TPD: An Educational Programming Language Based on Turkish Syntax

Sercan Tutar, Cem Bozşahin and Halit Oğuztüzün

Middle East Technical University
Department of Computer Engineering
06531 Ankara, TURKEY
{sercan, bozsahin, oguztuzn}@ceng.metu.edu.tr

Abstract. TPD is an experimental programming language the syntax of which is inspired by that of Turkish. It supports both imperative (procedural) and functional paradigms. General lists are provided as a built-in data type. TPD's intended users are high school students and novice programmers who are native speakers of Turkish. Our aim is to leverage their native language competence for learning the essentials of programming. Given the educational concerns, the design of the programming language goes hand in hand with the design of the development environment. Diagnostic features of the compiler are emphasized. Generated target code is in Java. The development environment also includes a language-based editor.

1 Introduction

In most traditional programming languages, the sequencing of symbols is inspired by mathematical conventions at the expression level and syntax of English at the statement level. Simply adopting some words from natural languages as symbols of a programming language may not be enough to ensure comprehensibility for the novice programmer. The effects of employing the structural similarity between programming languages and natural languages on learning the programming language and on prevention of errors is an area of intense research [1, 2].

The goal of TPD is to give a head start in programming to Turkish students. The syntax thus attempts to mimic Turkish syntax with the hope that statements of the language would be more comprehensible to a non-programmer who relies on her native language to express her actions to perform. Thus, TPD is expected to make the process of learning the basics of programming easier by allowing the users to make an analogy between their knowledge of Turkish and the source code of the programming language. Without having to bother with learning the intricacies of English syntax and word order at an early stage, the students are expected to move on to industrial-strength languages once they get a good grasp on the basic principles of programming. This new programming language can mainly be used to teach programming to high school students in Turkey who are taking an introductory course in computer science.

2 Previous Work

Today's popular programming languages (e.g. C++, Java) mostly use English keywords and have some resemblance to English syntax. Designing a programming language with syntax similar to the natural language is a concept which is controversial [1]. Unfortunately, we have not been able to find this kind of work related to a natural language other than English.

Perhaps, the oldest work along this line is COBOL (Common Business Oriented Language) [3]. An important concept that is considered in the design of COBOL was manipulability. It can be defined as the expressive power regarding its domain of application, namely business data processing. Another important concept was unambiguity. COBOL was designed to prevent possible ambiguities in the source code. Finally, learnability was another crucial concept that was considered in the design of COBOL.

Hypertalk [4] can also be listed as a programming language designed for novice programmers. Hypertalk makes use of a GUI that allows programmers to define interfaces for their own programs. The designers tried to develop an English-like language for achieving their goals.

Traditional programming languages are very successful in representing the model of a problem that the computers can run. But, this model could be far away from the mental plan that humans think of for the same problem. For this reason, developing new programming models is a research area that aims to design more comprehensible programming languages. The Natural Programming Project [5] in Human Computer Interaction Institute at Carnegie Mellon University is an example for the kind of work that is related to developing new programming models to make programming easier for novice programmers. The project group is identifying the problems of current programming languages and tackling these problems in their research. They conducted some experiments with children in order to understand how they formulate real-life problems. The information coming from these experiments is important for understanding a non-programmer's thinking style and constructing a programming model suitable for them.

Substituting Turkish counterparts of keywords in Java, C or a Pascal-like language while maintaining an apparently English syntax (e.g. head-initial for imperatives) produces a very unnatural flow of instructions. The novice programmer could hardly reconcile it with her way of thinking about imperatives in Turkish. The situation gets worse as instructions are nested. There are some programming languages that use Turkish keywords, such as TUPOL for which public documentation is not available. TUPOL has a syntax that is very similar to the syntax of the well-known programming language Pascal, as seen in Figure 1. But, as is the case with TUPOL, replacing the keywords does not necessarily make that language comprehensible. In case of replacing these keywords with the words of another language (e.g. English to Turkish), it is very likely that the resulting programming language will be much worse than the original because of the differences between Turkish and English (e.g. word order). In Figure 2 the same code fragment written in TPD is shown.

