# Josef, Programming for Everybody

Ivan Tomek
Acadia University
Wolfville, Nova Scotia, Canada

## Abstract

The author believes that everybody should be introduced to programming but that standard general purpose languages are not suitable for this purpose, mainly because they do not provide an environment offering natural problems. Another characteristic which makes them unsuitable for the purpose is that they are all more or less burdened by restrictions imposed by legitimate concerns of professional programmers with security and economical aspects of programming. This paper briefly considers the general features that a programming language intended for the introduction of an average non-programmer should have and describes some aspects of one such language developed by the author.

## The Need for a Special Programming Language

Most children now play electronic games and are progressing from games towards computers. Today's five year olds will not only be required to learn about computers when they reach the middle of their compulsory education but, if the atmosphere is conducive, will be eager to learn about computers on their own initiative. Are we prepared to offer them a programming environment which will satisfy their interest?

The question is not only whether we can make programming easier but also, and foremost, whether we can make the reward for writing a program sufficiently attractive to make the effort worthwhile. My opinion is that traditional programming languages fail to offer a stimulating environment. The visible reward that they offer is minimal - why would an average person bother to learn what is a real array just to be able to write a program which sorts a sequence of numbers or a similar routine and rarely encountered problem of little general interest? On the other hand, to write a program in one of these languages, which does something that an average person finds interesting requires an inordinate effort. There is, of course, a hidden reward in all programming, particularly if you are new to it, the intellectual satisfaction derived from having solved a difficult problem, but before this reward can be felt there must be a first stimulus to arouse the potential programmer's interest in programming. If the first stimulus is missing the student will not even start thinking about the task. And this necessary stimulation is lacking in Pascal, BASIC, or any other general purpose programming language. They do not easily lead to the kind of problems that an average person finds interesting.

In addition to the lack of motivation, general purpose programming languages make the life of a beginning programmer unnecessarily difficult by rules imposed on them by the justifiable concerns of professional programmers (usually attributable to the concern for security and correctness) and their willingness to accept various restrictions due to historical machine limitations, speed, limited capacity of memory, etc. All these influences are reflected in strict rules requiring declaration of all variables, restrictions on the length of variables, limited flexibility of use of built-in data types, etc. These strictly enforced restrictions, perfectly in order in professional environments but essentially unjustified in "leisure programming", only add to the boring environment that such languages appear to present to an ordinary novice programmer. The result is the rarely admitted fact that for most people programming is unattractive and difficult.

Must programming be unattractive and difficult? Programming is a form of problem solving and most people are happiest when they can solve problems - play games, read detective stories, spend hours rotating Rubik's cube, etc. If we could recast programming into a game which produces quick satisfaction and intrigues even the novice, it would gain instant popularity and, with proper attention, develop into a pastime at least as attractive as any other intellectually stimulating game. My premise is that we want to achieve such state of affairs and I will describe my approach to achieving this goal.

The first condition for programming to become attractive is the existence of a programming language and environment which offers a stimulating set of realistic, understandable, and natural problems and allows an average person to solve them and to obtain a perceptually gratifying response from the computer. The obvious answer to the search for such an environment is to create a language which allows the programmer to do something interesting, preferably dynamic, with the screen. This is, of course, an old proposition which has been successfully applied in particular in the Logo Turtle (1). The Turtle essentially allows the programmer to use the screen as a draw-

ing board and drawing is one of the most natural human activities. There are a number of similar natural environments which can exploit the screen of the computer terminal and the next section describes one of them.

The second condition for programming to become attractive has to do with the philosophy of the language. A toy language designed for fun or introductory programming rather than for professional "production environments" does not have to be built on the same principles as standard general purpose programming languages. The purpose of such a toy language is to facilitate the communication between the user and the computer. The most natural communication does not have to be concerned with security, speed (as long as it remains within reasonable limits), guaranteed and rigorously enforced non-ambiguity and similar criteria which are essential for programs controlling large amounts of money, dangerous weapons, etc. Our ordinary communication is a somewhat inaccurate and redundant process in which there is usually room to correct misunderstandings before they cause a disaster, in which faulty formulations can be rectified, and strategies reformulated. Similar considerations, within the limits of our present technical abilities, should apply to toy programming languages.

