A PROGRAMMING LANGUAGE FOR BEGINNERS BASED ON TURKISH SYNTAX

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

SERCAN TUTAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2004

Approval of the Graduate School of Natural and Applied Sciences.

Prof. Dr. Canan Özgen
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Ayşe Kiper
Head of Department

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

Assist. Prof. Dr. Halit Oğuztüzün
Co-Supervisor

Assoc. Prof. Dr. Cem Bozşahin
Supervisor

Examining Committee Members

Assoc. Prof. Dr. Göktürk Üçoluk (METU, CENG)

Assoc. Prof. Dr. Cem Bozşahin (METU, CENG)

Assist. Prof. Dr. Halit Oğuztüzün (METU, CENG)

Assoc. Prof. Dr. İ. Hakkı Toroslu (METU, CENG)

Assist. Prof. Dr. Soner Yıldırım (METU, CEIT)

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name: Sercan Tutar

Signature :

ABSTRACT

A PROGRAMMING LANGUAGE FOR BEGINNERS
BASED ON TURKISH SYNTAX

Tutar, Sercan

M.S., Department of Computer Engineering

Supervisor : Assoc. Prof. Dr. Cem Bozşahin

Co-Supervisor : Assist. Prof. Dr. Halit Oğuztüzün

September 2004, 70 pages

Programming is a difficult activity because it requires thinking in a way that ordinary people are not familiar with. It becomes more complex considering the unusual and sometimes contradictory (with daily life usage) symbols when designing programming languages. This thesis introduces an experimental programming language called TPD, which is designed to reduce the syntax- and semantics-oriented difficulties to a minimum and provide a head start in programming to high school students and novice programmers who are native speakers of Turkish. TPD mimics Turkish syntax in order to obtain a better learning curve by making use of the user's native language competence in learning the essentials of programming.

TPD supports both imperative (procedural) and functional paradigms. General lists are provided as a built-in data type. 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 features a graphical interface and a language-based editor.

Keywords: Programming Language Design, Development Environments, Psychology of Programming, Computers and Education

ÖZ

YENİ BAŞLAYANLAR İÇİN TÜRKÇE SÖZDİZİMLİ
BİR PROGRAMLAMA DİLİ

Tutar, Sercan

Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi : Assoc. Prof. Dr. Cem Bozşahin
Ortak Tez Yöneticisi : Assist. Prof. Dr. Halit Oğuztüzün

Eylül 2004, 70 sayfa

Programlama, sıradan insanların alışkın olmadığı bir biçimde düşünmeyi gerektirdiğinden dolayı, zor bir faaliyettir. Dilin tasarımında, olağan dışı ve bazen de günlük hayattaki kullanımları ile çelişkili sembollerin kullanılması ile daha da zorlaşır. Bu tez, TPD ismindeki, lise öğrencilerine ve ana dili Türkçe olan acemi programcılara programlamada bir başlangıç sağlamak ve sözdizimsel ve anlamsal kaynaklı zorlukları en aza indirmek için tasarlanmış deneysel bir programlama dilini sunar. Ek olarak, TPD, programlamanın temellerini öğrenmede, kullanıcıların doğal dil yeteneklerini kullanarak daha iyi bir öğrenme eğrisi elde edebilmek için Türkçe'nin sözdizimini taklit eder.

TPD hem prosedürel hem de fonksiyonel paradigmaları destekler. Genel listeler temel veri yapısı olarak sağlanmıştır. Eğitimsel endişeler göz önüne alındığında, programlama dilinin tasarımı, geliştirme ortamının tasarımı ile beraber düşünülmelidir. Derleyicinin hata tesbit özellikleri vurgulanmıştır. Hedef kod Java'da üretilmektedir. Geliştirme ortamı, bir grafik arabirim ve bir dil tabanlı editör sunar.

Anahtar Kelimeler: Programlama Dili Tasarımı, Geliştirme Ortamları, Programlama Psikolojisi, Bilgisayarlar ve Eğitim

To my parents and my younger brother

ACKNOWLEDGMENTS

The author wishes to thank his supervisors Assoc. Prof. Dr. Cem Bozşahin and Assist. Prof. Dr. Halit Oğuztüzün for their guidance throughout the design.

The author would also like to thank Burak Yolaçan for his help and support in the beginning of the idea of developing an educational programming language and Erek Göktürk for his participation in the design.

The assistance of Çağlar Günel in the implementation of the development environment is gratefully acknowledged.

Finally, the author would like to thank Utku Erdoğdu for his LaTeX support.

TABLE OF CONTENTS

PLAGIARISM iii

ABSTRACT iv

ÖZ v

DEDICATON vi

ACKNOWLEDGMENTS vii

TABLE OF CONTENTS viii

LIST OF TABLES xii

LIST OF FIGURES xiii

CHAPTER

1 INTRODUCTION 1

1.1 Previous Work 2

1.1.1 COBOL 2

1.1.2 Logo 2

1.1.3 Hypertalk 3

1.1.4 GRAIL 3

1.1.5 Karel the Robot 3

1.1.6 Natural Programming Project 3

1.1.7 TUPOL 4

1.2 Design Parameters 5

1.2.1 Textual vs. visual 7

1.2.2 Paradigm 7

1.2.3 Other parameters 7

1.3 Thesis Overview 8

2 DESIGN ISSUES 9

2.1 What to include? 10

2.2 Usability Problems 11

2.2.1 Identified inconsistencies 11

2.2.2 Error messages . . . 14
2.2.3 Avoiding common errors . . . 15
2.2.4 Other usability issues . . . 17
3 STRUCTURE OF THE LANGUAGE . . . 20
3.1 Programming Paradigm . . . 20
3.2 Lexical Elements . . . 21
3.2.1 Identifiers . . . 21
3.2.2 Comments . . . 21
3.3 Declarations and Definitions . . . 21
3.3.1 Variable declarations . . . 22
3.3.2 Constant declarations . . . 22
3.3.3 Type definitions . . . 22
3.3.4 Function definitions . . . 22
3.4 Types . . . 23
3.4.1 Primitive types . . . 24
3.4.1.1 Integer . . . 24
3.4.1.2 Natural number . . . 25
3.4.1.3 Floating-point number . . . 25
3.4.1.4 Character . . . 25
3.4.1.5 Boolean . . . 25
3.4.2 Composite types . . . 26
3.4.2.1 Array . . . 26
3.4.2.2 List . . . 27
3.4.2.3 Record . . . 27
3.4.2.4 String . . . 28
3.4.2.5 Function . . . 28
3.4.2.6 File . . . 28
3.4.3 Type coercions . . . 29
3.5 Expressions . . . 30
3.5.1 Grouping . . . 30
3.5.2 Operators . . . 30
3.5.2.1 Arithmetic operators . . . 31
3.5.2.2 Relational operators . . . 32
3.5.2.3 Logical operators . . . 33
3.5.2.4 Other operators . . . 33

3.5.3 Operator precedence and associativity 35
3.6 Predefined Functions 36
3.6.1 Mathematical functions 36
3.6.2 Functions for file handling 37
3.6.3 List functions 38
3.7 Commands 38
3.7.1 Iterative commands 38
3.7.1.1 While command 38
3.7.1.2 Repeat command 39
3.7.1.3 For command 40
3.7.2 Conditional command 40
3.7.3 Block command 41
3.7.4 Assignment command 41
3.7.5 Escapes 42
3.7.5.1 Return command 42
3.7.5.2 Break command 42
3.7.5.3 Continue command 42
3.7.6 IO commands 43
3.7.6.1 Print command 43
3.7.6.2 Read command 43
3.7.7 Expression command 44
3.7.8 Begin command 44
3.8 Overall Program Structure 44
4 INTEGRATED DEVELOPMENT ENVIRONMENT 48
4.1 Compiler 48
4.1.1 Lexical and syntactic analysis 49
4.1.2 Semantic analysis 49
4.2 Run-time libraries 50
4.2.1 Error checking 52
4.3 Development Environment 53
5 CONCLUSION 57
5.1 Future Work 58
REFERENCES 60
APPENDIX

A EBNF GRAMMAR OF TPD 62
B ENGLISH EQUIVALENTS OF TPD KEYWORDS AND PHRASES . . . 69
B.1 Types 69
B.2 Operators 69
B.3 Commands 69
B.4 Declarations 70
B.5 Constants 70
B.6 Predefined Functions 70

LIST OF TABLES

TABLES

Table 3.1 TPD operator precedence hierarchy 36

LIST OF FIGURES

FIGURES

Figure 1.1 Code fragment in Pascal and its counterpart in TUPOL 4
Figure 1.2 Code fragment in Figure 1.1 written in TPD 5
Figure 1.3 Relationship between the program complexity and the difficulty of use 6

Figure 2.1 Dangling-else example in TUPOL and its counterpart in TPD 16
Figure 2.2 Nested block command in TUPOL and its counterpart in TPD . . . 19

Figure 3.1 Example of disallowed access to outer function's locals 24
Figure 3.2 Code fragment that shows a possible ambiguity 40
Figure 3.3 Insertion sort example in TPD 45
Figure 3.4 Bubble sort example in TPD 46

Figure 4.1 Classes used for run-time type checking 52
Figure 4.2 Screenshot of the development environment 54
Figure 4.3 Screenshot of the output window 54
Figure 4.4 Screenshot of the execution window 55
Figure 4.5 Screenshot of the error window 55

CHAPTER 1

INTRODUCTION

Today, most of the computer users are not familiar with programming languages, and they cannot make use of them in their daily life. Many people who are enthusiastic about learning main programming concepts are discouraged in a very short period of time because of the difficulties. There are two main difficulties. One is the nature of the programming activity, which requires expressing the problem in a way that a computer can interpret, which is not necessarily the way a human being would describe it. The other is the design of the programming language. In the design, most of the times, the comprehensibility of the language is not the topmost concern of the designer. This results in inappropriate choices of symbols which forces the user to spend an extra effort.

In most traditional programming languages, the sequencing of symbols is inspired by mathematical conventions at the expression level, and by syntax of English at the command level. Efficiency of parsing is a major concern.

For comprehensibility, simply adopting some words from natural languages as symbols of a programming language may not be enough 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, 3]. Although some arguments about the potential dangers of such a similarity can be made [1], making use of the users' competence in natural language can be a good way to provide comprehensibility [2].

This thesis introduces a new programming language called TPD some of whose antecedent versions are announced previously [4]. The goal of TPD is to provide a head start in programming for Turkish speaking novice programmers, with minimal training. To reach this goal, comprehensibility has higher precedence than all

the other concepts in the design. The grammar thus attempts to mimic Turkish syntax with the hope that commands 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 - such as recursion, nesting and higher order functions - easier by allowing users to make an analogy between their knowledge of Turkish and the source code of the programming language. Without having to bother with learning 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 concepts of programming. Thus, one of the design criteria was not to deviate too much from established structures found in commonly accepted languages. 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 (such a course is presently offered as an elective in many high schools in Turkey).