Using a programming language includes writing code and also reading the code. Hence, both are crucial for the usability of a programming language. Figure 3 illustrates a comparison of some programming languages in terms of the effect of program

```
not_finished := true;
sum := 0;
while not_finished do
    begin
       read(num);
       if num < 0 then
          not_finished := false
       else
          begin
             fct := 1;
             while num > 0 do
                begin
                   fct := fct * num;
                   num := num - 1
                end;
             sum := sum + fct
          end
    end
```

```
not_finished <= 1;
sum <= 0;
kosul not_finished iken
    blok
       oku(#s, num);
       eger num < 0 ise
             not_finished <= 0
       degilse
          blok
             fct <= 1;
             kosul num > 0 iken
                blok
                   fct <= fct * num;
                   num <= num - 1
                son;
             sum <= sum + fct
          son
    son
```

Fig. 1. A code fragment in Pascal and its counterpart in TUPOL.

```
not_finished <- doğru.
sum <- 0.
not_finished doğru oldukça {
    klavyeden num oku.
    eğer num < 0 {
       doğruysa:
          not_finished <- yanlış.
       yanlışsa: {
          fct <- 1.
          num > 0 doğru oldukça {
             fct <- fct * num.
             num <- num - 1.
          }
          sum <- sum + fct.
       }
}
```

Fig. 2. The code fragment in Figure 1 written in TPD.

sophistication on the difficulty of use [5]. There are two two considerations in interpreting this graph. One is the point where the lines intersect the y-axis (i.e. the axis that shows the difficulty of use). This point shows us the initial effort for a novice programmer to start programming in that particular language. As Figure 3 shows, it is easier to start programming with Visual Basic compared to other languages such as C++ and functional languages. The vertical jumps indicate that the programmer needs to stop and learn something totally new. For C++, the vertical jump shows the time when the programmer needs to learn the Microsoft Foundation Classes (MFC) to do graphics. Second is the slope of these lines. The slope directly shows the relationship between program complexity and difficulty of use. So, having a smaller slope is better in terms of usability and also code readability. Therefore, a programming language requiring an initial effort that is close to Visual Basic's initial effort and having a slope that is close to the slope of the functional languages (such as Scheme) would be ideal.

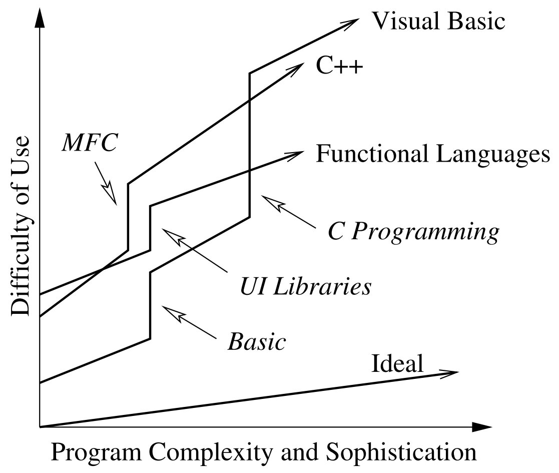

Fig. 3. The relationship between the program complexity and the difficulty of use [5].

3 Design Parameters

The programming process requires the transformation of a mental plan into another one that is compatible with the internal representation of the computer. Therefore, this process is easier if the transformation is straightforward. It is expected that using a syntax similar to that of Turkish will make this transformation process easier by guiding users to find an analogy with Turkish syntax. That was our motivation to design a language that is based on Turkish syntax. For example, in TPD, each statement has the verb at the end as it is the case in Turkish.

Many people think programming is difficult because it requires the use of a formal language. There is the claim that a visual language may be better for novice programmers to start with [6]. However, nobody has been able to show that a visual language is better than a textual one for *all* tasks. Often textual languages are superior to visual ones [7]. Considering these facts, we have decided to design a textual language. This should ease the TPD programmers' transition to common languages.

TPD is primarily intended to be used as a learning aid for fundamentals of programming for Turkish-speaking high school students. For this reason, TPD adopts well-established programming models instead of developing a new one. We hope to make it easier for TPD users to learn popular programming languages by drawing semantic parallels between these languages and TPD.

