a programming language meet the requirements of a good language.

1 Introduction

The degree of usefulness of a programming language is no indication of its qualities as a *good* language. Nor is a good language necessarily useful.[1] However, people often do not distinguish between usefulness and goodness of a programming language. Subconsciously, most people tend to emphasise one and take the other for granted by assuming that either the other aspect follows naturally or it is insignificant. In either case, one of the two aspects is sacrificed reducing either the effectiveness or the relevance of programming drastically.

Any measure of the goodness of a programming language must be intimately related to the three fundamental requirements of software : reliability, maintainability, and efficiency. Without implying that it is insignificant, we would like to consider efficiency as of secondary importance since seeking efficiency at the cost of reliability and maintainability is fraught with disastrous consequences — efficiency is important *after* reliability and not *before* it.

Usefulness, on the other hand is intimately related to the goals, rather than the qualities of software. In the larger perspective of software life cycle, usefulness is evidently a necessary but not a sufficient requirement. Usefulness alone cannot guarantee reliability and maintainability of software which are of prime importance in the longer run. It is this realisation that suggests that a programming language should support writing programs that are easy to read, easy to understand, and easy to modify.

The phrase “easy to read, easy to understand, and easy to modify” can only be expressed and appreciated at an intuitive level. Consequently, the qualities of a programming language for writing programs which are “easy to read, easy to understand, and easy to modify” cannot be defined formally or described accurately. However, we can express them intuitively by laying down some criteria based on experience.

We conclude this section by quoting an excerpt from the preface of (7) :

> Why do some ideas feel good? Perhaps there is a measure of beauty for ideas, and some are simply more appealing than others. One may worry, however, that a discussion of aesthetic issues is not compatible with the practice of computer science, and such arguments belong with other questions of taste, such as those about the right way to indent nested loops or the proper choice of variable names. . . .
>
> Should we reject aesthetic considerations as contrary to scientific method? Experience has shown otherwise. *Proper attention to the goals of aesthetics leads to measurably better designs.*

[1] Popularity and virtues need not necessarily go together, eh?

2 Programming Language Design

Programming language design does not imply merely listing the features which, when present in a language, make it possible to write useful programs. It is more like an art which seeks elegance of expression. In the context of programming languages elegance implies coherence, consistency, and accuracy of programs. This is achieved by exploring how the features desired in a language relate to each other as also how they relate to the programmer's visualisation of a software solution. If these relationships can be understood by programmers easily, the resulting programs would be easy to read, easy to understand, and easy to modify. Further, a good language design is of no significance unless it is accompanied by a good language implementation[2] — ultimately, it is the language implementation which realises the concepts defined by the language design. Hence a good design must also consider good implementation as one of the design goals.

The above discussion suggests the following "virtues" of a good programming language.

1. Writability and Readability
2. Simplicity
3. Definiteness
4. Orthogonality
5. Expressiveness
6. Implementability
7. Efficiency

Of the above seven qualities, the first five are the ones which are directly related to the programs that are easy to read, easy to understand, easy to modify. The last two qualities are guided by issues concerning language implementation and as such are beyond the scope of this chapter.

It must be emphasised that each one of the above qualities embodies a certain philosophy. These philosophies :

- can neither be defined formally, nor described accurately,
- may be conflicting (viz. readability and orthogonality), co-operative (viz. readability and simplicity), or independent (viz. readability and definiteness).

As a consequence, these qualities form only a "wish-list" in some sense. In practice there is no language that would meet the stringent – and often conflicting – requirements imposed by these qualities. Every language that we come across may receive a beating on some count while excelling on the other. When a language is found deficient on some count, it is deficient *only with respect to the particular criterion*. In the larger perspective, there may well be strong reasons justifying a particular decision leading to the feature which is perceived as a deficiency in the limited perspective of a particular criterion. What is painted here is an ideal scenario and we must evolve our own "concoction" of a good programming language by blending various qualities in a judicious way and to a reasonable extent by keeping in mind our own experience, familiarity, sense of comfort with a particular programming paradigm, and the ultimate goal of the intended use of the language.

[2]By a language implementation, we mean a programming system with a complete range of language processors, assorted tools and utilities for software development and maintenance.

3 Writability & Readability

Though a programming language is primarily a tool for communication with the machine, a program must also enable an accurate and precise communication between human beings concerned with the task of writing and maintaining the program. Dependence on external and internal documentation is not enough. This becomes obvious if one were given abundant internal and external documentation and asked to maintain and/or debug a large software written entirely in an assembly language!

The best documentation of a program is the program itself and a programming language must not only enable writing programs with clarity and understandability, it must prohibit writing program which are not clear and understandable. Though it is easier said than done, writability and readability of programs must be uppermost in any programming language design.

The issues of writability and readability are far more important and complex than they seem to be. They are not just two sides of the same coin. They are distinct issues often leading to conflicting requirements. Writability is motivated by convenience while readability is concerned with clarity. The ease of writing need not necessarily imply the ease of reading. In the longer run, the ease of reading seems to outweigh the importance of ease of writing since it is the ease of reading that enables modifiability and maintainability. The "powerful" and "convenient" features included in a language for writability often tend to obscure the programs. We consider the following important issues in this regard.

1. Keywords.
2. Abbreviations and concise notation.
3. Comments.
4. Layout or format of programs.
5. Overuse of notation.

It must be emphasised that this list is illustrative rather than exhaustive.

3.1 Keywords

Keywords in a program are the first to influence/affect readability. The choice and usage of keywords should ideally be governed by the following principles.

1. Keywords should be conceptually as independent as possible. For instance, in general there is not need to provide many iterative constructs. C provides

   - `for` ⟨*condition_and_increment*⟩ ⟨*statement*⟩,
   - `while` ⟨*condition*⟩ ⟨*statement*⟩, and
   - `do` ⟨*statement*⟩ `while` ⟨*condition*⟩.

   Each one of them can be used to simulate the other two. In particular, the presence of both `while` ...`do` and `do` ...`while` can cause a lot of confusion.