1.1 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 [2]. Unfortunately, we have not been able to find this kind of work related to a natural language other than English.

1.1.1 COBOL

Perhaps, the oldest work along this line is COBOL (Common Business Oriented Language) [5]. An important concept that is considered in the design of COBOL was manipulability. It can be defined as expressive power regarding its domain of application, namely business data processing. Finally, readability was another crucial concept that was considered in the design of COBOL.

1.1.2 Logo

Logo is a very successful language which has been used and developed over the past 28 years [6]. The textual part of Logo is a dialect of Lisp. Its syntax was designed to

make the language readable. There are various Logo environments today. Most of them involves a turtle which is a robot that could be moved by using commands. For example, it can be used to draw some shapes on the screen.

1.1.3 Hypertalk

Hypertalk [7] 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.

1.1.4 GRAIL

Another language which is designed for educational purposes is GRAIL [8]. It is an imperative programming language which also uses English-like syntax. An important point about GRAIL is that it uses non-ASCII characters for some operations in order to reduce the common errors. For example, it uses ÷ operator for division. It has been showed that, compared to Logo, GRAIL significantly reduces the number of errors that the students make [9].

1.1.5 Karel the Robot

There are also some limited languages which are quite simple and designed to simplify learning the principles of programming without spending too much effort. These so called "mini-languages" [10] are similar to the existing languages, so they do not introduce new programming models. An example for these type of languages is Karel the Robot which can serve as an introduction to Pascal [11].

1.1.6 Natural Programming Project

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 [12] 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.

1.1.7 TUPOL

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 1.1.

```
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;
```

Figure 1.1: A code fragment in Pascal and its counterpart in TUPOL.

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. But, as is the case with TUPOL, replacing

the keywords does not necessarily make that language comprehensible. It might even have adverse effects. Replacing keywords with the words of another language (e.g. English and 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). We provide the TPD code for the same problem in Figure 1.2 for comparison. As we shall see later, as the nesting of commands get deeper, the comprehensibility of TUPOL gets worse but TPD manages to remain comprehensible. This has to do with word-order differences in English and Turkish.

```
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.
    }
  }
}
```

Figure 1.2: The code fragment in Figure 1.1 written in TPD.

1.2 Design Parameters

Using a programming language includes reading the code as well as writing the code. Hence, both are crucial for the usability of a programming language. Figure 1.3 illustrates a comparison of some programming languages in terms of the effect of program sophistication on the difficulty of use [12]. There are 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 1.3 shows, it is easier to start programming with Visual Basic compared to other lan-

guages 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 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.

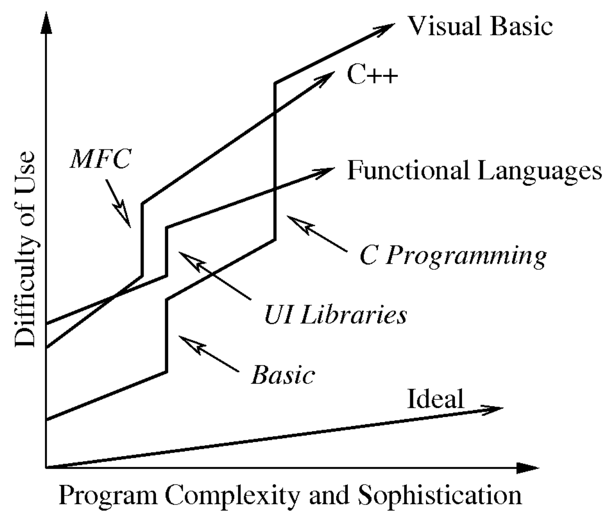

Figure 1.3: The relationship between the program complexity and the difficulty of use [12].

The programming process requires the transformation of a mental plan into another one that is compatible with the internal representation of the computer [13]. 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. Hence, TPD source codes should be very similar to an ordinary text in Turkish. To make it possible, each command has the verb at the end and finishes with a period as it is the case in Turkish sentences. But note that, it is not a natural language; we are still creating a formal language in which ambiguities are eliminated by design so that there is exactly one interpretation for any program.

1.2.1 Textual vs. visual

Many people think that 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 [14]. 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 [15]. Visual languages may be more suitable for small tasks but it is not the case for larger ones. Considering these facts, we have decided to design a textual language. This should ease the TPD programmers' transition to common languages (e.g. at the college level).

1.2.2 Paradigm

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 comprehensible for different parts of the activity of programming [3]. Furthermore, it has been found in studies on users that inheritance hierarchies cause difficulty for children [14]. Even for professional programmers, using advanced features of object-oriented programming like inheritance or polymorphism is not necessarily natural [16]. With these considerations in mind, TPD supports two well-known paradigms: imperative (procedural) and functional programming.

1.2.3 Other parameters

Inspired by Turkish, an agglutinating language with many productive use of suffixes, we debated 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. For example, assume that we have an

assignment command like

identifier "'" *obj_suffix* *expression* **olsun** "."

Here, "*obj_suffix*" corresponds to the objective suffix in Turkish. Depending on the phonological properties of the "*identifier*" this suffix can have eight different forms (i.e. ı, i, u, ü, yı, yi, yu and yü). For example, if the programmer defines a variable named "a", and she assigns different values to this variable in different parts of the program such as:

```
a'yı 5 yap.
```

Later on, if she decides to change the name of the variable to "b", she have to update all the suffixes. For example, the assignment command above will become:

```
b'yi 5 yap.
```

An alternative can be making no phonological tests and behaving all the objective suffixes as they are the same. Hence, the programmer does not have to update all the suffixes whenever name of a variable changes. But this significantly reduces the readability of the code. The compiler will accept commands like:

```
b'yı 5 yap.
b'ü 5 yap.
```

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 [2]. Moreover, morphological features would probably hinder learning other programming languages.

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 who tend to write short programs, so efficiency of neither the produced code nor the compiler itself was the top priority.

1.3 Thesis Overview

After this brief introduction, the thesis continues with Chapter 2 which introduces the issues that are identified in the design phase. Considering these issues, the structure of the resulting programming language will be explored in Chapter 3. Implementation and the properties of both the compiler and the development environment can be found in Chapter 4. Finally, conclusion and future work will be explained in Chapter 5.

CHAPTER 2

DESIGN ISSUES

As explained in Section 1.2, we decided to design a textual programming language that supports both imperative and functional paradigms, and we also decided not to provide morphological features. After making these decisions, we have started to investigate the ways to design a programming language that requires minimal initial effort to start programming.

Designing an educational programming language and trying to minimize the design oriented difficulties 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. As it can be understood from the definition, 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 commands and this makes the usability of the language even worse. We tried to balance readability and expressibility by providing optional keywords for commands that have long forms. Short forms can be used with more practice.

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 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. Hence, choosing even a single keyword or an operator can have a significant effect on the comprehensibility of the resulting language, as exemplified in the case of GRAIL (which is described in Section 1.1.4).

Considering these facts, we paid extra attention to every single detail while determining the structure of TPD.

2.1 What to include?

Before starting to think about the structure of the language, the elements to be included should be determined. After doing so, these elements can be designed one by one.

We do not want the users of TPD to succumb to the initial urge of novice programmers to use global variables (with program scope). In TPD, it is not possible to define global variables. In addition, TPD uses call-by-value as the only 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 (i.e. if it is defined inside another function, it cannot access locals of the outer one). This way, the programmers are forced to use the return values of functions. The notion of a function thus becomes closer to its mathematical counterpart.

Moreover, providing labels and jumps in an imperative programming language may not be a good idea. Therefore, these are excluded, and by doing so, it is aimed to force the novice programmer to get used to using conditional commands, iterative commands and escapes whenever she wants to control the direction of flow.

Determining the built-in data types of TPD was another crucial point. We decided to provide numerical types such as integers, natural numbers and floating-point numbers, which are commonly used in programming languages. Beginners will be familiar with these types from their mathematics courses. We also decided to provide boolean, character and string types, which are supported by most of today's programming languages. In addition, array and record types, which are frequently used in imperative languages, and similarly lists and functions of functional paradigm are added to TPD. Finally, to be able to introduce file processing concepts to novice programmers, files are also included.

Standard arithmetic, logical and relational operations are supported by TPD. Additionally, operators on lists and strings (such as concatenation) are also provided.

We aimed to teach imperative programming, therefore, decided to provide commands which are executed sequentially. Considering the properties of Turkish, com-

mands of TPD are designed in the form of a Turkish sentence. Turkish is a free-word-order language. Changing word order to manipulate the focus of a sentence is possible. But the main word order of Turkish is SOV (subject-object-verb), unlike the word order of English (which is subject-verb-object). Therefore, in parallel with regular Turkish sentences, TPD commands usually have the verb at the end and each command ends with a period.

2.2 Usability Problems

Using a programming language involves writing source codes and reading them. In this section, both tasks will be considered. Inconsistencies of current programming languages will be explored and how these inconsistencies influenced the design will be explained. Some other concepts, such as error message-oriented usability problems, mistakes which are frequently made by the programmers and ways to avoid them will be listed.

2.2.1 Identified inconsistencies

As mentioned in the beginning of this chapter, providing consistency is of key importance while trying to minimize the initial effort.

Some experiments that were 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 [3]. In these experiments, 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 the first experiment, 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"). The result goes up to 91 percent in the second experiment. But the meaning of the word "then" in programming languages is "consequently" or "in that case". So, using keywords consistent with their daily life usage in order to avoid possible misunderstandings appears to be a crucial point.

The division operator in C is also an example of inconsistency. It performs integer division by default when both parameters are integers. That is, `9/4` evaluates to `2`, not to `2.25`. However, relying on their knowledge of mathematics, the novice

users expect this expression to have the value `2.25`. TPD handles this situation by returning a floating-point number as a result of division and coercing a floating-point number with a zero fractional part to an integer without a warning message. It is simply equivalent to checking if the dividend is divisible (with no remainder) by the divisor and if it is so, returning an integer, else a floating point number (i.e. `8/4` evaluates to `2.0` which can be coerced to `2`, and `7/4` evaluates to `1.75`).

The "not equal" operator is another example. There were some other alternatives such as `!=` or `<>` but we chose `=/` to be consistent with the representation of this sign in mathematics because it looks like $\neq$. As it can be seen, there is another alternative which is `/=` but, `=/` reads more like Turkish (i.e. "equal not").

Continuing with the design choices, we can also give the modulo operator as another example. 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 arranged our notation as such and chose `//` as a newly introduced operator. Choosing `%` (as in C) does not seem to be intuitive because in daily life, it is used to denote percentage.