When designing a language, probably the first thing the designer must do is to choose a suitable programming paradigm. And, it is suggested that the designers can improve usability by not limiting the language to a single paradigm. Different paradigms are expected to be more natural for different parts of a task of programming [2]. Furthermore, it has been found in studies on users that inheritance hierarchies cause difficulty for children [6]. Even for professional programmers, using advanced features of object-oriented programming like inheritance or polymorphism is not necessarily natural [8]. Thus, TPD supports two well-known paradigms: imperative (procedural) and functional programming.

Inspired by Turkish, an agglutinative language, we discussed whether to exploit morphology of the language or not, and decided not to. The reason is that, using morphology causes more confusion and burden than good when the code is updated frequently. A natural-language-like programming language that still has a formal and rigid syntax is expected to be more likely to develop in a desirable manner [1]. Moreover, morphological features would probably hinder learning other programming languages.

We also do not want the users of TPD to succumb to the initial urge of novice programmers to use global variables. In TPD, it is not possible to define global variables. In addition, TPD uses call-by-value as the parameter passing mechanism. So, in a function, the programmer does not have a chance to change the value of a variable that is not local to that function. This way, the programmers are forced to use the return values of functions.

TPD compiler does not attempt to produce the most efficient code, because of the emphasis on diagnostics. We were designing a programming language for beginners, so efficiency of neither the produced code nor the compiler itself was the top priority.

4 Salient Features

The basic data types of TPD are integer, floating-point number, unsigned integer, character, boolean and file. There are also complex data types: record, list, array and function. The string type is a special case of array, i.e. character array. TPD adheres to the type completeness principle. Therefore, the programmer is able to declare variables of any type and perform operations on these variables. There are no arbitrary restrictions on the use of types. For example, a value of any type is storable via assignment. Literals for all complex data types are also supported.

Common mathematical, relational and boolean operations are provided. There are also conditional (multiway) and loop (both definite and indefinite) statements that allow the user to control the program flow. Some of these statements will be explained later.

5 Issues Identified

Designing an educational programming language first requires identifying some well-known problems of current programming languages in terms of usability. We tried to avoid the problematic cases and aimed to design a language that is more usable. By usability, we mean a concept that includes readability, expressibility and also maintainability. There is a trade-off in making a language more readable and making it more expressive. That is, if we want the code to be easily readable, then we have to provide verbose statements and this makes the usability of the language even worse. We tried to balance readability and expressibility by providing syntactic sugars for statements that have long forms. In addition, we have added a feature to the development environment that allows the user to replace syntactic sugars with their expanded forms. This feature aims to improve the readability of the code.

The novice programmers expect to find an analogue in their competence (e.g. in mathematics or knowledge of natural language) for the syntax of the programming language. Therefore, consistency with representations of the language and the programmer's other academic experience with them is of key importance in design. Inconsistent representations may cause the programmer to spend an extra effort to understand while writing or reading the source code.

An example for these inconsistencies is the division operator in C. It performs integer division by default when both parameters are integers. That is, `5/4` evaluates to `1`, not to `1.25`. However, relying on their knowledge of mathematics, the users expect this expression to have the value `1.25`. TPD handles this situation by checking if the dividend is divisible (with no remainder) by the divisor or not. If so, the operation returns an integer, else it returns a floating point number (i.e. `8/4` evaluates to `2` not `2.0` and `7/4` evaluates to `1.75`). The "not equal" is another example. There were some other alternatives such as `!=` or `<>` but we chose `=/` because it looks like $\neq$ and reads more like Turkish ("equal not"). Further, replacing `=/` with $\neq$ in the development environment the source code becomes more readable.

Another well known inconsistency is using equal sign for the assignment operation, as is the case in C and Java. The long form of the assignment statement in TPD is as follows:

variable `değişkenini` *expression* `yap .`

Considering the excessive usage of the assignment statement, we decided to provide some syntactic sugars for that statement:

variable `<-` *expression* `.`

expression `->` *variable* `.`

Here, we chose the arrow notation because we think that it conveys the sense of directionality. Choosing the equal sign for the assignment operation would make it hard for the users to understand its semantics, because it is used to express equality in mathematics.