2. Keywords should represent a simple concepts directly.

   A gross violation of this principle is the keyword `static` in C. Local `static` variables are easy to visualise : They are guaranteed to retain their values across function invocations.[3] Global variables, on the other hand, are external to all functions and as such retain their values throughout the execution of the program. Hence they are static by default.[4] Still they can be explicitly specified as `static` where the keyword `static` restricts the visibility of a variable to a particular file. Then, why call it `static`? Why not `restricted` or something to that effect?

[3]Since they are allocated in the static area rather than on the stack.
[4]And this cannot be changed.

3. Keywords should be reserved so that they are not used as names in the programs. While this may seem to constrain the choice of programmer, how readable is the following statement from PL/I?

```
IF IF THEN THEN = ELSE ELSE ELSE = THEN;
```

   Apart from readability, this also affects analysis of the program and makes error detection more difficult.

4. A language should mandate use of few keywords unless they represent simple independent concepts. COBOL is a notorious violator of this principle. C goes to the other extreme by using library functions and excluding many features from the language per se.[5] Pascal follows the magical middle path.

3.2 Abbreviations and Concise Notation.

Shorthands tend to obscure the meaning and encourage trickery. For instance, C provides the conditional

$\langle \textit{condition} \rangle$`?` $\langle \textit{expression} \rangle$`:` $\langle \textit{expression} \rangle$

as a "powerful" operator for selection of values which can sometimes be used in place of the

`if` $\langle \textit{condition} \rangle$ $\langle \textit{statement} \rangle$ `else` $\langle \textit{statement} \rangle$

.

How readable is the following (single line) macro definition from the *lex* generated file lex.yy.c?[6]

```
# define input() (((yytchar=yysptr>yysbuf?U(*--yysptr):getc(yyin))
==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)
```

In fact C provides a large set of such shorthand notations and allows a programmer to write highly cryptic programs which is a certain disadvantage in the longer run.

If shorthands are bad, the alternative is not excessively long notation! A language must use reasonable abbreviations and reasonable notation.

- Familiar arithmetic operations like say multiplication can be represented by a `*` rather then forcing the programmer to use the keyword `multiply` or `times`.

  Trigonometric functions like *sin* are better not abbreviated to *s*!

- The keyword `procedure` may safely be abbreviated to `proc` but one must avoid the simultaneous abbreviation `prog` for `program`.

  `integer` may be abbreviated to `int` but is it advisable to abbreviate `real` to `re`? `external` may well be abbreviated to `extern` but not to `ex`.

[5]The advantage of using library functions is that they can be over-ridden. The disadvantage is that the compiler has no knowledge of the functions, hence their impact on the program cannot be checked at compile time. For instance, function `malloc` is used for dynamic memory allocation in C programs. The compiler can assume nothing about `malloc` (since it is just another function) and as such cannot guard against dangling pointers.

[6]Section 6.3 demystifies the macro.

3.3 Comments

There are three ways in which comments can be provided.

1. Single line comments starting with a particular character (or character combination) and ending with a newline character.
2. Multiline but non-nested comments with appropriate delimiters.
3. Multiline and nested comments with appropriate delimiters.

The third approach is most powerful and is likely to be most misleading since one may end up spending a lot of time trying to understand a code fragment which is commented out. The second approach also has the same danger but at least it imposes some discipline on the programmer since he cannot comment out code so very easily. He has to remove previous comments in order to avoid nesting. In either case, a misplaced delimiter may cause a lot of confusion.

First approach seems to be the simplest and the best from the viewpoint of readability though it affects writability. A misplaced delimiter in general would have a local effect and problems caused by runaway comments would become non existent.

3.4 Format of the Program

Unlike FORTRAN-77, COBOL, and BASIC, all modern programming languages are *free form* languages in that they do not require the program to adhere to a fixed layout. This has two connotations :

1. A program statement may be split across line boundaries, and
2. The lexical units of a program (i.e. the keywords, names, operators, punctuation marks, numbers, etc.) may be placed anywhere on a line except that a unit cannot be broken across a line boundary.

While free form structure permits a programmer to use tabs, spaces, and blank lines to make a program more readable, it is also a licence to potential unreadability since it does not *require* the use of tabs, spaces and blank lines except to separate some lexical units which, when combined, may make up yet another lexical unit (viz. keywords and variable names). Though FORTRAN-77 is not a free form language, it does not require spaces even between such lexical units. Thus FORTRAN-77 programs can potentially be the most unreadable programs, though in practice programmers almost always use spaces judiciously.

In general, programmers develop a certain style with experience and only novices seem to write programs which are unreadable due to inconsistent use (or abuse!) of the flexibility provided by the free-form structure.

3.5 Overuse of Notation

Overuse of notation impairs the readability of programs. For instance LISP uses parenthesis heavily and it is not unlikely for a certain program fragment in LISP to be deeply nested in dozens of parenthesis. In such a situation, it becomes very difficult to figure out the exact structure of a program and the place of a program fragment in the structure.

Consider the following LISP definition whose C version also follows.

```
( defun f (x)
  ( cond
   (((= x 0) (setq x 1))
    ((> x 0) (cond
     (((> x 10) (setq x (* 2 x)))
      ((< x 10) (setq x (+ 2 x)))
      ((= x 10) (setq x (+ 1 x))))))
    ((< x 0) (setq x 0)))))
```

Unless programmers take extra precaution in indenting the code, they will invariably be lost in the maze of parenthesis. The C version that follows can also be made rather unreadable but the use of `if` construct provides a much more understandable structure clearly separating the constituent sub-structures. Function `cond` in LISP, on the other hand, depends on parenthesis to distinguish between the various constituents of the structure.

```
f (int x)
 {   if (x == 0)
   x = 1;
  else if ( x > 0)
  {   if (x > 10)
    x = 2 * x;
   else if (x < 10)
    x = 2 + x;
   else if (x == 10)
    x++;
  }
  else if (x < 0)
   x = 0;
 }
```

A special case of overuse is usually evident in the choice of end signifiers. In order to minimise the keywords one may use the same delimiter for compound statements regardless of the control construct in which it appears. C uses { and } while Pascal uses `begin` and `end`. How many programmers put a comment at the end of the `while` statement as shown in the following code fragment?

```
while (something)
{
  /* do something */
} /* end of while */
```

From the viewpoint of readability, it would be better if we use the following kind of end signifiers

```
if    fi
case  esac
do    od
```

However, ending a `while` with `elihw` or a `select` with `tceles` could be very awkward. Use of `endwhile` and `endselect` seems a much more reasonable choice.