Another well known inconsistency is using equal sign for the assignment operation, as is the case in C and Java. An example assignment command in TPD is as follows:

```
x değişkenini 5 yap .
```

Considering the excessive usage of the assignment command, we decided to provide two assignment operations that performs the same task. Using these operations in an expression command results in the following:

```
x <- 5 .
5 -> x .
```

The only difference from the standard assignment command is that these operations have unused return values. So, actually these kind of commands can be thought of as syntactic sugars for the assignment command. We chose the arrow notation for the assignment operation because it conveys the sense of directionality. Choosing the equal sign for the assignment would make it hard for the users to understand its semantics, because it is used to express equality in mathematics. By using the arrow notation programmers are also able to write chained assignments like:

```
x <- y <- 5 .
```

```
5 -> y -> x .
x <- 5 -> y .
```

Note that these commands make use of the return value of the operation. Some erroneous usage of these operators might happen. Here are some examples of this kind of usage:

```
4 -> y <- 5 .
4 <- y .
```

Most of these problematic cases can be easily identified by considering the semantics of the arrow notation. For the first example, the programmer can easily find out that it is not possible to assign two values to an identifier at the same time. The details of assignment operators can be found in Section 3.5.2.4.

It is possible to expand the list of inconsistencies further. For example, ML uses $\sim$ for unary negative operation and minus sign is only used in subtraction. In addition, arrays of C and Java start from index value zero whereas usually people think of counting from one.

Designing a programming language based on Turkish syntax, while trying to be consistent with external representations, cannot be done without using accented characters (i.e. 'ı', 'ğ', 'ö', 'ü', 'ç' and 'ş') of the Turkish alphabet. A Turkish programming language without those (as it is the case in TUPOL) requires the users to get used to the notation and this takes some time. Considering this fact, we decided to allow the users of TPD to use all the accented characters in Turkish.

Despite the inconsistencies with real life experiences, some programming languages are not even consistent with their own representations. For example, Visual Basic uses equal sign for assignment unless the assigned value is an object. If it is so, "`Set`" keyword and equal sign are used together. Another example for this type of inconsistencies is that, in Pascal, accessing components of structured constants such as arrays is prohibited, but for variables this type of access is allowed. Finally, passing some variables by reference and some others by value can also be listed in this category.

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.

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 to represent decimal point. One of the reasons for us to choose the period is that using that notation will help the user to learn other languages easily. Another reason is that choosing comma would 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 is used for separating the numbers. An alternative could be forcing programmer to leave white space characters between values and commas used for separating them. But this solution requires the programmer to be careful while writing this kind of comma separated lists and also brings unwanted common mistakes. In addition, none of the today's popular programming languages uses this approach.

2.2.2 Error messages

A usability problem with most of 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 [17]. The error messages must be clear, helpful, precise and constructive. Saying just "segmentation fault" 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 the TPD development environment are designed to be self-explanatory and simple. For example, when a type mismatch occurs, the line that includes the expression that causes the error is marked, by double-clicking that mark the user can see the error message in a newly opened window, and the error message shows the required type, the provided expression's type, and states that they are not compatible. Another issue 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. An example for these run-time errors is the "array index out of bounds" error.

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. Type coercions will be explained in Section 3.4.3.

2.2.3 Avoiding common errors

Avoiding errors can be seen as the responsibility of the programmer, but the programming language also has the chance to help the programmer to avoid common ones. 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`**" commands 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 command in a block as follows:

```
Eğer x<y {
   doğruysa:  ...
   yanlışsa:  ...
}
```

This command is like the "**`if`**" command 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.

Figure 2.1 illustrates our solution to the dangling-else problem clearly. Code segments (a) and (c) are equivalent in TUPOL, and the compiler interprets both as the "**`degilse`**" part, which corresponds to "**`else`**", belongs to the second conditional statement (note that white spaces, that are used to indent, are not considered). And this can be confusing for the programmer. But in TPD, different interpretations have

different representations, which are illustrated in (b) and (d). TPD uses curly braces to group "**then**" and "**else**" parts in a block in order to deal with the dangling-else problem.

```
eger num < 0             eğer num < 0 {
   ise                      doğruysa:
     eger num < -5            eğer num < -5 {
       ise ...                  doğruysa:  ...
       degilse ...              yanlışsa:  ...
                              }
                         }
          (a)                       (b)

eger num < 0             eğer num < 0 {
   ise                      doğruysa:
     eger num < -5            eğer num < -5 {
       ise ...                  doğruysa:  ...
                              }
   degilse ...              yanlışsa:  ...
                         }
          (c)                       (d)
```

Figure 2.1: A dangling-else example in TUPOL and its counterpart in TPD.

We also designed another conditional command that is similar to the "**switch**" command of C. Later on in the design, we decided to join these two conditional commands in one. An example of the resulting conditional command is as follows:

```
Eğer x {
   < 5 ise :  ...
   5 ise :  ...
   hiçbiri değilse :  ...
}
```

In this command, the user does not have to write a "**break**" command at the end of each case as it is the case in C and Java. Another property of this command is that the programmer can use non-discrete-valued data types such as floating-point numbers. So, the "if" command can be thought of a special case of this one. The details of this command can be found in Section 3.7.2.

Another error that novice programmers frequently commit is choosing the wrong operator for performing an operation. Perhaps, the best example is using the assignment operator instead of equality check operator in C. That should also be seen as

a design oriented usability problem. Choosing similar representations for operators which have similar semantics results in this sort of confusion. Hence, small typos can result in compilable programs that perform incorrectly. This can also be listed as another reason for us to choose the arrow notation for the assignment operation and also using the equal sign for equality check.

Iterative commands are hard to understand because they require the programmer to keep track of exit conditions. However, there are definite iterative commands which allows the user to concentrate on the job done and as a matter of fact, they are more comprehensible. In these commands, usually, there is a variable that starts with an initial value and incremented (or decremented) until it becomes greater than (or less than) a final value. But in most of the programming languages, these commands have a potential usability problem. That is, the programmer is allowed to change the value of the variable that is used to determine the exit condition. When such a situation occurs, a programmer who is reading the code would have to start keeping track of that variable. In order to prevent this and benefit from the comprehensibility of the definite iterative command, TPD does not allow its programmers to change the value of the condition variable. It is a variable with no write permission, and it is not available outside the loop. The details of this command will be explained in Section 3.7.1.3.

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. There is no need (for the teacher) to spend an extra effort while trying to explain the semantics of the name equivalence mechanism. Expressions having the same structure are equivalent.

2.2.4 Other usability issues

In Section 2.1, we have listed the built-in data types, and string was one of them. Almost all the current programming languages support strings. But most of them handles strings in different ways. We had mainly three alternatives. First one is defining string as a primitive type, and providing some predefined functions. Another alternative is treating a string as an array of characters. This enables retrieving or updating character elements via array index operator. Finally, there is another

alternative, which is defining a string to be a list of characters. Note that, it is a homogenous list, unlike the lists of TPD. It has been argued that the last approach is the most natural [18]. We have considered all three options and decided to define string as both array of characters and list of characters. Array properties can be used with imperative structures such as iterative commands whereas list properties can be preferred when recursion is used.

When designing commands, we did not restrict ourselves by putting down some parameters such as 'every command must begin with a keyword'. For example, the repeat command directly starts with the subcommand, the keywords and the condition comes after. So, we tried to make the language more readable as long as the grammar is parsable. As a result, we obtained an LALR(1) language.[1]

Such as all the other imperative languages, TPD has a block command which allows the programmer to group several commands and introduce local variables as well. This command uses two atoms to denote the beginning and the end. Here, we had two alternatives for these atoms: using two keywords or two symbols (such as curly braces). We tried some keywords. Some of them were good in representing the block. But, in case of nesting, there was a potential problem that these keywords (especially the ones denoting the end) can pile up. Comparing the pile of keywords with pile of symbols, we decided to use the curly braces for the block command. The reason is that, a repeating keyword does not improve the readability of the code. Most of the times makes it even worse.

Figure 2.2 compares a nested block statement of TUPOL with its counterpart in TPD. As it can be seen from this figure, TPD provides a natural flow of commands whereas there is a pile of keywords (i.e. "**son**" is repeated thrice) in TUPOL which reduces readability.

Moreover, the editor part of the development environment shows the matching curly brace. By providing this feature, we tried to improve the readability of the blocks. The same block structure is also used for function bodies.

Function constants that are defined using function definitions can be called before their declarations take place. This feature allows the user to have mutually recursive functions without having to deal with writing forward declarations of each function that is used before its definition. But this does not apply to type definitions, variable

[1] TPD does not seem to be LL(1), although we have no proof of this yet.

```
kosul not_finished iken          not_finished oldukça
   blok                             {
     eger num < 0                     eğer num < 0 {
       ise ...                          doğruysa:  ...
       degilse                          değilse:
         blok                             {
           ...                              ...
           dongu i<=0 >> num                0 ile num
                                              arasındaki i için
             blok                             {
               ...                              ...
             son                              }
           num <= num - 1;                  num <- num - 1.
         son                              }
                                        }
   son;                             }
```

Figure 2.2: A nested block command in TUPOL and its counterpart in TPD.

and constant declarations. They must be declared before use. Recursive or cyclic type definitions are not allowed either. One can argue that, having the option to use variables before declaring them eases on-the-fly typing, but it certainly decreases the readability and maintainability of the code.

There are two main processes in the programming. These are compiling the source code and running the executable. We thought that it would be hard for the beginners to understand the semantics of these operations, and decided to hide the compilation from the user. As a result, TPD user just selects “run” from the menu and the environment compiles the source code and then runs the resulting program.

CHAPTER 3

STRUCTURE OF THE LANGUAGE

We have tried to keep the language as simple as possible, yet powerful. In this chapter, the aspects of TPD will be introduced in a systematic way.

3.1 Programming Paradigm

TPD mainly supports the imperative programming paradigm. This paradigm is the oldest but still the dominant one. It shows lots of similarities with machine languages, such as variables and assignment commands. Hence, theoretically, imperative languages can be implemented very efficiently. Today, it is possible to find lots of commercial software written in imperative languages, and almost all the programmers have experience in this paradigm. Concurrent and object-oriented programming paradigms can be thought of as subparadigms of imperative programming [18] as well.

Furthermore, TPD supports some advanced properties of functional languages. Programs in the functional paradigm consists of functions and expressions. This paradigm has some powerful features such as higher-order functions. In a functional language, there is no need for commands or procedures. Almost everything is handled by function evaluation. A function can be given as an argument to another, and by doing so, complex functions are constructed from simpler ones.

TPD tries to combine the expressional power of a functional language with the commands of an imperative one. Although it is not an expression-oriented language, it provides all the necessary features for functional programming. Lists as recursive data types and higher-order functions can be given as examples. TPD fully supports the imperative paradigm as well. Imperative concepts such as arrays, loops, and selective commands are provided.