Providing consistency with other academic experiences is not always possible. For example, for the multiplication operator, we chose `*` because the cross character is not available in a typical keyboard. The character `x` could be an alternative but then the user could not have a variable named `x` or use it in a variable name. Another example is the modulo operator. We initially decided to use `mod` but considering that the usage of `mod` in high schools was different (i.e. "`5 = 2 mod 3`" not "`5 mod 3 = 2`"), we changed our notation and chose `//` as an operator.

In Turkey, comma is traditionally used (instead of period) to separate the fractional part of real numbers. So, we had to choose between comma and the period. One of the reasons for us to choose the period is that using the notation will help the user to learn other languages easily. Another important reason is that choosing comma will make the parsing process very complicated, and in some cases, ambiguous. For example, when the user is writing a comma-separated list of numbers, it is not possible for the compiler to understand in a single pass which comma is used in a floating-point number and which comma is used for separating the numbers.

An experiment that was conducted for the Natural Programming Project clearly shows that choosing some keywords can result in misunderstanding while reading the source code of a programming language [2]. In this experiment, a problem was given to a group of children, and they were asked to formulate the problem. The results were quite revealing. For example, in 66 percent of the time the participants used the word "then" to express sequencing (e.g. "first he listens to his mother, then he goes to the school"). But the meaning of the word "then" in programming languages is "consequently" or "in that case". So, we paid extra attention when choosing the keywords of TPD in order to avoid possible misunderstandings.

Another common problem with the traditional programming languages is the error messages that the compilers produce. For a new programming language for the novice to be successful, the error messages are of key importance [9]. The error messages must be clear, helpful, precise and constructive. Saying just "`syntax error`" does not help the user to find the source of the error. Moreover, many times the error messages are completely unrelated to the real problem.Therefore, the diagnostic features of the compiler cannot be separated from the design of both the language and its environment.

Error messages of TPD are designed to be self-explanatory and simple. For example, when a type mismatch occurs, the expression that causes the error is highlighted in the editor part of the development environment, and the error message shows the required type, the provided expression's type, and states that they are not compatible. Another example is the run-time errors. Run-time system indicates the place of the error in the source code, and the development environment highlights them. This feature enables the user to easily debug the program.

In some cases, giving an error message and terminating the compilation can be the easy way for the designer, but it might make things more tedious for the user. For reducing this complexity, in order to eliminate very common and unnecessary error messages about type mismatches, some of the type casting is done automatically by the TPD compiler. For example, when the required type is floating-point number and the user provides an integer, the compiler casts the integer to the corresponding floating-point number without issuing an error. But in the case that the required type is integer, and the user provides a floating-point number with a non-zero fractional part (e.g. `1.01`), the run-time system gives a warning message to report the loss of precision.

Avoiding common errors can be seen as the responsibility of the programmer, but the programming language also has the chance to help the programmer to avoid them. For example, in a case- sensitive language, a novice programmer is likely to make some errors related to case-sensitivity. So, making the language case-insensitive helps the novice programmer. Also, dangling-else ambiguity and requiring "`break`" in each branch of C and C++ switch statements can be given as the examples for the sources of common errors that the programmers commit.

We tackle the well-known dangling-else problem by joining "`then`" and "`else`" parts of the conditional statement in a block as follows:

`Eğer` *expression* { `doğruysa :` *statement* `yanlışsa :` *statement* }

This statement is like the "`if`" statement of traditional programming languages. "`doğruysa`" corresponds to the "`then`" part and "`yanlışsa`" corresponds to the "`else`" part. The order of "`doğruysa`" and "`yanlışsa`" parts may change, and any one of them can be omitted. The general form of this statement is as follows:

`Eğer` *expression* {
    expression `ise :` *statement*
    ...
    [`hiçbiri` *]* `değilse :` *statement*
}

It is just like the "`switch`" statement of C. The difference is that the user does not have to write a "`break`" statement at the end of each case, and she can use non-discrete-valued data types such as floating-point numbers for *expression.* "`hiçbiri`" is an optional keyword. "`değilse`" case is same as the "`default`" case in C, and it may be omitted. Actually the first conditional statement is a special case of the second one. The keywords "`doğruysa`" and "`yanlışsa`" are just shortcuts for "`doğru ise`" and "`yanlış ise`" ("`doğru`" means "`true`", and "`yanlış`" means "`false`").