4 Simplicity

A language is simple if its features are easy to learn and remember. It is difficult to describe simplicity, it is much more easier to list the features which impair simplicity.

1. Simplicity is not achieved through a lack of structure; the result of this is chaos. The best example in this regard is that of machine languages which are often quite simple : the number of machine instructions is usually small and each instruction has a definite, easily understood effect. Yet, the lack of structure in data and control renders the machine language programs hard to understand.

2. Simplicity is also not achieved through limitless generality by allowing a rather large number of features. The result of this is a language which is extremely difficult to master truly and/or implement completely. PL/I was designed to provide the facilities of FORTRAN-IV, COBOL, and ALGOL 60 in one language! Advocates of PL/I often claim that it is not necessary for a programmer to be aware of all the features of the language. This may be fine for the programs that work as the programmer intended. However the program may not work as intended, but may accidentally invoke some unknown feature of the language. The larger a language, the more surprises it is likely to produce.

   Consider the following statement in PL/I which may have been accidentally written by the programmer.

   ```
   CODE = ('57' || 8) + 17
   ```

   This is accepted as a legal statement and is processed in the following way : Since `||` is a string concatenation operator, `8` is converted to a string and concatenated with `'57'` giving the string `'578'`. However, `+` is an arithmetic operator, hence the string is converted to number `578` which when added to `17` gives `595`. If `CODE` happens to be declared as a string then the number `595` is converted to the string `'595'`. The result may be far from what the programmer had in mind!

   PL/I was based on the philosophy : *If a programmer has written it, it must be correct*. It may be quite flattering to assume that programmers know what they are writing. However, this assumption is a standing invitation to danger since experience suggests that *programmers often do not know in precise details what they have in mind*. A language which assumes correctness rather than checking for it is committed to a path to peril.

3. Simplicity is impaired if a language provides several alternative ways of specifying the same concept.

   (a) Providing more than one form to denote a concept increases the size of the language and favours the development of "dialects", "users-groups", or "company-standards" that use only a subset of the alternatives allowed. A user expert in one dialect can have difficulties in reading the programs written in a different dialect.

   - COBOL provides a concise mathematical notation as also an English-like notation. So one may write

     ```
     MULTIPLY WORK_HOURS BY HOURLY_PAY GIVING DAILY_PAY
     ```

     or one may write

     ```
     DAILY_PAY = WORK_HOURS * HOURLY_PAY
     ```

   - C allows a user to add 1 to an integer variable `a` in one of the following four ways :

     (i) `a++;`

     (ii) `a = a + 1;`

     (iii) `a += 1;`

     (iv) `++a;`

   - C allows the `i`th element of an array `a` to be accessed by either `a[i]` or by `*(a + i)`.
   - Similarly, a component `x` of a structure `S` pointed at by pointer `p` can be retrieved either `p -> x` or by `(*p).x`.

   (b) As a special case of allowing for many alternative ways, a language may provide defaults for the most common[7] choices.

[7] Or so perceived by the designer!

- It is not necessary to declare variables and their types in a FORTRAN-77 program. All variables whose names begin with character `I` through `N` are `INTEGER` by default, all other variables are `REAL`.[8] A programmer may also specify types explicitly through a declaration and override the defaults.
  Confusion may arise if someone who follows the default declaration, were to read a program with explicit declarations (say a program in which a variable called `INTEREST` is declared to be `REAL`).
- PL/I has the following syntax for declarations

  $$\boxed{\texttt{DECLARE}\ \langle \textit{name} \rangle\ \langle \textit{base} \rangle\ \langle \textit{scale} \rangle\ \langle \textit{mode} \rangle\ \langle \textit{storage} \rangle\ \langle \textit{visibility} \rangle}$$

  Here the ⟨*base*⟩ could be `DECIMAL`, ⟨*scale*⟩ could be `FIXED` (i.e. not floating-point), ⟨*mode*⟩ could be `REAL (8,3)`, ⟨*storage*⟩ could be `STATIC` and ⟨*visibility*⟩ could be `EXTERNAL`.
  All of the above except the keyword `DECLARE` and ⟨*name*⟩ are optional with built-in defaults. This is anything but simple!

4. Simplicity is also impaired if the language allows different concepts to be expressed by the same notation. This is usually achieved through overloading.

   (a) Overloading of arithmetic operators is usually considered fair due to familiarity from school days as also due to the proliferation of operators that would otherwise take place.

   (b) When overloading is extended to user-defined operators and subprograms (as in Ada or C++), the intended meaning of the program may not be evident.

   (c) Sometimes even the well-established concepts may be allowed to be redefined. For instance in ALGOL 68, one may reverse the precedences of `+` and `*` if one prefers that the value of `(2*3+2*4)` should be `40` rather than `14`!

   Simplicity is best achieved through restriction of objectives, great care to readability and obviousness, and the basing of the language around a few well-defined simple concepts.

5 Definiteness

A language should be based on a definite philosophy, should serve a definite purpose and should be be defined as accurately as possible. We discuss the following four aspects of definiteness.

1. Definiteness of philosophy
2. Definiteness of purpose
3. Definiteness of documentation
4. Definiteness of implementation

5.1 Definiteness of Philosophy.

Many different philosophies of program design have evolved over the years. Some of them can mutually coexist leading to more effective programming while some others follow radically different approaches to computation. A language should subscribe to a definite philosophy (or to a collection of cooperative philosophies) rather than hoping to follow all of them. Some of the philosophies are

1. Structured Programming

[8] FORTRAN is a divine language since `GOD` is `REAL` in FORTRAN by default!