Much of the above applies in varying degrees to the Logo language and Karel (2), the language developed at Stanford University to teach Pascal.

The following section describes another system which attempts to fulfill the needs outlined above.

## Josef

Josef is a robot simulated on the screen of an ordinary computer terminal. The screen is a map of Josef's world and he can execute commands referring to it. He has a small built-in vocabulary, somewhat like the Turtle, which the user can extend by programming new words. The main differences between Josef and the two languages to which it is related (Turtle and Karel) can be summarized as follows:

1. In comparison with the Turtle, Josef's language more closely resembles orginary programming languages. It is quite rich in structures and operations. This possibly makes the language somewhat more powerful and suitable for an introduction to "serious programming". In comparison with Karel, Josef is (much like the Turtle), relatively free in syntax and not tied to any existing programming language.

2. Unlike Turtle and Karel, Josef allows truly top-down problem development. This makes the language suitable for natural formulation of solutions and easier to understand.

3. Unlike the Turtle, Josef works on an ordinary CRT terminal. This is because Josef's world is essentially a discrete orthogonal world while Turtle's world is basically continuous.

4. In comparison with Karel, Josef'slanguage is quite rich. Its purpose is not to introduce the user to some other programming language but to provide a sufficiently rich environment to keep the user interested. Josef is a goal in itself, not a prelude to something else.

5. Josef's world is similar to the world of an ordinary human. It is not an oversimplified geometrical abstraction. At the same time it is flexible enough so that it can be treated as a simple geometrical system as well.

The above statements can best be demonstrated by a simple example. Assume that Josef's map is as in Figure 1 (the user can define any number of his own maps if he wants to) and that we want to teach him a new command (declare a program) to go around the block Sherwood, Main, Orchard, and Pleasant, collect all coins, mark locations with coins by letter C, return to his initial location, and report whether he found any coins. (This program, which is probably too difficult for a complete novice, is given here to demonstrate the environment in a nutshell.)

The program could be

```
NEW BLOCK
BEGIN
 MAIN  -- Go to Main Street collecting coins
 LEFT  -- Turn left
 ORCHARD  -- Go to Orchard Avenue collecting coins
 LEFT
 PLEASANT
 LEFT
 SHERWOOD
 REPORT
END
```

None of the words used in this commented program, with the exception of LEFT, are in Josef's basic vocabulary and the system will automatically prompt us to declare them. (This is the promised facility of top-down development.) We could have

```
NEW MAIN
BEGIN
 DO 13 TIMES MOVE_GET
END
```

or, with less effort and more generality,

```
NEW MAIN
BEGIN
 REPEAT MOVE_GET
 UNTIL  BLOCKED
 -- Here we are taking advantage of the fact that
 -- on this map Josef cannot move off the street
END
```

MOVE_GET is not in Josef's vocabulary either. It could be

```
NEW MOVE_GET
BEGIN
 MOVE
 IF   SEE('PENNY')  THEN  GET_MARK('PENNY')
    -- Objects are essentially string constants
 IF SEE('NICKEL') THEN GET_MARK(NICKEL')
 etc.
END
```

Command GET-MARK could be

```
NEW GET_MARK(COIN) -- Note the user of parameter
                   -- COIN
BEGIN
 GET(COIN)-- GET is in Josef's basic vocabulary
 MARK('C')-- MARK is also a built-in word
END
```

As the last example, let us declare command REPORT used in the "main program":

```
NEW REPORT
BEGIN
 IF HAVE('PENNY') THEN SAY('I found a penny')
 IF HAVE('NICKEL') THEN SAY('I found a nickel')
 etc.
END
```

The word HAVE is a built-in command. To command Josef to execute the complete program we just type its name. Typing a word from Josef's current vocabulary makes him execute the corresponding program. BLOCK and all the other declared words can be made part of Josef's permanent vocabulary.

Josef can do a number of other things. For several examples of built-in words and a few sample programs which give a general idea, read the appendices. In addition to the listed special words Josef has a full slate of control statements, arithmetic, variables, and procedures and functions with parameters (as partially demonstrated above).