3.2 Lexical Elements

TPD is case insensitive: The uppercase and lowercase versions of a letter are treated as they are the same. There are 53 keywords and 10 predefined function names in TPD. An EBNF grammar of the language including keywords can be found in Appendix A. English equivalences of the keywords and the phrases is given in Appendix B. In this section, lexical elements such as identifiers and comments will be covered. The other lexical elements such as literals of primitive types will be introduced in Section 3.4.1.

3.2.1 Identifiers

Identifiers of TPD start with a letter. The other elements of an identifier can be letters, digits or underscore characters. Each underscore must be enclosed within letters or digits. Considering these facts, identifiers of TPD has the form

```
LETTER { ( "_" ALPHA ) | ALPHA }
```

Here `ALPHA` represents the alphanumeric characters (i.e. letters or digits).

3.2.2 Comments

TPD only supports line comments. Each line comment starts with an exclamation mark (`!`), and ends with the end of line.

3.3 Declarations and Definitions

It is possible to declare variables, constants, and define types and functions in TPD. Each definition or declaration binds an identifier. There is only one name space in TPD. Therefore, all bound identifiers must be unique, which means, in the same scope, it is not possible to define a function and declare a variable with same names and vice versa. TPD does not support defining functions with same names and identifiying them in the context, namely, overloading of functions.

Each bound identifier has a scope which determines the lifetime of that identifier. The scope of an identifier is simply the nearest antecedent function body or block command in the parse tree. Constant access and variable access are the same in TPD. There is no restriction in accessing components of structured constants.

3.3.1 Variable declarations

In TPD, variables of any type can be declared. The variable declaration has the form

identifier { "," *identifier* } *type_denoter* **olsun** "."

Here, each *identifier* represents the name of the variable to be declared, and *type_denoter* shows its type. This declaration is the binding occurence of the variable, and it must take place before its applied occurences. An example variable declaration is as follows:

a, b, c **tamsayı olsun.**

3.3.2 Constant declarations

Constant declarations have the form

sabit *type_denoter identifier expression* **olsun** "."

Here, *identifier* is the name of the constant, *type_denoter* shows the type, and *expression* is the value of it. This declaration is the binding occurence of the constant, and it must take place before its applied occurences. An example constant declaration is as follows:

sabit gerçel sayı pi 3.14 **olsun.**

3.3.3 Type definitions

Type definitions of TPD are basically equivalent to renaming existing types. These have the form

tür *identifier type_denoter* **olsun** "."

Here, *identifier* is the name of the newly introduced type, and *type_denoter* shows the type that is renamed. This declaration is the binding occurence of the newly introduced type, and it must take place before its applied occurences. An example type definition is as follows:

tür integer **tamsayı olsun.**

3.3.4 Function definitions

Function definitions of TPD are just like shortcuts for defining function constants using the constant declarations except one difference. They have the form

fonksiyon *identifier*

```
"(" parameters [ "->" type_denoter ] ")" body
```

Here, `type_denoter` indicates the return type of the function, and it is optional. And `parameters` corressponds to

```
[ type_denoter identifier
    { "," type_denoter identifier } ]
```

where `type_denoter` represents the return type of the function, and `body` denotes the function body. Each function body has a seperate scope, and the identifiers declared or defined in a function body cannot be reached from the outside of the function. This declaration is the binding occurence of the function constant. Unlike other definitions and declarations, applied occurences can precede the binding occurence. This is the only difference between using a function definition and defining a function constant via constant declaration.

In addition, inner functions can be defined inside functions. But, a function cannot access to a variable or constant that is not a parameter or local of its own. That is, the inner function does not have the right to use or modify the value of a variable that belongs to an outer one. The reason is that, in TPD, writing higher order functions is possible, and as a result, a function can return another function. So it is possible to face with a situation like in Figure 3.1. In this piece of code, first, function "`h`" calls function "`f`" with an integer parameter "`5`". As you see, function "`f`" returns the value of function "`g`", and terminates. Then, the control comes back to function "`h`" again, and it assigns the return value to a variable named "`i`", and calls it. Here, the value of "`i`" is equal to the value of "`g`" hence, calling "`i`" would try to reach "`a`" which is defined in a previously terminated function. As you see, this operation must be prohibited in order to prevent this kind of invalid access (dangling reference). In addition, this restriction also simplifies the definition of a function, and makes it easier to comprehend.

3.4 Types

In TPD, every variable and constant has a fixed type which is declared by the programmer, and it is not possible to change a variable's type. Therefore, TPD is a statically typed programming language. In TPD, all values are first-class (i.e. can be evaluated, passed as arguments, assigned or used as components of composite val-

```
fonksiyon f(tamsayı n -> fonksiyon()) {
  a tamsayı olsun.
  ...
  fonksiyon g() {
    ...
    a <- a + 1.
    ...
  }
  ...
  g değerini ver.
}
fonksiyon h() {
  i fonksiyon() olsun.
  i değişkenini f(5) yap.
  i().
}
```

Figure 3.1: An example of disallowed access to outer function's locals.

ues [18]). TPD uses structural equivalence to check whether types of given composite values are compatible or not. Moreover, TPD gives more expressional power to its users by adhering to the type completeness principle (i.e. "no operation is arbitrarily restricted in the types of the values involved" [18]). Furthermore, all TPD types are bindables. The types of TPD can be classified according to their values.

3.4.1 Primitive types

Primitive types are the types with atomic values. The primitive types of TPD are as follows:

- Integers
- Natural numbers (unsigned integers)
- Floating-point numbers
- Characters
- Booleans

3.4.1.1 Integer

Integer is a storable and a discrete primitive type which is denoted by the keyword "**tamsayı**"

An integer can have values between -2^{63} and $2^{63}-1$. Integer literals have the form

DIGIT { *DIGIT* }

3.4.1.2 Natural number

Natural number is a storable and a discrete primitive type which is denoted by

"**doğal** [**sayı**]"

where "**sayı**" is an optional keyword. A natural number can have values between 0 and $2^{63}-1$. There is no specific natural number literal in TPD. Through the type coercion mechanism, integer literals are converted easily to natural numbers whenever needed.

3.4.1.3 Floating-point number

Floating-point number is a storable and a non-discrete primitive type which is denoted by

"**gerçel** [**sayı**]"

where "**sayı**" is an optional keyword. A floating-point number can have values up to $(2-2^{-52}) \times 2^{1023}$, and the smallest positive nonzero value is 2^{-1074}. Floating-point number literals have the form

DIGIT { *DIGIT* } '.' *DIGIT* { *DIGIT* }

3.4.1.4 Character

Character is a storable and a discrete primitive type which is denoted by the keyword

"**karakter**"

Character values and character literals are the same as Java's character values and Java's character literals.

3.4.1.5 Boolean

Boolean is a storable and a discrete primitive type which is denoted by

"**doğruluk** [**değeri**]"

where "**değeri**" is an optional keyword. Boolean values are true and false, and they correspond to "**doğru**" and "**yanlış**" respectively.

3.4.2 Composite types

There are also composite types whose values are composed of atomic values. The composite types of TPD are as follows:

- Arrays
- Lists
- Records
- Strings
- Functions
- Files

3.4.2.1 Array

Array is a composite type, and it is actually a finite mapping (i.e. maps index values to the corresponding array elements) which is denoted by

type_denoter "[" [*expression*] "]"

Here, *type_denoter* represents the element type, and *expression* should be an integer representing the size of the array. Size parameter is optional for parameters of functions. Multi-dimensional arrays are provided. The arrays of TPD are static. They are allocated during activation, and they cannot be resized afterwards. Array bounds are checked at runtime, and indexing starts from one. Array aggregate has the form

dizi *type_denoter*
"(" [*expression* { "," *expression* }] ")"

This aggregate constructs an array whose type is *type_denoter*. Elements are set in row-major order. An example array aggregate is as follows:

dizi tamsayı[2][2] (1,2,3,4)

Arrays can be both selectively and totally updated. Selective updating or retrieval is possible by using array indexing operator, and it is not limited by array elements. It is also possible to selectively retrieve or update the subarrays of a multi-dimensional array.

3.4.2.2 List

List is a storable and a composite type which is denoted by the keyword

“**liste**”

Lists are recursive types, and they are non-homogenous (i.e. they can have elements of different types at the same time). They are dynamic data structures that can be extended at any time in the execution. List aggregate has the form

"[" [*expression* { "," *expression* }] "]"

An example list aggregate is as follows:

[1, 2.8, ’c’, "75", [2, **doğru**], [ ]]

There exists two predefined functions that allows the programmers to retrieve the first element or the remaining list namely “**ilki**” and “**gerisi**” respectively. In addition, two operators “:” and “+” allows the programmer to construct new lists.

3.4.2.3 Record

Record is a composite type, and it is actually a Cartesian product which is denoted by

kayıt "{" [*field* { "," *field* }] "}"

where *field* is

type_denoter identifier

Record aggregate has the form

kayıt "{" [*field_agg* { "," *field_agg* }] "}"

where *field_agg* is either

type_denoter identifier

or

type_denoter identifier "=" *expression*

In the first case, the defined field does not have a value. In the second case, the field takes the value of the *expression*. An example record aggregate is as follows:

kayıt {**tamsayı** a=5, **gerçel** b=2.3}

Records can be both selectively and totally updated. Selective updating and retrieval is possible by using the record field selection operator. Each field has a name denoted by the *identifier* which is local to the record.

3.4.2.4 String

String is a composite type which is denoted by the keyword

"**yazı**"

A string can be treated as a list of characters or an array of characters. Strings are advanced data types which include the properties of both arrays and lists. They are dynamic such as lists, and they can be selectively retrieved or updated as arrays. Selective updating or retrieval is possible using the array notation. Predefined functions of lists, "`:`" and "+" operators are also applicable to strings. String aggregate is exactly the same as the string aggregate of Java.

3.4.2.5 Function

Function is a storable and a composite type, and it is actually an (potentially) infinite mapping (i.e. maps argument values to a return value) which is denoted by

```
fonksiyon "(" types [ "->" type_denoter ] ")"
```

where *types* denotes the parameter types, and it has the form

```
[ type_denoter { "," type_denoter } ]
```

Function aggregate has the form

```
fonksiyon "(" parameters [ "->" type_denoter ] ")" body
```

Here, each non-terminal has the same role as the ones in Section 3.3.4 which explains the form of the function definition. An example function aggregate is as follows:

```
fonksiyon (tamsayı a -> tamsayı) {
    a*a değerini ver.
}
```

3.4.2.6 File

File is a storable and a composite type which is denoted by the keyword

"**dosya**"

Variables of type file are first-class values like all the others. Programmers can open a file and close it by using functions "**aç**" and "**kapat**" respectively. Reading from a file or writing to it can be performed by using input and output commands. In addition, there are two postfix operators "**bitti**" and "**bitmedi**" to check the end of a file. Unlike all other types, a file type is actually a reference to the file itself.