2. Stepwise Refinement
3. Encapsulation : Design around data invariants
4. Inheritance : Reusability with refinements
5. Computing with functions which enable equational reasoning
6. Computing with relations which enable deductive inferencing

These philosophies lead to different programming paradigms (discussed in section 7.2). The first two are used in *imperative paradigm*, third and fourth in *object oriented paradigm*, fifth in *functional paradigm* while sixth is used in *logic paradigm* of programming.[9]

A language ceases to be cohesive if does not restrict its philosophies. C++ is a case in point. It supports object oriented paradigm and tries to retain compatibility with C at the same time. As a consequence of trying to support two dissimilar approaches, C++ design has incorporated many adhoc and ill-conceived features. Smalltalk, on the other hand supports object oriented paradigm only hence it is a much cleaner and much easier-to-learn language.[10] Ada is another example of a not-so-cohesive language. It supports imperative paradigm, encapsulation, and concurrency (for real-time applications). By contrast, CSP (Communicating Sequential Processes) is designed for concurrency and seems to be a nice cohesive language.

5.2 Definiteness of Purpose

A language should serve a definite purpose rather than hoping to serve the diverse needs of all kinds of programmers. It has been amply proved time and again that the monstrous languages designed to "serve the entire humanity" rather than a specific purpose, are either useful in a limited domain or die their natural death. Ada is an example of the former while PL/I is an example of the latter.

The following languages are good examples of definiteness of purpose.

Language	Purpose
FORTRAN	Scientific Computing
COBOL	Business oriented data processing
Pascal	Teaching the fundamentals of programming
C	Systems programming

Note that definiteness of purpose does not constrain a language from being general. It merely puts restriction on it being *too* general. A language design serves no purpose if the language is not general enough to enable writing different kinds of programs for solving problems in a particular area.[11]

5.3 Definiteness of Documentation

Regardless of the experience in a language floating around, the documentation has the last word in case of conflicting interpretations of a feature. If there is any difference between the language as defined by the documentation and as discovered by a programmer through usage, the programs as understood by a one programmer may not be the same as the program as understood by another programmer which is perilous. There are two aspects of the definiteness of documentation :

1. A language definition should be precise, accurate, and complete. By completeness, we mean

[9] There also are some not so widely used philosophies viz. computing with recurrences, computing with constraints etc.

[10] The name "smalltalk" was invented since it was conceived as a language for children. It did serve its purpose very well.

[11] There often is a conflict between generality and simplicity. If one designs something which is both simple and general one has made an important contribution.

- completeness of definition.

  The original definition of Pascal did not specify the exact notion of type equivalence (see section 7.1.1) leading to avoidable confusion.
- completeness of documentation.

  Language documentation should be targeted at three categories of programmers : novices, somewhat experienced, and veterans.[12]

  – *Tutorials*. For the novices.
    They help new users to develop perspective quickly.
  – *Users' manuals*. For somewhat experienced programmers.
    They provide a detailed topic wise description. They are good for mastering the language.
  – *Reference manuals*. For veterans.
    They provide a (usually) alphabetic organisation to the definition for a quick search. They are extremely helpful when one is looking for a specific information.

2. A language definition should maintain internal as well external consistency. Internal consistency implies that a particular term used in the definition should mean the same throughout the documentation. External consistency means that every term used in the definition should refer to the same meaning that is attached to the term elsewhere in the programming parlance.

5.4 Definiteness of Implementation

In the final analysis, a language is only as good as its implementation. A language implementation should follow the principle of "least astonishment" whereby a language is implemented exactly as it is defined with neither any extensions nor any omissions — modifications are taboo. In any case, the implementation should not allow any undocumented feature; if there are extensions, they must be documented.

6 Orthogonality

6.1 What is Orthogonality?

Orthogonality implies that a language should be based on few simple features which can be combined to produce predictable results. It is important that there be no restrictions on the combinations, no exceptions to the general rule, and no "special cases". The implicit assumption is that once the primitives and ways of combining them are understood, a programmer can develop and understand a program in a modular way since "various pieces fit together" without any special cases. This philosophy dictates that program should have an inherent recursive structure where the rules that govern the combination should also govern the sub-structures which make up the combination. In other words, the combinations should be coherent and harmonious.

6.2 Orthogonality : Examples and Counter Examples

The following examples illustrate how the principle of orthogonality is adopted in practical languages as also how it is violated.

1. A *block* in a block structured language provides a local referencing environment and has the following structure.

$$\boxed{\langle block \rangle \rightarrow \langle sequence\_of\_declarations \rangle \ \langle sequence\_of\_statements \rangle}$$

[12] Overheard somewhere : "A language documentation should be targeted at three categories of programmers — *novices*, *novices*, and *novices*!"

where, declarations may declare other blocks apart from the local data items. In effect, blocks may be nested and the entire program may be looked upon as a single block which may contain other blocks. The rules that govern this block are the rules that govern the nested blocks too.

Pascal is a fully block structured language from this viewpoint in that the procedures/functions are independent named blocks.

C blocks are essentially compound statements with local declarations. Thus, C allows for un-named blocks *except* for the outermost blocks which are essentially functions in C. As a consequences, a block need not be explicitly activated — its declaration serves the purpose of activating it implicitly at the point where it is declared. *However* the outermost blocks (i.e. functions) have to be explicitly activated. Similarly, blocks can be nested in C *except* for the outermost blocks. Thus, a C program is *not* a single block but a sequence of blocks *though* C functions form independent single blocks. Note the use of words *except*, *however*, *not* and *though* in describing the C block structure which violates the principle of orthogonality.

2. C++ supports object oriented programming where a program is a collection of interacting objects. Each object encapsulates data and the operations allowed on that data. Each object is an instance of a class. However, a class is *not* an object — a class is a template using which objects are created. So there are entities in a program which are not objects. Also, an integer variable is not an object since there is no encapsulation of operations in one unit.

   On the other hand, a Smalltalk-80 program is a collection of objects. A class is also an object — an instance of a metaclass object which is used to create classes! This metaclass object is supplied by the language processor with an operation of creating a new class out of it. The primitive data types are also system defined classes (and hence objects). As a matter of fact, Smalltalk-80 is a complete programming system with a user interface for creating and executing programs, of which the language is only a small part. Every entity in the system (viz. the editor, the browser etc.) is an object which supports some methods (Smalltalk terminology for operation) which can be executed by sending a message (Smalltalk terminology for invoking an operation).