TPD uses structural equivalence mechanism while checking whether the types of two expressions are compatible or not. This is also useful for preventing the user from making simple mistakes, because it is more intuitive than the name equivalence mechanism.

Moreover, any variable and also any function in TPD can be called before its declaration takes place. This feature allows the user to have mutually recursive functions without having to deal with writing the forward declarations of each function that is used before its definition. Having the option to use variables before declaring them eases the writability but certainly decreases the readability of the code. For that reason, the development environment has an option to organize the code. That is, with this option turned on, the source code is automatically reorganized by moving the declarations to the beginning of the corresponding block or body.

6 Structure of the Language

Almost all the statements in TPD are in the form of a sentence in Turkish. These statements have the verb at the end, and each statement ends with a period. Grouping statements in a block statement is also possible. The syntax of the block statement is as follows:

{ *statement_list* }

The block statement is just a list of statements inside curly braces. As is the case in some programming languages such as Pascal, we could use a pair of keywords to mark the beginning and the end of a block. But this alternative causes problems in nesting and it makes the code look verbose. In case of nesting, the development environment helps the programmer by auto-indenting the code and by showing the matching parentheses.

In TPD, there is no pointer type. Although pointers are powerful tools, we did not include them in the language. The main reason is that, for a novice programmer, it will be quite difficult to deal with pointers, mainly because of their complicated semantics. Instead of pointers that allow dynamic allocation, there are dynamic data types (i.e. lists) that are allocated automatically by the run-time system.

Lists are very similar to those of Lisp; they are non-homogeneous collection of elements. That is, an element of a list can be of type integer, boolean, character, string or another list or an array. Moreover, there are some pre-defined functions on lists to perform basic operations such as getting the first element of a list (`ilki`), and discarding the first element and getting the resulting list (`gerisi`). Higher order operations such as applying an operation to all the elements of a list are available to the user as well.

Arrays in TPD are like the arrays of C. The programmer defines their size in the declaration, and later they can be reallocated, so the running program can increase or decrease the size of an array. When she tries to assign a value to an array member that is not allocated, an error message is produced at run-time. The arrays can be multi-dimensional in TPD, and all arrays start indexing from one (not zero, as in C). The reason is that usually people count from one upwards.

There are three loop structures in TPD. One is a *while-do statement*, the second is a *repeat-until statement*, and the last one is a *for statement*. Input/output statements allow the user to take input (from a file or from the standard input) and give output (to a file or to the standard output). Furthermore, type definitions are possible in TPD.

TPD also permits the user to define and use functions. The arguments are passed by value Functions are complete types in TPD. It is possible to define variables of type function. Therefore, the user can define higher order functions by having a function as a parameter of another function.

TPD also allows the user to include comments inside the code. A comment starts with a `!` and ends at the end of the line.

A sample TPD code is given in Figure 4.

7 The Compiler

The implementation of the compiler is accomplished through the use of Eli [10]. Eli is a compiler-generator which allows the designer to concentrate on abstract design by

```
! empty: takes a list as an argument, checks if it is empty or not,
!     and returns a boolean value
! insert: takes an integer and a list, inserts the integer to the list,
!     and returns the resulting list
! insertion_sort: takes a list, and returns this list sorted

Fonksiyon empty(liste a) : doğruluk değeri {
    A=[] değerini ver.
}

Fonksiyon insert(tamsayı elem, liste sorted) : dizi {
    First tamsayı olsun.

    Eğer empty(sorted) {
       yanlışsa: {
          First değerini ilki(sorted) yap.
          Eğer (first < elem) {
             doğruysa:
                First:insert(elem, gerisi(sorted)) değerini ver.
          }
       }
    }
    Elem:sorted değerini ver.
}

Fonksiyon insertion_sort(liste unsorted) : liste {
    Sorted liste olsun.

    Empty(unsorted) yanlış oldukça {
       Sorted değerini insert(ilki(unsorted), sorted) yap.
       Unsorted değerini gerisi(unsorted) yap.
    }
    Sorted değerini ver.
}
```

Fig. 4. Insertion sort in TPD.

providing a suite of tools for the compilation tasks. Therefore, Eli reduces the effort of the designer without sacrificing the efficiency of the generated compiler [10].