Therefore, the programmer can assign the value of a file variable to another, and she can use both variables to read from or write to the same file.

3.4.3 Type coercions

TPD has a powerful type coercion mechanism for numeric types (i.e. integer, natural number or floating-point number). The following coercions can be done without loss of precision, and they are performed without giving a warning or an error message:

- From natural numbers to integers
- From natural numbers to floating-point numbers
- From integers to floating-point numbers

There are cases where loss of precision is possible:

- From integers to natural numbers: If the integer is greater than or equal to zero, it is coerced to a natural number without giving a warning message. Otherwise, a warning message indicating the loss of precision is given, and the absolute value of integer is converted to a natural number.

- From floating-point numbers to natural numbers: If the floating-point number can be coerced to a natural number without loss of precision, it is coerced without giving a warning message. Otherwise, a warning message indicating the loss of precision (i.e. loss of sign or fractional part or both) is given, and the floor of the absolute value of floating-point number is converted to a natural number.

- From floating-point numbers to integers: If the floating-point number has a fractional part that is equal to zero, it is coerced to an integer without giving a warning message. Otherwise, a warning message indicating the loss of fractional part is given, and the floor of the floating-point number is converted to an integer.

These type coercions do not apply to composite types. For example, an array of integers will not be coerced to an array of floating-point numbers and vice versa. There is no explicit type casting operator in TPD.

3.5 Expressions

In TPD, expressions are constructed from literals, aggregates, declared variables, and operators. These items can be grouped by using parantheses to manipulate the precedence. Variable declarations are introduced in Section 3.3.1. In addition, literals and aggregates are introduced in Section 3.4. TPD uses eager evaluation while determining the values of expressions. In this section grouping and operators will be explained in detail.

3.5.1 Grouping

Grouping of subexpressions in TPD can be performed by using parantheses. It is very much like an operator with the highest precedence, and it can be used by the programmer to change the evaluation order of expressions. Its application has the form

```
"(" expression ")"
```

Unlike the operators, it has no effect other than grouping. That means, if you have a left-value inside the parantheses, you will still have a left-value after the grouping takes place. This mechanism can be used when the programmer wants to modify the precedence in a left-value. For example the programmer can write an expression which is a left value and includes two selection operators as it is the case in

```
a @ b [ 5 ]
```

Using the precedence rules this is equivalent to getting the “`a`” element of record “`b`” and applying array index operator on that element which corresponds to

```
( a @ b ) [ 5 ]
```

But the user may also want to get the record field “`a`” of the indexed element of “`b`”. In that case she can use an expression like

```
a @ ( b [ 5 ] )
```

The order of operator evaluation can always be forced by the use of parantheses. It is often a good idea to use parantheses even when they are not required to make it explicitly clear how an expression is evaluated.

3.5.2 Operators

Let us start with classifying the operators:

- Arithmetic operators
- Relational operators
- Logical operators
- Other operators

3.5.2.1 Arithmetic operators

There are seven arithmetic operators in TPD. These are:

- **Addition operator:**
  It is adds two numeric values, and returns the result of the addition. It is an infix operator denoted by "+".

- **Subtraction operator:**
  It subtracts the second operand from the first one, and returns the result of the subtraction. Its operands must be numeric values. It is an infix operator denoted by "-".

- **Multiplication operator:**
  It multiplies two numeric values, and returns the result of the multiplication. It is an infix operator denoted by "*".

- **Division operator:**
  It divides the first operand by the second one, and returns the result of the division. The operands must be numeric values. It does not perform integer division even if both parameters are discrete numeric types. It is an infix operator denoted by "/".

- **Modulo operator:**
  It performs modulo operation by using its numeric valued operands, and returns the result of the modulo operation. Having a negative second operand is not allowed. It is an infix operator denoted by "//".

- **Unary minus operator:**
  It performs negation operation on a numeric value, and returns the result of the negation. It is a prefix operator denoted by "-".

- **Unary plus operator:**
  It is the regular unary plus operator, and it can be applied to a numeric value. It is a prefix operator denoted by "+".

3.5.2.2 Relational operators

There are six relational operators in TPD. These are:

- **Equal operator:**
  It is used to compare two values of any type except function, and it returns true if they are equal, false otherwise. It cannot take function parameters because, comparing values of two functions requires checking equality of two infinite mappings which is, in general, not decidable. It is an infix operator denoted by "=".

- **Not equal operator:**
  It is used to compare two values of any type except function, and it returns true if they are not equal, false otherwise. It is an infix operator denoted by "=/".

- **Less than operator:**
  It is used to compare two numeric values, characters or strings. Comparison of strings and characters are done in lexicographical terms. It returns true if the first operand is less than (or precedes) the second one, false otherwise. It is an infix operator denoted by "<".

- **Greater than operator:**
  It is used to compare two numeric values, characters or strings, and it returns true if the first operand is greater than (or succeeds) the second one, false otherwise. It is an infix operator denoted by ">".

- **Less than or equal operator:**
  It is used to compare two numeric values, characters or strings, and it returns true if the first operand is less than (or precedes) or equal to the second one, false otherwise. It is an infix operator denoted by "<=".

- **Greater than or equal operator:**
  It is used to compare two numeric values, characters or strings, and it returns

true if the first operand is greater than (or succeeds) or equal to the second one, false otherwise. It is an infix operator denoted by "`>=`".

3.5.2.3 Logical operators

There are three logical operators in TPD. These are:

- **Negation operator:**
  It is a postfix operator, and it performs logical NOT operation on its boolean valued operand. It has the form "**`değil`**".

- **Conjunction operator:**
  It is an infix operator, and it performs logical AND operation on its boolean valued operands. It has the form "**`ve`**".

- **Disjunction operator:**
  It is an infix operator, and it performs logical OR operation on its boolean valued operands. It has the form "**`veya`**".

3.5.2.4 Other operators

There are also twelve non-classified operators in TPD. These are:

- **Array indexing operator:**
  It is a postfix operator, and it allows the programmer to fetch (or update if the operand is a left-value) the indexed element of an array. This operator has the form
  `"[" `*`expression`*` "]"`.
  Here *`expression`* must evaluate to an integer between the array boundaries.

- **Record field selection operator:**
  It allows the programmer to retrieve (or update if the given expression is a left-value) the value of a record field. An application of this operator must have the form
  `identifier`` "@" `*`expression`*.
  Here the *`expression`* must be a record type, and it should have a field named *`identifier`*.

- **Function call operator:**

  Function call can be thought of as the application of a postfix operator whose application has the form

  expression "(" *arguments* ")".

  Here, *expression* must be of type function, and the *arguments* corresponds to

  [*expression* { "," *expression* }].

  TPD uses static binding mechanism which means the function bodies are executed using the environment of function definitions, not function calls [18]. Number of parameters check and individual type checks of parameters are performed at run-time. TPD uses call-by-value parameter passing mechanism (i.e. the values of the arguments are passed to the functions called, not the references. So, it is not possible to change the value of a variable by passing it as an argument to a function).

- **List concatenation operator:**

  It is an infix operator that is used to append two given lists. It has the same form as the addition operator, so "+" is an overloaded operator.

- **String concatenation operator:**

  It is an infix operator that is used to append two given strings. If only one of the parameters is string, and the other one is any other type (except file and function), the non-string typed parameter is converted to its string representation, and then the concatenation operation is performed. It has the same form as the addition operator, so actually it is another overloaded operator.

- **List construct operator:**

  It is an infix operator that allows the programmer to insert an element to the beginning of a list. The first operand is the element to be inserted, and the second one is the list. The element could be any type. It has the form ":".

- **String construct operator:**

  It is an infix operator that is used to insert a character to the beginning of a string. The first operand must be the character to be inserted, and the second one is the string. It has the same form, and it belongs to the same precedence

level as the list concatenation operator, so actually it is an overloaded operator.

- **Conditional operator:**

  It is an operator with three operands, and the first one must be boolean. If first operand evaluates to true, it returns the value of the second one, otherwise, the return value is equal to the value of the third one. Its application has the form *expression* **doğruysa** *expression* **değilse** *expression*.

- **Assignment operators:**

  These are infix operators that assign the value of one operand to another. One of them is the left assignment operator which has the form "`<-`". It assigns the value of the second operand to the first one, and it returns the assigned value. The right assignment operator has the form "`->`". Similarly, it assigns the value of the first operand to the second one, and it returns the assigned value.

- **End of file operators:**

  These are postfix operators that check the end of a file which is opened for input. One of them is the end of file operator which has the form "**bitti**". It returns false if the end of the file is not yet reached, true otherwise. The other operator is the not end of file operator which has the form "**bitmedi**". It returns false if the end of the file is reached, true if it is not the case.

3.5.3 Operator precedence and associativity

TPD operators are evaluated according to the precedence hierarchy shown in Table 3.1. Operators at low precedence levels are evaluated before operators at higher levels. Operators within the same precedence level are evaluated according to the specified association, either right to left or left to right. In Table 3.1, operators in same precedence level are not listed in any particular order.

For some operators, the operand types determine which operation is carried out. For instance, if the "+" operator is used on two lists, list concatenation is performed, but if it is applied to two numeric types, they are added in the arithmetic sense. If only one of the operands is a string, the other one is converted to a string, and string concatenation is performed.

Table 3.1: TPD operator precedence hierarchy.

Precedence level	Operator	Operation	Associates

1	`@`
	`-`
	`+`
		Record field selection
		Unary minus
		Unary plus
			Right to left

2	`[ ]`
	`(` *`arguments`* `)`
	`değil`
	`bitti`
	`bitmedi`
		Array indexing
		Function call
		Negation
		End of file
		Not end of file
			Left to right

3	`*`
	`/`
	`//`
		Multiplication
		Division
		Modulo
			Left to right

4	`+`
	`+`
	`+`
	`-`
		Addition
		List concatenation
		String concatenation
		Subtraction
			Left to right

5	`<`
	`>`
	`<=`
	`>=`
		Less than
		Greater than
		Less than or equal
		Greater than or equal
			Left to right

6	`=` `=/`	Equal Not equal	Left to right
7	**`ve`**	Conjunction	Left to right
8	**`veya`**	Disjunction	Left to right
9	`:` `:`	List construct String construct	Right to left
10	**`doğruysa değilse`**	Conditional	Right to left
11	`<-`	Assignment (left)	Right to left
12	`->`	Assignment (right)	Left to right

3.6 Predefined Functions

There are 10 built-in functions in TPD. These will be examined in this section.

- **Length:**
  This function is named "**`boyu`**". It has only one parameter of array type. Its return type is integer, and it returns the length of the first dimension of its argument.

3.6.1 Mathematical functions

Mathematical functions are as follows:

- **Floor:**

  This function is named "**taban**". It has only one parameter of numeric type. Its return type is integer, and it returns the floor of its argument.