3. In C, assignment operation is viewed as a binary operation which can return a value as a result of the assignment operation. Effectively, it can be treated at par with binary arithmetic operations. Hence we can construct expressions which may use assignment as an operation. It follows that multiple assignments can be "combined" to create an expression which itself may be an assignment operation.

   Pascal on the other hand, treats assignment as a special operation by providing an assignment statement. Though assignment is fundamentally a binary operation (requiring an assignable data item and an assignable value), it is not treated at par with other binary operations.[13]

4. In C all subprograms can potentially return a value and as such are viewed as functions. Thus, subprograms may be invoked at a place where a value is required. Pascal distinguishes between function subprograms that return a value and procedure subprograms that cannot return a value. Effectively, some subprograms can be invoked at the expression level while some others can be invoked at the statement level only.

5. Some exceptions to the general rule in the context of parameter passing and returning result.

   - In C parameters of all types except arrays and files are passed by value. Arrays and files are passed by reference.
   - In C functions can only be passed through pointers.
   - In C functions cannot return arrays and files.

[13]For another view on assignment, see section 7.2.4.

- The following procedure definition is invalid in Pascal

```
procedure this ( var x :  array [1..10] of real; y char);
```

In order to indicate that x is an array, it should be specified by first defining an array type as follows.

```
type T = array [1..10] of real;
procedure this ( var x :  T ; y char);
```

6. The above mentioned violations are at least well documented and well understood. Violation of orthogonality in the context of exception handling facility provided by a language is a far more serious problem since it is less likely to be understood well. When it comes to exception handling, seemingly simple language features interact in a way that produces programs whose behaviour is hard to predict. An exception handler may produce different results if parameter passing is by value than when it is by reference in a language.

   The following key questions need to be answered for a good understanding of the likely impact of exception handling on the execution of the rest of the program.

   (a) What is the scope and visibility of an exception?
   (i.e. what all is visible to an exception routine and where is it callable from ?)
   Both C and PL/I do not allow the programmer to pass parameters to exception handling routines. Thus the necessary information exchange can only be established through global variables which is a very unsafe programming practice. Besides, the use of global variable may not provide complete information about the violation which caused the exception. For instance in C, if a particular signal is generated, the signal handling routine knows the signal but cannot know precisely who generated the signal.

   (b) What kind of "clean-up" actions are allowed by the exception handler?
   (Can it carry out a "destructive" clean-up on its own without the intervention of the user?)

   (c) Where does control flow after an exception is handled?

The lack of orthogonality can be annoying since a programmer cannot apply uniform generalisations and special cases/restrictions need to be kept in mind always.

6.3 Orthogonality and Simplicity : Friends or Foes?

In general, simplicity is enhanced by orthogonality. For instance violation of orthogonality in exception handling (section 6.2) affects simplicity adversely. However, if taken too far, the principle of orthogonality tends to conflict with simplicity. The rules of combination may be simple but there may be far too many combinations or far too many levels of nestings which may hinder clarity and understandability. The following two examples suffice to elaborate on this issue.

1. The C extreme ...

   We once again look at the code fragment from the file `lex.yy.c` generated by *lex* program which was considered in section 3.2.

```
# define input() (((yytchar=yysptr>yysbuf?U(*--yysptr):getc(yyin))
==10?(yylineno++,yytchar):yytchar)==EOF?0:yytchar)
```

In order to understand it, we rewrite the definition of input by adding spaces comments and newlines. (Note that we have not included a \ at the end of every line which is required if a macro definition is spread across many lines in C.)

```
# define input () /*  Get a new character                         */
((( yytchar = /*  in this variable                                */
     (yysptr > yysbuf)?  /*  Was a character pushed back?         */
     * -- yysptr :  /*  If yes, then it is the next one           */
     getc (yyin) /*  Else get one from input file                 */
  ) == 10 ?  /*  Is this character a newline?                     */
  (yylineno++, yytchar):  /*  If yes, increment line counter      */
  yytchar /*  If no, simply pass it along                         */
 ) == EOF ? /*  Was it an end of file?                            */
 0 :  /*  If yes, the parser expects a 0                          */
 yytchar /*  This is the next character                           */
) /*  End of the macro input                                     */
```

The complete body of the above macro is just an expression. However, many actions have been packed into this expression making it very complicated.

2. The Algol 68 extreme ...

```
( real x,y;              {  Declare two local variables.          }
  read((x,y));           {  Read two values in them from input.   }
  if x < y then a else b {  Choose one of them as the             }
  fi                     {  left hand side of the assignment.     }
 ) :=                    {  The right hand side consists of b and }
   b +                   {  a conditionally selected second       }
     if a:= a+1; a > b   {  operand.  If a > b, increment a,      }
     then                {  and the second operand is             }
       c:=c+1; +b        {  +b, increment c in the meanwhile      }
     else                {  If a is not > b, then the second      }
       c:=c-1; a         {  operand is a, decrement c.            }
     fi                  {  End of selection of the second operand }
```

This is a still more complicated example in which even the left hand side of an assignment can be selected conditionally. Further a value in an expression can be selected by using the control construct `if ... then ... else` and the condition that is tested may have an assignment operation in it! This example highlights the pitfalls of allowing all possible combinations.

The tradeoff between simplicity and orthogonality is hard to quantify. What is clear is that a good language must possess *both* the "virtues". What is not clear is the extent to which orthogonality should be supported. Opinions may differ on the limit after which orthogonality ceases to be a virtue and becomes a vice!

7 Expressiveness

Machine language programs are hard to understand due to the low expressive power of machine instructions. Expressiveness of a language is a measure of how naturally a problem solving strategy can be mapped into a program. In other words, expressiveness relates a program to what the programmer has in mind. Thus a language is expressive if it provides features which are similar to the ones employed by a programmer in his/her mind while devising a solution to a problem.