It is my experience from a brief use of the language in teaching computer literacy to high school teachers and non computer science university students that the language provides a natural environment which most people understand very easily and find attractive and stimulating, particularly because of the immediate response of the screen to their commands. The language is designed and implemented to be relatively free of syntax restrictions to satisfy the second requirement formulated in the previous section. Two examples of this aspect of Josef are the demonstrated ease of top-down programming and the free use of "overloading" of identifiers (the use of the same word in several different meanings without ambiguity), a feature strongly discouraged in serious programming languages but very natural in ordinary communication. As an example, MARK can be used in two meanings:

```
MARK('C')
```

means

Mark the current location with letter C

but MARK without parameter is a character function which returns the character with which Josef's current location is marked.

The use of overloading allows us to restrict the size of the vocabulary of the language which makes it easier to learn and use it, without introducing ambiguity. Another example of the liberation of the language from traditional restrictions is that the same identifier can be used to hold different data types in the same program or even to allow the same data to be interpreted differently in different parts of the program. This is again a very natural aspect of human communication strictly forbidden in most "serious" languages. This feature, unacceptable in the production environment, appears to be very natural and desirable in a toy language as it was argued before. This is particulary true if restrictions may be imposed if the programmer wishes.

## Concluding Remarks

Josef is growing in stages. The first module (3) has been about 60 percent completed on a Cyber 171 at Acadia and used in two courses. It is now being rewritten to reflect the changing specifications and converted into a microcomputer implementation which is expected to become available in early sping 1982. This module contains most of the essential features of standard programming languages with the notable exception of data structures. They and other features will be implemented in the second module which is largely at the stage of conceptual development.

In its entirety the language should include most concepts found in modern programming languages, plus the means to easily perform toy-like functions on the screen as indicated in the above example. It is my belief that this language should be appropriate for a natural, enjoyable, stimulating, and yet serious introduction to programming, suitable for use at almost any scholastic level - possibly after modifications resulting from field experience. It appears that the language, which allows very easy manipulation of the screen, could be even useful for certain specialized functions in serious programming, particularly in programs designed for the manipulation of screens.

## Appendix 1. Some Words From Josef's Built-in Vocabulary

BLOCKED Boolean function which returns TRUE when Josef cannot move.
comment, everything on line following two minus signs --.
CORNER Boolean function which returns TRUE if Josef's current location is a meeting point of two different streets.
DIRECTION character function which returns U, D, R, L depending on Josef's current direction.
ERASE replaces mark placed by Josef by the original map symbol.
GET makes Josef get the object whose name is given as text parameter. In other words, transition of the specified object from SEE to HAVE.
HAVE Boolean function which returns TRUE if Josef has the object specified as the text parameter. Complement of SEE.
LEAVE makes Josef leave the object specified as text parameter. The inverse of GET, performs transition from HAVE to SEE.
LEFT makes Josef turn left by 90 degrees.
LISTEN makes Josef read input displayed in the communication area and save it in the variable given as parameter. Multiple parameters.
LOCATION returns name and number of Josef's current location in its two parameters.
MAP requests transition to a map stored in a file whose name is the parameter of the command.

MARK makes Josef mark his present location with the character given as its parameter. In case of longer parameters only the first character is used.

Can also be used as character function with no parameter. It then returns the mark character of the current location, the null character '' if not marked by Josef.

MOVE makes Josef move to the next location if such a move is possible in the direction in which he is presently facing.

PAUSE when used with parameter makes Josef pause for the specified number of internal units of time before proceeding to the next command. When used without parameter, the command makes Josef pause until the user types GO. In the parameterless mode the user may type any number of commands which Josef will execute before returning to the interrupted program by GO.

RIGHT makes Josef turn right by 90 degrees.

SAY makes Josef display the specified text parameter(s) in the communication area. Parameters are expressions.

SEE Boolean function which returns TRUE if the object specified as its text parameter is in Josef's current locaion but not in his possession.

SPEED is a command whose single parameter specifies the speed at which Josef executes MOVEs. The value must be an integer number between one (slow motion) and ten (fast motion). When used without parameter SPEED is a numerical function which returns Josef's current speed.