- **Ceiling:**

  This function is named "**tavan**". It has only one parameter of numeric type. Its return type is integer, and it returns the ceiling of its argument.

- **Round:**

  This function is named "**yuvarla**". It has only one parameter of numeric type. Its return type is integer. It rounds its argument to the closest integer, and it returns that value.

- **Absolute value:**

  This function is named "**mutlak**". It has only one parameter of numeric type. Its return type is floating-point number, and it returns the absolute value of its argument.

- **Square root:**

  This function is named "**karekök**". It has only one parameter of numeric type. Its return type is floating-point number, and it returns the square root of its argument, provided it is non-negative.

3.6.2 Functions for file handling

There are two predefined file handling functions. These are:

- **Open:**

  This function is named "**aç**". It has two parameters. The first parameter must be a string representing the complete path of the file to be opened. The second parameter is also a string which shows the mode of the file. A file can be opened in write mode, read mode, both read and write mode, append mode, etc. Note that, both parameters are operating system dependent, therefore it is the programmer's responsibility to supply the correct parameters (e.g. using slash or backslash in the first parameter that shows the path). This function opens the file, and returns it. Some examples of calling this function are as follows:

```
aç("input.txt","o")  !opening an input file
aç("output.txt","y")  !opening an output file
```

- **Close:**
  This function is named "**kapat**". It takes only one parameter of type file, and it does not return any value. This function closes the previously opened file properly.

3.6.3 List functions

There are two predefined functions which operate on lists. These are:

- **First:**
  This function is named "**ilki**". It takes only one parameter of type list, and it returns the first element of this list. Note that, the return type can only be determined at run-time.

- **Rest:**
  This function is named "**gerisi**". It takes only one parameter of type list. It removes the first element of a list, and it returns the resulting list.

3.7 Commands

TPD supports the imperative programming paradigm as explained in Section 3.1. Therefore, commands to control sequential execution are provided in TPD. Commands also gives the opputunity to manipulate the direction of flow (e.g. conditional, iterative commands and escapes).

3.7.1 Iterative commands

An iterative command is a command including a loop body that is to be executed repeatedly. Usually, the iteration terminates when the terminating condition holds.

3.7.1.1 While command

This command provides indefinite iteration (i.e. number of iteration steps is not definite), and it has the form

expression [**doğru** | **yanlış**] **oldukça**

command

Here, *expression* is the condition that must hold in order to continue iterating the *command*, which constitutes the body of the loop. If keyword "**yanlış**" is used, then the condition must be false to continue the iteration. An example while command is as follows:

```
n < 5 doğru oldukça {
    sum <- sum + n.
    n <- n + 1.
}
```

3.7.1.2 Repeat command

This command also provides indefinite iteration, and it has the form

dot_removed_command

ta ki *expression* [**doğru** | **yanlış**] **olana kadar** "."

Here, *expression* is the condition that must hold in order to stop iterating the *dot_removed_command*. If keyword "**yanlış**" is used, then the condition must be false to continue the iteration. Unlike the while command, the terminating condition check is performed after executing the body, therefore, whatever the condition is, the body will be executed at least once.

In this command, *dot_removed_command* is used instead of *command*. It is same as the regular *command* but the only difference is the period (if there is one) is removed from the end. The reason is that, each TPD command must have a period at the end, and by doing so, we prevented to have multiple periods in a sentence-like command.

As it can be seen, there is a potential ambiguity problem that results from not using any leading keyword for the repeat command. That can be seen in Figure 3.2. Here, the problem is that the while command can include the repeat command in its body, and the opposite is also possible. This problem is solved by applying a precedence rule to the commands. Considering the most natural interpretation, we decided to design the repeat command to belong to the least precedence level among other commands. All the other commands are in the same precedence level.

```
x < 5 oldukça {
  x <- x + 1.
  y <- y + 1.
} ta ki y > 5 olana kadar.
```

Figure 3.2: A code fragment that shows a possible ambiguity.

3.7.1.3 For command

This command provides definite iteration (i.e. number of iteration steps is definite), and it has the form

```
expression ile expression arasındaki identifier için
  command
```

Here, `identifier` is a newly declared integer variable whose scope is the body of the for command (i.e. *command*). This variable takes values starting from the first *expression* up to the second one, and the user is not permitted to change its value. In every single iteration, its value is incremented by one. The loop terminates when the value of this variable becomes greater than the value of the second *expression*. An example for command is as follows:

```
1 ile n arasındaki i için
  sum <- sum + i.
```

3.7.2 Conditional command

A conditional command provides subcommands from which exactly one is chosen for execution. The conditional command of TPD is unique, and it has the form

```
eğer expression "{" ( case ":" command )+ "}"
```

The grammar rule for *case* is as follows:

```
case =
  expression ise |
  operator expression ise |
  doğruysa |
  yanlışsa |
  [ hiçbiri ] değilse;
```

Here, the first case is for checking equality. The second case consists of an infix re-

lational or logical operator and a second operand. It takes the `expression` which comes right after the "**eğer**" keyword as the first operand. Additionally, "**doğruysa**" and "**yanlışsa**" are syntactic sugars for "**doğru ise**" and "**yanlış ise**" respectively. The "[**hiçbiri**] **değilse**" case is the default case which is executed whenever there is no matching case above.

This command executes first matching case (i.e. executes the `command` which succeeds that case). The compared expressions may be any type except function. There is no such restriction as using only discrete values. An example conditional command is as follows:

```
eğer n {
    < 5 ise:  n <- n + 1.
    5 ise:  n <- 0.
    < 7 ise:  n <- n - 1.
    değilse:  fonksiyondan çık.
}
```

3.7.3 Block command

This enables grouping of a number of subcommands in one command. The execution of a block command is equal to executing all subcommands sequentially. Like the function body, which is explained in Section 3.3.4, each block command introduces a new scope. The block command of TPD has the form

```
"{" { definition | declaration | command } "}"
```

An example block command is as follows:

```
{
    n tamsayı olsun.
    n <- 5.
}
```

3.7.4 Assignment command

Assignment command has the form

```
expression değişkenini expression yap "."
```

The first `expression` must be a variable, and its type must be compatible with

the second one. With the execution of this command, second *expression*'s value is assigned to the first one. Unlike the assignment operators which are explained in Section 3.5.2.4, there is no return value of this command just like all the other commands. An example assignment command is as follows:

n **değişkenini** 5 **yap**.

3.7.5 Escapes

An escape is a sequencer that allows the programmer to terminate the execution of a block. Escapes are the only sequencers in TPD. There are no jumps or exceptions.

3.7.5.1 Return command

Using the return command, the programmer is able to exit a function body whenever she wants. There are two different return commands. One of them has the form

expression [**değerini**] **ver** "."

Here, the function exits returning the value of the *expression*. In TPD, it is also possible to define functions without a return value. The second return command provides an exit from this type of functions. It has the form

[**fonksiyondan**] **çık** "."

3.7.5.2 Break command

Break command of TPD allows the programmer to exit an iterative command's body. It has the form

bırak "."

3.7.5.3 Continue command

Continue command should be placed inside an iterative command. It skips the remainder of the loop body, and another iteration can start. This command has the form

devam et "."

3.7.6 IO commands

TPD also provides input and output commands that enables the programmer to read from the standard input or from a file and to write to the standard output and also to a file.

3.7.6.1 Print command

Print command allows the programmer to perform output operations. There are two distinct print commands. One of them has the form

ekrana *expression* { "," *expression* } **yaz** "."

This command writes string representations of all its parameters to the standard output. There is also another print command which provides file output. It has the form

expression **dosyasına**

expression { "," *expression* } **yaz** "."

Here, the first *expression* should be a file which has been opened for writing, and similarly this command writes other parameters to this file.

Functions and files do not have any string representations so that they cannot be written. An example print command is as follows:

Ekrana "Sum = ", 5+7 **yaz.**

3.7.6.2 Read command

Read command allows the programmer to perform input operations. There are two distinct read commands. One of them has the form

klavyeden *expression* { "," *expression* } **oku** "."

Here, the parameters should be variables. This command reads new values for these variables from the standard input, and it assigns each value to the corresponding variable. There is also another read command which provides file input. It has the form

expression **dosyasından**

expression { "," *expression* } **oku** "."

Here, the first *expression* should be a file that has been opened for reading. Similarly, the other parameters must be variables (left-values), and this command reads their values from the file.

Only integers, floating-point numbers, natural numbers, and characters can be read. An example print command is as follows:

```
Input dosyasından a, b, c, d oku.
```

3.7.7 Expression command

Expression command consists of an expression followed by a period. It is provided in order to be able to call functions or use assignment operators as a command. Note that the value of the expression is not used, therefore, using expressions with no side effects are not useful.

3.7.8 Begin command

The begin command is the only command which can be used outside the function bodies. Unlike the other commands, it cannot be used inside a function body. The execution of a TPD program starts with the begin command. Hence, multiple begin commands are not allowed. This command has the form

```
başla expression "."
```

Here, the TPD program starts with evaluating `expression`. The type of this expression cannot be file or function because the value of this expression will be printed on the screen, and these types cannot be printed. The value returned by this expression is shown to the user by using the development environment. In addition, it is possible to use non-valued expressions (e.g. functions without a return type) in this command. If it is the case, nothing is shown to the user as the output of the program. Each TPD program should exactly have one begin command. An example print command is as follows:

```
Başla factorial(15).
```

3.8 Overall Program Structure

A TPD program consists of function definitions, type definitions, constant declarations and a begin command. Inside a function body, the programmer can use variable declarations, constant declarations, function definitions, type definitions and commands (except the begin command). These can also be used inside the block commands. Introducing new scopes with function definitions and block commands is

```
!  empty:  takes a list as an argument, checks if it is
!       empty or not, and returns a boolean value

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

!  insert:  takes an integer and a list, inserts the integer
!       to the list, and returns the resulting list

Fonksiyon insert(tamsayı elem, liste sorted -> liste) {
   First tamsayı olsun.

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

!  insertion_sort:  takes a list, and returns this list
!       sorted

Fonksiyon insertion_sort(liste unsorted -> liste) {
   Sorted liste olsun.

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

Başla insertion_sort([7,6,2,4,3,5,1,8]).
```

Figure 3.3: An insertion sort example in TPD.

```
!  bubble_sort:  takes an array of integers, and returns this
!       array sorted

Fonksiyon bubble_sort(tamsayı[] given -> tamsayı[]) {
   length, tmp tamsayı olsun.

   length değişkenini boyu(given) yap.

   !  each iteration guarantees that the ith element of the
   !       array is set properly

   1 ile length-1 arasındaki i için

     !  each sub-iteration compares the value of the ith
     !       element with a succeeding (jth) element

     i+1 ile length arasındaki j için
       Eğer given[i]>given[j] {
         doğruysa:  {

           !  the following three commands swaps the values
           !       of ith and jth elements of the array

           tmp değişkenini given[i] yap.
           given[i] değişkenini given[j] yap.
           given[j] değişkenini tmp yap.
         }
       }

   given değerini ver.
}

Başla bubble_sort(dizi tamsayı[8] (7,6,2,4,3,5,1,8)).
```

Figure 3.4: A bubble sort example in TPD.

possible, and variables of outer scopes are reachable from block commands. Global variables or commands (excluding begin command) which are not in any function body are not allowed in TPD. Two example TPD programs can be found in Figure 3.3 and Figure 3.4.