While devising a solution mentally, we do not in general think of it as a monolithic task. We decompose the task into subtasks which may further be decomposed if necessary. This decomposition is never arbitrary — we try to find subtasks which are relatively independent. The more the independence, the cleaner our decomposition is, since we can concentrate on the details of one task at a time and later think only about how the tasks relate to each other.

In essence we employ *modularisation* and *abstraction* as two fundamental tools for organising our tasks. A closer look reveals that they are the two sides of the same coin : Modularisation involves decomposition into smaller pieces while abstraction involves combining the pieces back concentrating on just those properties which are essential to the purpose; irrelevant details are safely ignored by hiding them.

A good programming language should not only facilitate, but explicitly support, modularisation and abstraction in programs. This can be achieved by providing features which help programmer create abstract models of data and operations. The closer these models are to his/her mental image of the program, the more reliable the program is and the more productive the programmer is.

7.1 Language Support for Abstraction

A good programming language should support the following abstractions for enabling a programmer to create abstract models and express them naturally.

1. *Abstraction of data*

   We consider the following aspects of data abstraction :

   (a) Data types

   (b) Semantics of data types

   (c) Storage allocation for data

2. *Abstraction of operations*

   Operations may be abstracted at many different levels. We discuss the following

   (a) Expressions

   (b) Statements

   (c) Subprograms

3. *Abstraction of scope/environment*

   (a) Blocks

   (b) Subprograms

A detailed treatment of the above is beyond the scope of this chapter, we look at some specific issues in the following sections.

7.1.1 Abstraction of Data

In order to enhance expressiveness, a language design must consider the following semantic issues raised by data abstraction. It is important to take unequivocal policy decisions on these issues and state them unambiguously in the documentation. We merely list the issues here rather them explaining them in detail.

1. *Data Types*

   Languages provide the facility of dividing data items into various types in order to restrict the operations that can be performed on the data. For instance, adding two characters is meaningless unless addition when applied to characters, implies concatenation. Concatenation of two numbers, on the other hand, is numerically meaningless.

   The data types included in a language depend on the language designers visualisation of the kind of programs that would be written in the language. While this may vary from language to language, a good programming language should provide :

   - A set of basic data types.
   - A set of basic data structures (i.e. a set of constructors which can be used to create derived or structured data types.)
   - Facilities for naming user defined data types.

   The last point above makes sense only for the languages where a programmer is required to declare the types of variables (refer to the discussion on semantics of data types.)

2. *Semantic of data types*

   (a) Is the language typeless?

   Typelessness does not mean that variables do not have types. It means that :

   - Types need not be declared.
   - Types may change dynamically.

   It should be noted that if types can change dynamically, it does not mean that type compatibility, and hence type checking, is not required. For instance, `A` and `B` must have the same types in the expression `A + B`. They may have other (dissimilar) types for other computations.

   (b) When can two types be said to be equivalent?

   Initial version of Pascal definition did not specify if the language implied *name equivalence* or *structural equivalence* leading to non-portable programs.

   Of the two, structural equivalence is more general and hence a little less safer.

   (c) When are types checked?

   When types are checked statically (i.e. before execution), type errors are detected early and more exhaustively. Besides, there is less work during execution leading to more efficient programs. Dynamic type checking can check types only along the execution paths taken during a particular execution. Thus, some errors may remain undetected. Besides, it requires extra space during runtime to store the type information.

   (d) Is the language strongly typed?

   A language is strongly typed if a program can be guaranteed to run without type errors during execution. Thus strongly typed languages require static type checking, and hence lead to safer and more efficient programs. However, not all type checking can be performed statically, and practical languages only try to approach strong type checking ignoring the checks which cannot be performed statically.

   (e) Does a language require types to be declared? If yes, should the types of *all* data items need to be declared?

   Declarations provide useful redundant information for type checking and aid static type checking. In absence of declarations (or default types as in FORTRAN), the language processor has to infer types which is a price to be paid in terms of inefficiency — if type analysis is done statically, then the translation is inefficient; if it is done dynamically, then the execution is inefficient.

In general, a program with explicitly declared types is likely to be more expressive. Exception to the rule are generic data types viz. lists, queues, stacks etc. where some operations can be naturally visualised without the need of knowing the type of an element in the data structure.

(f) Does the language allow overloading/polymorphism?

As noted in section 4, overloading may enhance expressiveness if it is restricted to familiar cases. General facility of overloading may obscure the meanings in a program. Polymorphism also enhances expressiveness provided the application of a function/operation is restricted to a specific generic type. On the flip side of the coin, the language processor may have to perform type inferencing and hence may end up paying the price of inefficiency.

(g) What operations are allowed on structured data types?

Typically, languages provide operations for selecting components of structured data types. Operations on a data type as a *whole* may enhance expressiveness but implementability may suffer adversely.

3. *Storage allocation for data*

We consider two categories of data items :

(a) Statically specified data items.

By statically specified data items, we imply the variables whose storage requirements can be deduced before execution viz. an integer variable.

(b) Dynamically created data items.

These are the data items whose storage requirements may vary during execution. Generic types viz. lists, stacks etc. are examples of dynamically created data items.

A programmer should have the freedom of creating data items dynamically. Whether this is achieved by providing pointers and explicit requests for memory allocation, or whether the programmer may just introduce new data items and the language processor manages the allocation and deallocation is a matter of policy decision. The latter approach, which is employed by functional languages, is safer than the former approach primarily because the language processor can keep track of the allocated storage much easier and more reliably than a programmer can. Besides, this relieves the programmer of the burden by providing a higher level of abstraction thereby enhancing the expressiveness of programs. On the flip side of the coin, by entrusting the language processor to perform this task, the implementability as a virtue of the programming language suffers adversely.

7.1.2 Abstraction of Operations

A programmer should be allowed to specify abstract operations at the following three levels.

1. Expressions

A language should provide logical, relational, and arithmetic operators for combining various subexpressions to specify a complex expressions. The following points should be noted in this regards :

(a) There should be as few levels of precedences as possible. Typically, the precedences of arithmetic operations are well understood. So one may use them without the use of parenthesis. Operators for the selection of components of structured data types should have the highest precedence.