The compiler generates Java code, to be assembled to obtain the resulting Java Virtual Machine (JVM) byte code. This also provides the portability of the compiled programs.

Emphasis is given to the error-checking part of the compiler. Error messages are explanatory for avoiding the novice programmer from getting confused. The compiler generates an error-free Java code, and all the run-time error messages are generated by this code; the user does not see any JVM errors. As a result, the Java code that the compiler produces performs a lot of run-time checks. For instance, array references are checked and meaningful error messages with precise coordinates are produced if the programmer uses an array index that is out of bounds.

8 The Development Environment

TPD compiler is assembled in the development environment. The implementation of the development environment is in Java. Java is chosen because it has advanced libraries for writing an editor and some other tasks. Moreover, by choosing Java, we have a development environment that is platform-independently running on JVM.

The development environment has a language-based editor. The editor colors the input text reflecting its role. That is, keywords, comments and constants are displayed in different colors. Some errors are prevented by using this coloring mechanism. For example, the compiler colors the incomplete string literals in red, so that the user has a chance to see the error while typing. Another feature of the editor is auto-indentation; the code is automatically indented while the user is entering it.

Another property of the development environment is that, run and compile modes are not separately visible to the user. These modes are combined so that, the user only says run and the development environment compiles the program and runs it. In doing so, our aim was to simplify the operation of the environment.

Some other properties of the environment are: commenting out selected lines, and conversely, activating selected lines by removing comments, showing matching parentheses, and replacing sequences of ASCII symbols with their non-ASCII counterparts (i.e. it replaces `=/` with $\neq$, `>=` with $\geq$, etc.). In addition, the environment has some options to replace syntactic sugars with corresponding long forms and to organize the code by moving the declarations to the beginning of their scopes.

9 Conclusion

TPD is the first programming language whose syntax is based on Turkish syntax. So, TPD is important to see the effects of using a syntax similar to that of a natural language, in the design of a programming language. It is also important for us to understand the success of a natural programming language in Turkish. For making an assessment, psycho-linguistic experiments should be conducted on non-programmers.

Additional information on TPD can be found on the following web site:

`http://www.ceng.metu.edu.tr/tpp`

Acknowledgments

Thanks are due to Burak Yolaçan for his help and support in the beginning of the idea of developing an educational programming language. Also thanks to Erek Göktürk for his participation in the design.

References

1. Bruckman, A., Edwards, E.: Should We Leverage Natural-Language Knowledge? An Analysis of User Errors in a Natural-Language-Style Programming Language. Proceedings of the Conference on Human Factors in Computing Systems (1999) 207–214.
2. Pane, J. F., Ratanamahatana, C. A., Myers, B. A.: Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. International Journal of Human-Computer Studies **54(2)** (2001) 237–264.
3. Sammet, J.: The early history of COBOL. ACM SIGPLAN Notices **18(6)** (1981) 199–277.
4. Decker, R., Hirshfield, S.: The Analytical Engine, An Introduction to Computer Science Using HyperCard. Wadsworth Publishing, Belmont, 1994.
5. Myers, B. A.: Natural Programming: Project Overview and Proposal. Carnegie Mellon University School of Computer Science Technical Report no. CMU-CS-98-101 and Human Computer Interaction Institute Technical Report no. CMU-HCII-98-100 (1998).
6. Smith, D. C., Cypher, A., Spohrer J.: KidSim: Programming Agents Without a Programming Language. Communications of the ACM, Special Issue on Intelligent Agents **37(7)** (1994) 54–67.
7. Green, T. R. G., Petre, M.: When Visual Programs are Harder to Read than Textual Programs. Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics) (1992).
8. Detienne, F.: Difficulties in designing with an object-oriented programming language: an empirical study. Proceedings of IFIP INTERACT'90: Human-Computer Interaction (1990) 971–976.
9. McIver, L., Conway D.: Seven deadly sins of introductory programming language design. Monash University Dept. Computer Science Technical Report no. 95/234 (1995).
10. Waite, W. M.: Beyond LEX and YACC: How to Generate the Whole Compiler. University of Colorado Technical Report (1993).