CHAPTER 4

INTEGRATED DEVELOPMENT ENVIRONMENT

Today, practical programming languages are used with the help of some integrated development environment (IDE). An IDE is a programming environment that is packaged as an application program, typically consisting of a compiler, a graphical user interface (GUI), and a source code editor. Some IDEs may also provide a debugger or a GUI builder (for programming languages that support GUI development).

IDEs provide a user-friendly framework for many modern programming languages, such as Java and Visual Basic. They ease the work done by the programmer by providing advanced features. These features can be syntax coloring, auto-indentation, auto-completion of keywords and library routines, code folding, matching parentheses, showing errors while editing (before compilation), etc.

We considered that it would be very useful to provide a simple IDE with a source code editor for TPD. This IDE is expected to reduce the common errors of the programmers, compared to writing codes with a regular text editor. With the help of this IDE, we would be able to hide the compilation step from the user as previously explained in Section 2.2.4.

In this section, the certain aspects of the programming system are introduced. The parts of the system (the compiler and the development environment) will be covered in detail.

4.1 Compiler

The implementation of TPD parser is done in Eli [19]. Eli is a compiler-generator which allows the designer to concentrate on abstract design by providing a suite of tools. These tools are specialized in compilation tasks that are performed by most

of the compilers. For example, PTG (Pattern Based Text Generator) can be used for code generation, OIL (Operator Identification Language) eases tasks such as type checking of operands and overloading operators. There are other tools that provide monitoring of the produced compiler. For example, if there is a dependency problem in the attribute grammar (e.g. cyclic dependency), the programmer can use GORTO (Graphical Order Tool) to find the source of the problem. Therefore, Eli reduces the effort of the designer without sacrificing the efficiency of the generated compiler.

4.1.1 Lexical and syntactic analysis

Eli generates a parser for a given LALR(1) grammar. Hence, we first built an LALR(1) grammar of TPD. The resulting grammar can be found in Appendix A (without precedence rules).

In order to generate precise error reports, grammar is designed in order to be able to parse some erroneous inputs. For example, using commands that are not in any function body, defining global variables, and using expressions instead of variables, results in parsable input files. Instead of giving meaningless syntactic error messages for these errors, TPD compiler gives more comprehensible semantic error messages.

Tasks like construction of the parse tree, and inserting tokens to the symbol table are handled automatically by Eli.

4.1.2 Semantic analysis

In addition to lexical and syntactic analysis, TPD compiler also performs some semantic checks. Semantic analysis is performed by carrying information in the parse tree with the help of attributes and properties.

Scope checking is one example. Each bound identifier has a scope property, which is a unique integer value associated with it. In each scope, TPD checks if a binding occurence

- has a unique name,

- has a name which does not collide with any of the predefined names,

- (for variable declarations) is inside a function body.

In addition to these binding occurence checks, TPD also checks to see if an applied occurence

- is bound,
- is used after binding occurence.

If the input does not satisfy any of these, the compiler will produce an error message.

All the checks related to type definitions are done at compile-time. As previously mentioned in Section 3.3.3, type definitions are equivalent to renaming of current types. Hence, the compiler replaces these newly introduced type names with corresponding types. For type definitions, in addition to standard binding occurence checks, compiler checks if the definition is recursive or not.

No type checking is performed by the compiler. The main reason is that, in TPD, it is possible to have expressions whose types cannot be identified at compile-time. For example, consider applying the predefined function first to a list. Since lists are non-homogenous data types, it is not possible to determine element types at compile-time.

Limited checks are done for expressions. One of them is checking to see if the given expression is a variable or not. Another check is for the array aggregates: The type of the array must be provided while writing an array aggregate. Compiler checks to see if the given type denoter is an array or not. Finally, expressions used in a function body are checked to see if they include any variable or constant that is declared in another function body. As previously explained in Section 3.3.4, using an outer function's variables or constants in an inner function is not allowed.

There are also some command related checks that TPD compiler performs. For the begin command, compiler checks to see if it is used globally (outside the function definitions). Similarly other commands are checked to see if they are inside function bodies. In addition, escapes are checked to see whether they are used in iterative commands or not. There is another check for expression commands. If it is not possible for the expression command to have a side effect (i.e. no assignment operation or function call is used inside the expression), the compiler issues a warning message, and does not generate any code for that command.

4.2 Run-time libraries

TPD compiler generates Java code, to be assembled to obtain the resulting JVM (Java Virtual Machine) byte code. The Java programming language gives the opportunity

of compiling output (called "bytecode") that can run on any computer system platform for which a JVM is provided to convert the bytecode into instructions that can be executed by the actual processor. The bytecode can optionally be recompiled at the execution platform by a just-in-time compiler.

The generated Java code uses run-time libraries written to perform semantic analysis (such as type checking) in run-time. Every expression in a TPD code is converted to a piece of Java code which includes information about both beginning and end coordinates, and also the value of that particular expression. These coordinates are used in reporting errors with their locations in the source.

The libraries involved in type checking use a set of classes. These classes are summarized in Figure 4.1. Using these classes, implementation of a dynamically typed language is possible. The class "MyType" is a super-class of all others. Every TPD expression is actually an instance of a subclass extending "MyType". These expressions are accepted to be an instance of "MyType", and by using a method which gives the type of the expression, the subclass that the expression belongs to is found. After finding out the subclass, the expression is cast to that specific class. By doing so, the types of expressions are determined.

Every subclass includes methods that are applicable to that type. "MyType" also includes the same methods, and if any of these are called (that means the method is applied to a wrong type), an error message is issued.

The other classes in run-time libraries of TPD are:

- MyArrayIndex: This class is used to represent array indexes which are used to retrieve or update array elements.

- MyCoordinates: This class is used to keep beginning and end coordinates (which shows its place in the TPD source code) of an element. This element could be an expression, an operator or a command.

- MyIO: This class includes methods which are used for reading from standard input and writing to standard output.

- MyMessage: This class includes methods which are used to issue error messages. It uses "MyCoordinates" to report the coordinates of the error.

- MyOperator: This class includes methods that are used to evaluate the values

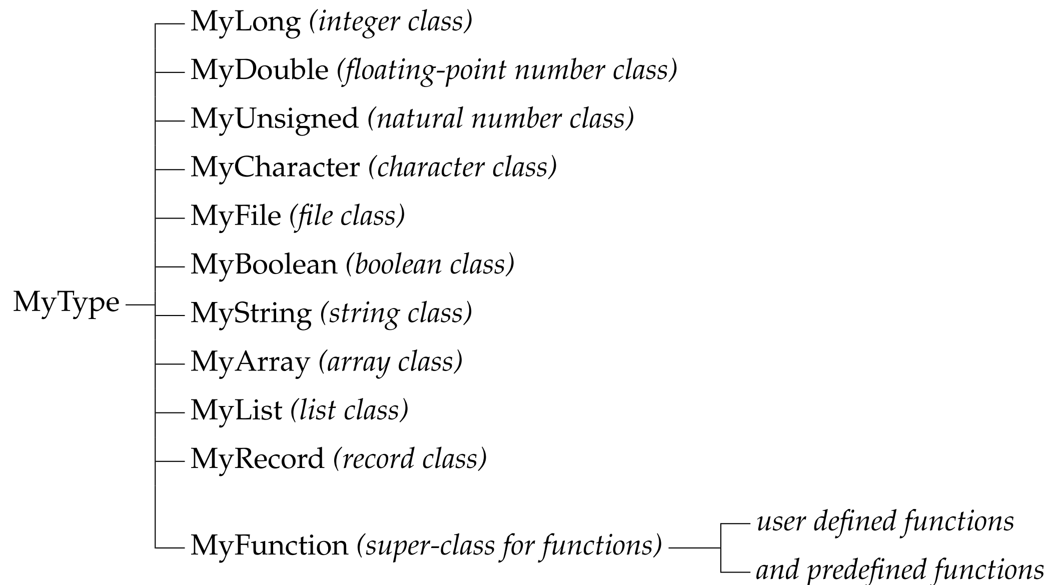

Figure 4.1: The classes used for run-time type checking.

of TPD operations. It also performs type-checking on the operands.

- MyTypeChecker: This class includes methods used to determine if two given types are compatible or not.
- MyTypeQualifier: This class is used to keep type-related information (such as element type and size of an array) for each TPD expression.

4.2.1 Error checking

The compiler generates error-free Java code; the user does not see any JVM errors. As a result, the Java code that the compiler produces performs extensive run-time checks. There are almost 100 unique error and warning messages that the run-time libraries can report. We give the flavour of error reporting, rather than provide an exhaustive list.

Every variable used as a right-value is checked if it is initialized or not. Variables of composite types, for which partial retrieval is possible (i.e. arrays and records), will be initialized if and only if all the elements are initialized.

There are also some special checks for composite types. Array bounds, number of dimensions and names of the record fields are checked in run-time. Files are checked whether it is possible to write to or read from them.

Operators are checked whether they are applicable to the provided operands. If not, an error message including the coordinates of the operands and the operator is issued. The error message shows the types of the operands, and states that this operation is not applicable to these operands. In addition, disallowed operations (such as division by zero) are reported to the programmer.

As explained in Section 3.4.3, some warning messages are issued to report type coercions with some kinds of precision loss.

Applications of predefined functions are also type-checked. For example, applying first or rest to an empty list or an empty string, and trying to get the length of a non-array-typed expression are not allowed. Some errors may also indicate the failure of operations, such as opening a file (e.g. trying to open a read-only file for output, or trying to open a folder instead of a file).

There are also some checks for commands. Some examples are trying to read from a file that is opened for output, comparing incompatible types in a conditional command, using a non-boolean expression for the termination condition of an iterative command, etc.

Examples of function-related error cases can be listed as follows:

- Using return command that returns no value

- Not using any return command in a function that has a return value

- Using a return command to return a value from a function that does not have one

- Giving wrong number of parameters to a function called

All the errors listed in this section (and many others) have unique error messages that clearly explain the type of the error.

4.3 Development Environment

TPD compiler and run-time libraries are assembled in the development environment. The implementation of the development environment is done in Java. Java is chosen because it has advanced libraries for writing a GUI, 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 editor colors the incomplete string literals in red, so that the user has a chance to see the error while typing.