C provides 15 (fifteen !) levels of precedences allowing extremely cryptic expressions like the following

```
a = b [c=d?e:f || g &&h | i^j==k < l>>m+n*o++].p
```

This is a legal expression acceptable by C compilers. What does it compute, however?

Providing fewer levels of precedence and forcing a programmer to use parenthesis explicitly to indicate precedences seems perfectly reasonable for sake of clarity.

(b) Operators that are used to combine the subexpressions should be free from sides effects i.e. they should not modify the value of their arguments in any way for if they do, the exact computation specified by an expression becomes hard to understand. The following operators provided by C have side effects :

```
= += -= *= /= %= &= ^= |= <<= >>= ++ --
```

The two instances of `i` and the two instances of `j` in the following expression do not have the same value. What exactly is the result of the expression?

```
a[i] = b * i += ++j - j
```

(c) As a natural extension of the previous observation, if subprograms calls may be made where values are expected (i.e. functions in C, Pascal, FORTRAN etc.), they should not be allowed to have side effects.

2. Statements

   A language should provide three categories of statements to enable *structured programming* : sequence, selection, and iteration. It should be emphasised that a sequence does not merely mean a list of statements to be executed one after the other. It implies being able to treat the list of statements as one single statement with no control transfer in the middle. In other words, it should be possible to *group* multiple statements into one compound statement which acts as a single statement. FORTRAN-77 and BASIC, for instance do not provide a compound statement.

   In principle, structured programming advocates the use of single-entry single-exit control structures though iterative loops with conditional premature exits are considered acceptable in practice. The emphasis of structured programming is on writing programs without the use of `goto` statements. A `goto` makes a program very hard to understand since it allows absolutely arbitrary control flow. In some situations, however, the use of `goto` is cleaner. For instance, conditional premature exits from a deeply nested loop may be very awkward unless `goto` is used.

3. Subprograms

   Subprograms help a programmer extend the language by defining his/her own operations and as such provide an excellent means of modularisation and abstraction. While supporting subprograms a language should allow separate translation of subprograms. However, the language should insist on a declaration of the function prototype for static type checking.

7.1.3 Abstraction of Scope/Environment

If a particular data item can be manipulated anywhere in a program (i.e. it is visible globally), it is said to have a global scope. In case of errors, it is very hard to identify operations which modify the global data item erroneously. Thus reasoning about the correctness of programs becomes very hard if the data items cannot be protected against erroneous modification. In order to be able to predict a reasonable behaviour of the programs, it should be possible to restrict either visibility or the modifiability of data items to small regions of programs. This can be done at two levels :

1. Blocks.

   Blocks provide a local referencing environment (see section 6.2) and restrict the visibility of locally declared names to the operations within the block. They however, cannot restrict the visibility of the non-local names unless the names are redeclared in the block in which case, the inner instance of the name hides the outer instance.

2. Subprograms.

   In general, subprograms are named blocks which can be called explicitly with parameters. Parameters help to establish a controlled visibility which can vary from call to call. Besides, the modifiability of parameters can also be controlled if they are passed by *value* rather than by *reference*. When parameters are passed by reference, the subprograms may modify parameters and produce side effects affecting the clarity and understandability of programs.[14]

7.2 Programming Paradigms and Expressiveness

7.2.1 Declarative Abstractions Are More Expressive

As observed in section 7.1.2, operations in a program may be specified at three levels : expressions, statements, and subprograms. Specifying operations at the expression level and at the subprogram level provides good abstractions. In the case of expressions, the programmer merely specifies *what* needs to be computed in terms of various subexpressions — *how* these subexpressions are computed and *how* their values are combined is for the language processor to decide. When a programmer uses a subprogram to represent an operation, he/she can concentrate on *what* the subprogram does and separate it from the task of specifying *how* the desired computation is to be performed.

When operations are provided at statement level, each individual statement may have a small well defined operation. However, the programmer is interested in the combined effect of a group of statements which collectively represents a particular action. In such a case, it is the programmers' responsibility to provide individual statements and the order in which they are to be carried out so that the combined effect gives the desired result. Thus a programmer specifies not just what the group of statements does but also how it is achieved. Forcing the programmer to specify step-by-step procedure of carrying out a certain task is what earns the name *procedural languages* for the languages which rely heavily on collection of individual statements. The abstractions provided for expressions and subprogram calls are *declarative* in nature while the abstractions for statements and subprogram definitions are *procedural* or *imperative* in nature.

In case of the declarative abstractions, the effects of two operations is combined by the language processor, and it is this combination that is specified (or *declared*) by the programmer. In the case of imperative abstractions, the effect of two operations is combined indirectly through the values of data items. It is the responsibility of the programmer to ensure that the operations store correct values in the data items. It is easy to conclude that declarative abstractions are more expressive then the imperative abstractions though their expressiveness may not be universally appealing.

7.2.2 Declarative Programming

Declarative programming paradigm relies heavily on the declarative abstractions. This has traditionally been done in the following three ways.

1. *Functional programming*

   Functional languages use the mathematical notion of *function* as abstraction for operations which is a more general abstraction than the abstraction of *expressions*[15] used by imperative languages. This is not to suggest that functions are not allowed in imperative languages. (See section 7.2.4.)

2. *Logic programming*

[14] The other two methods of parameter passing (i.e. by *value-result* and by *name*) are of historical interest only.

[15] Expressions are also essentially function computations where the operators are functions and variables appearing in the expressions are arguments to the function operators. However, expressions restrict us to the functions defined by the language.

Logic programming uses the mathematical notion of *relations* which is a more general then functions.[16] In particular, data is viewed as *relations* and operations as creating new relations by using a set of *logical deduction* rules.

3. *Domain Specific programming*

   These languages are essentially high level specification languages for tasks in a specific domain. A report generator is the most common example of domain specific languages. *Lex* and *yacc* can also be viewed as domain specific languages for generating lexical and syntax analysers, respectively.

7.2.3 Object Oriented Programming

Object oriented programming uses the notion of an object to *encapsulate* data and the operations allowed on it, in one unit thereby leading to a more disciplined modifiability of data. Thus a data in an object can only be manipulated by the operations supported by the object. The advantage is that inconsistent modification of (structured) data items is prohibited and the invariant relationship between different members of structured data is maintained. Another significant abstraction employed by object oriented programming is the notion of *inheritance* which allows reusability with refinements.