## Appendix 2. A Few Sample Programs

A program to move Josef to the end of the street by using recursion:

```
NEW TO_END
BEGIN
 IF NOT BLOCKED THEN
  BEGIN
  MOVE  TO_END
  END
END
```

A program to make Josef go to a location marked 'X' or the end of the street and return to the initial location:

```
NEW GO_RETURN
BEGIN
 MARK('*')  -- Mark initial point
 WHILE NOT  (BLOCKED OR MARK='X') DO MOVE
 AROUND  -- Turn around, must be declared
 -- Now return to the marked initial location
 WHILE NOT MARK='*' DO MOVE
 -- Finally clean up and assume initial position
 ERASE
 AROUND
END
```

A variation on the same problem:

```
NEW GO-COUNT
BEGIN
 COUNT:=Ø FOUND:=FALSE
 WHILE NOT (BLOCKED OR MARK='X') DO
 BEGIN
 MOVE
 COUNT:=COUNT+1
 IF MARK='X' THEN FOUND:=TRUE
 END
AROUND
DO COUNT TIMES MOVE  -- Josef counted steps
AROUND
IF FOUND THEN SAY ('I found the mark')
         ELSE SAY('I have not found the mark')
END
```

Find the bug in this program and correct

A useful function to get the required answer:

```
NEW ANSWER(A1,A2)
BEGIN
 REPEAT
  BEGIN
  SAY('Please answer ',A1,' or ',A2)
  LISTEN(RESPONSE)
  END
 UNTIL (RESPONSE=A1) OR (RESPONSE=A2)
 RETURN(RESPONSE)
END
```

A possible use for this new word is

```
IF ANSWER('LEFT','RIGHT')='LEFT' THEN LEFT
                                 ELSE RIGHT
```

or

```
IF ANSWER('*','+')='*' THEN MARK('*')
                       ELSE MARK('+')
```

As a final example, the following procedure with parameters will move Josef to the desired address on the present street if the address is there, or return him to the initial position if it is not:

```
NEW GO_TO(NAME, NUMBER)
BEGIN
 COUNT:=Ø  FOUND:=FALSE  DONE:=FALSE
 WHILE NOT DONE DO
  BEGIN
  LOCATION(NAM, NUM)
  IF NAM=NAME AND NUM=NUMBER THEN
   BEGIN DONE:=TRUE FOUND:=TRUE
   END
  DONE:=BLOCKED
  IF NOT DONE THEN
   BEGIN
   MOVE  COUNT:=COUNT+1
   END
  END
 IF NOT FOUND THEN
  BEGIN
  AROUND
  BACK(COUNT)  -- Assuming that we declared word
               -- BACK to return
               -- Josef COUNT steps
  AROUND
  END
END
```

A possible use of this new word is

```
GO TO('Main',17)
```

References


(1) Seymour Papert; Mindstorms, Children, Computers, and Powerful Ideas, Basic Books, Inc., 1980.

(2) Richard Pattis; Karel the Robot, Wiley, 1981.

(3) Ivan Tomek; The First Book of Josef, to be published by Prentice-Hall.


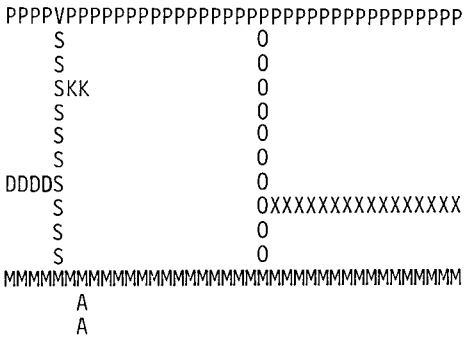


Figure 1. A sample map for Josef. Individual letters represent locations that Josef can access. In this example, S is Sherwood Drive, M is Main Street, O is Orchard Drive, P is Pleasant Street. The other streets are not important. Josef is shown as V (to indicate his current orientation which is down) at the top of Sherwood Drive, positioned to go down. Command MOVE makes him go to the next location, if he can, LEFT and RIGHT change his orientation.