7.2.4 A Comparison of Functional and Imperative Languages

Functional languages differ from imperative languages in the following ways.

1. The basic unit of computation in functional languages is a *function* i.e. the basic operation is a function evaluation. In imperative languages, the basic operation is an *assignment* of a value to a variable.

2. Functions are *first class* values in functional languages i.e. functions are treated at par with data. Thus functions can be passed as parameters, returned as results, created dynamically, and stored in data structures. Imperative languages, on the other hand, distinguish between operations and data.[17] During execution of a program, an operation specified in the program cannot be data for the program. By the same token, data specified in a program cannot serve as an operation for the program. Further, operations in imperative languages are static in that new operations cannot be created at runtime.

An assignment implies storing a value in a memory location and since this can be done many times over, values of variables are modifiable in imperative languages. By contrast, values computed by functional programs are results of functions and as such are non-modifiable. A pleasant consequence of this notion of non-modifiability is that the functions in functional languages are free of side-effects.[18]

Since functions are first class values in functional languages, they can be combined in many different ways viz. apart from invoking a function, it can be passed as a parameter to some other function to create either a more general or a more specific function. Alternatively, a function need not be specified in the program, it may be be computed as result of some other function. This ability of combining functions in many different ways has a significant implication — functional languages require very few basic operations as part of the language. Also, since very powerful operations can be defined very simply, few generic data structures suffice for a variety of purposes. By contrast, imperative languages need a larger set of basic operations and a larger set of data structures.

[16] Mathematically, functions are also relations though not vice-versa. In the case of functions, we have a domain and a range whereas in the case of relations one does not distinguish between between domain and range.

[17] This distinction was re-enforced by Algol hence it is often called *The Great Algol Wall*! The functional languages, however, have not really destroyed the Algol wall — Even though functions can be treated as data, data cannot be treated as functions. Object oriented languages come close to destroying the wall since both data and operations are part and parcel of one single unit (called and an object) and an operation of an object is invoked much in the same way as a data item is read for its value. However, the wall does not vanish altogether, it only becomes transparent while using the object. It very much exists in the definition of the object.

[18] In other words, a side-effect of non-modifiability is the absence of side-effects!

Since functions can be combined in in many different ways, the effects of various functions can be combined directly. Hence functional languages are essentially declarative. This must be contrasted with imperative languages in which the effects of various operations are combined indirectly by writing into a memory location and then reading from it.

Since functional languages are more expressive, writing programs in functional languages take much less time and effort. Besides, it is much more easier to get programs to work correctly in functional languages than in imperative languages. Thus functional languages help in rapid prototyping.

The main drawback of functional languages has been the lack of efficiency of execution. In order to support the flexibility that is required for the above advantages, functional languages have traditionally been interpreted rather than compiled. Historically, efficiency has not been an important concern in functional languages. This has started changing lately and the recent implementations have started providing the option of compiling also. The expectation of reasonably efficient programs in functional languages is not as unrealistic as it was before.

8 Suggestions for Further Reading and Bibliographic Notes

Programming language design has always been an intriguing area — there have been more failures than successes. For a few success stories (2) and (3) are excellent sources of (usually) the original creators' recollections from the past. These books are outcomes of the two ACM-SIGPLAN Conferences on *History of Programming Languages*. Apart from the creaters' position papers, they also contain the transcripts of the discussions and the question answer sessions held in the conferences. It is always easy to criticise an effort, it is very difficult to visualise the pioneers' insights in the perspective of the circumstances which led to the omissions or inclusions of good/bad features. These two volumes are a must for such a "mind reading".

A useful but still reasonably old discussion of programming language design is in form of a wonderful chapter in (11). They provide a detailed evaluation of ADA with respect to the proposed criteria. On the analysis side, (4) is an outstanding book. The authors go far deep into the time machine and do a splendid job of playing the "reel back" in the new light of understanding. They propose a (somewhat artificial) model called ISM (Ideal Software Machine) which can be used for describing all languages. An excellent discussion of programming language concepts can be found in a slightly oldish book (5). The concepts have been presented with a remarkable accuracy, precision, and coherence. (9) has traditionally been a rich source of analysis. He provides insights into implementation too. However the current edition is rather bulky; it needs to be thoroughly reorganised since the reader gets lost into the details very quickly. (8) is another book dedicated to analysis of programming language concepts. There are very many books on the line of (8), we do not include them in the bibliography.

The classical source of ideas on implementation has been (1) though the book is restricted to imperative paradigm. Fundamental ideas, however, can still be used for other paradigms. The classical source for implementation of functional languages has been (6).

(10) is a very interesting book for understanding the basics of different paradigms. He does an excellent job of putting all paradigms on a common platform. A must for appreciating the power as also the limitations of different paradigms.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] R. L. Wexleblat (ed.). *History of Programming Languages*. Academic Press, 1981. ACM Monograph Series.

[3] R. L. Wexleblat (ed.). *History of Programming Languages-II*. Academic Press, 1994. ACM Monograph Series.

[4] D. Gelernter and S. Jagannathan. *Programming Linguistics*. MIT Press, 1990.

[5] C. Ghezzi and M. Jazayeri. *Programming Language Concepts*. John Wiley & Sons, 1982.

[6] S. L. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall Inc., 1987.

[7] Wm Leler. *Constraint Programming Languages : Their Specification and Generation*. Addison-Wesley Publishing Company, 1988.

[8] M. Marcotty and H. F. Ledgard. *Programming Language Landscape, 2/e*. Science Research Associates, 1987.

[9] T. W. Pratt and M. V. Zelcowitz. *Programming Laguages – Design and Implementation 3/e*. Prentice-Hall Inc., 1996.

[10] R. Sethi. *Programming Languages – Concepts and Constructs*. Addison-Wesley Publishing Company, 1996.

[11] J. P. Tremblay and P. G. Sorenson. *The Theory and Practice of Compiler Writing*. McGraw-Hill Book Company, 1986.