The environment also allows the user to open multiple TPD programs at the same time as shown in Figure 4.2. The user can run any of these programs by activating its window and giving the run command by using the menubar.

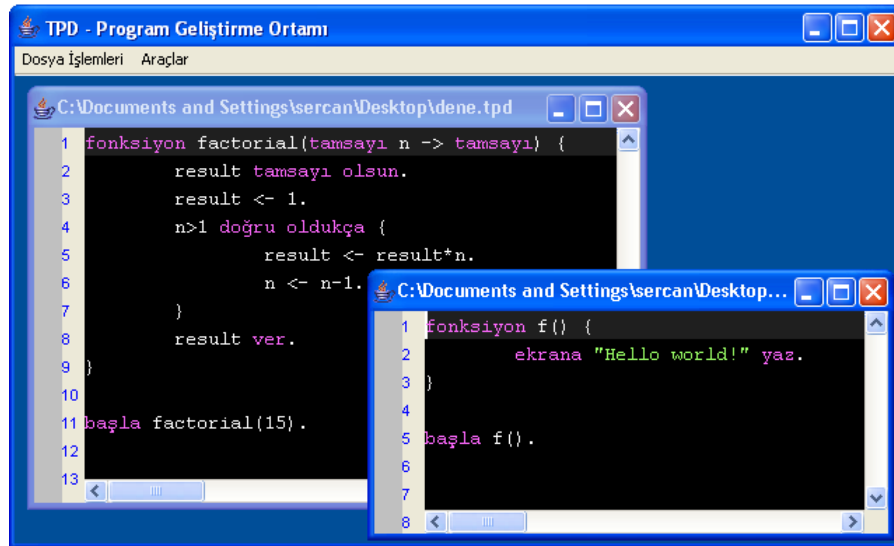

Figure 4.2: A screenshot of the development environment.

When the program starts to run, an output window is shown to the user. An example output window can be seen in Figure 4.3. The text field on top of this window enables the user to give input to the running program by writing text to this field and then pressing the button on the upper right corner. This text is sent to the standard input of the running TPD program. Similarly, the text area in the bottom of the window acts just like the standard output for the running program. A thread starts to run with the program, and directs its output to this window.

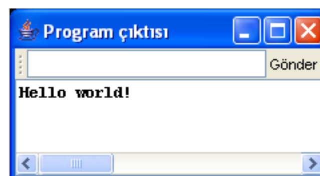

Figure 4.3: A screenshot of the output window.

In addition to coloring the text, the editor has another feature: matching parentheses. When the programmer moves the cursor to a parenthesis, the editor shows the matching parenthesis by drawing a bounding-box around it. This feature can be used, for example, to determine the beginning or the end of a block.

Another property of the development environment is that, the 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.

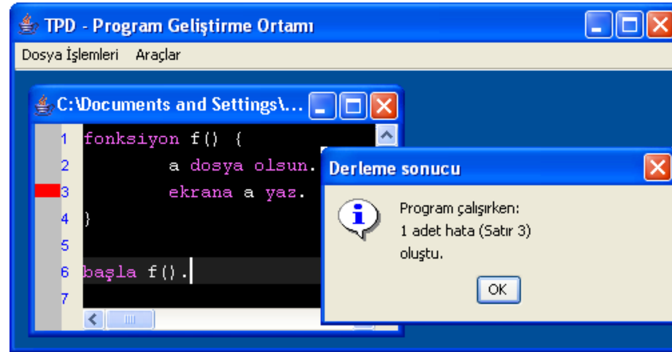

Figure 4.4: A screenshot of the execution window.

Error messages given by the compiler or by the run-time classes are shown to the user. These messages are parsed by the environment. The result of the execution is shown in a new window. This window basically shows number of warnings and errors with their line numbers. An example execution window is seen in Figure 4.4.

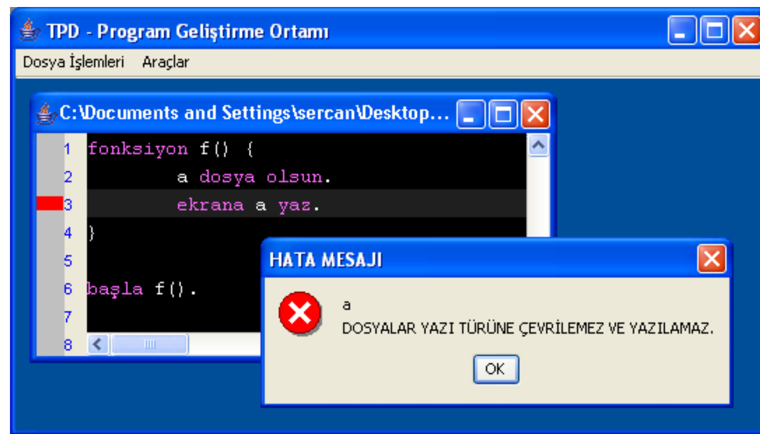

Figure 4.5: A screenshot of the error window.

After an execution which results in some errors or warnings, the problematic lines

are marked in the editor window. The color of this mark is different for errors and warnings (yellow for warnings, and red for errors). When the user double-clicks any of these marks, the elements of the error message are retrieved from the editor, and these are shown to the user in a new error window.

CHAPTER 5

CONCLUSION

This thesis introduces an experimental programming language, TPD. It is a language which tries to be consistent with the external representations of the programmer's other academic experiences. Furthermore, TPD aims to have more readable source codes for Turkish-speaking programmers compared to other programming languages by providing similarities between the commands of the language and the syntax of programmer's native language.

Therefore, TPD could be important to see the effects of both minimizing inconsistencies and using a syntax similar to that of a natural language in the design of a programming language. It could also be important for us to understand the success of a natural programming language in Turkish. Of course, it is not possible to evaluate the success of TPD without making assessments.

TPD is not the last step that is taken in order to obtain a programming language that perfectly satisfies the goal of usability. The design steps for a natural programming language (which has the same goals as TPD) may be as follows:

- *What will this programming language be used for?* Determine the needs and what to include in the resulting language.
- *Who will use this programming language?* State the target audience clearly.
- *Identify the target audience's academic experiences* which will also have representations in the language.
- *Design* a programming language trying to be consistent with the concepts identified in previous step.
- *Assess the success of the programming language* by making psycho-linguistic ex-

periments on a group of target audience. These experiments should be done by comparing learning curves of the newly introduced programming language with that of another language which has similar features.

- *Identify the problems* of the programming language by considering the results of the experiments. Listing common mistakes of the programmers may be useful.

- *Re-design* the programming language considering these problems. Assess the re-designed language and use the feedback coming from those assessments to improve the language. Repeat these steps until you get a learning curve which is "good enough".

After deciding on what to include, by identifying the target audience, designing the structure of the programming language and implementing a development environment for it, we had completed the first four steps of this design process. Now, there is a programming system which is ready to be evaluated and this can be the subject of a further study for effectiveness of TPD in learning programming.

5.1 Future Work

Before starting to evaluate the system, it is necessary to prepare a documentation of TPD in Turkish. The documentation can be prepared based on the Chapter 3 of this thesis which explains the structure of TPD. But unlike Chapter 3, this documentation is preferred to be in a tutorial style which explains programming concepts using code samples. Preparing a reference manual styled document would not be useful for our target audience.

Some extensions can be made in order to ease the assessment process. For example, a simple client-server module for the development environment can easily collect user programs and analyzing them becomes easier. That is, whenever the user tries to run a program, the source code is sent to a server. These source codes are collected in a database on the server-side. Later, if there is a need to find out the most common user mistakes, a script can run all the programs inserted to the database and results can be processed by the computer.

TPD run-time classes are implemented in order to be able to make further changes with minimal programming effort. This makes re-implementing the system easier.

For example, as explained in Section 3.4.2.1, arrays of TPD are static. But it is possible to make them dynamic by just commenting-out a few lines in the "MyArray" class.

In addition, as mentioned in Section 4.2, implementing a dynamically typed language by using these run-time classes is possible without modifying anything in the libraries. Therefore, if the results of the assessment requires a language without any variable declarations, the programmer does not have to write all the code from scratch. Just modifying the compiler will be enough.

Another example can be adding a run-time type-identification mechanism to the language. That is, the programmer may be able to retrieve type of an expression in run-time. This mechanism can also be added to TPD without re-implementing the libraries.

Run-time classes can also be modified in order to implement a debugger for TPD, if required. The infrastructure for step-by-step execution of the TPD source codes is already implemented. Just by adding a variable watch mechanism, a debugger will be ready to use.

Also Eli provides a lot of features that allows the designer to extend the language without having to change the structure of the current compiler. For example, it is possible to implement a language which supports definition of new operators or to add an include mechanism which allows the users to import functions which are previously implemented.

Finally improving the editor part of the development environment may be a good idea. Adding features such as auto-indentation and auto-keyword-completion may be useful for programmers.

REFERENCES

[1] B. A. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*. Cambridge, MA: The MIT Press, 1993.

[2] A. Bruckman and E. Edwards, "Should we leverage natural-language knowledge? An analysis of user errors in a natural-language-style programming language," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, (Pittsburgh, PA), pp. 207–214, ACM Press, 1999.

[3] J. F. Pane, C. A. Ratanamahatana, and B. A. Myers, "Studying the language and structure in non-programmers' solutions to programming problems," *International Journal of Human Computer Studies*, vol. 54, no. 2, pp. 237–264, 2001.

[4] S. Tutar, C. Bozşahin, and H. Oğuztüzün, "TPD: An educational programming language based on Turkish syntax," in *Proceedings of the 1st Balkan Conference on Informatics (BCI'2003)*, (Thessaloniki, Greece), pp. 650–661, 2003.

[5] J. E. Sammet, "The early history of Cobol," in *The First ACM SIGPLAN Conference on History of Programming Languages*, (Los Angeles, CA), pp. 121–161, ACM Press, 1978.

[6] S. Papert, *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books, 1980.

[7] D. Goodman, *The Complete HyperCard Handbook*. New York: Random House Inc., 1987.

[8] L. K. McIver, *Syntactic and Semantic Issues in Introductory Programming Education*. PhD thesis, Monash University, School of Computer Science and Software Engineering, Australia, 2001.

[9] J. F. Pane, *A Programming System for Children that is Designed for Usability*. PhD thesis, Carnegie Mellon University, School of Computer Science, Wexford, PA, 2002.

[10] P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko, and P. Miller, "Mini-languages: A way to learn programming principles," *Education and Information Technologies*, vol. 2, no. 1, pp. 65–83, 1998.

[11] R. E. Pattis, J. Roberts, and M. Stehlik, *Karel the Robot (2nd ed.): A Gentle Introduction to the Art of Programming*. John Wiley & Sons, Inc., 1994.

[12] B. A. Myers, "Natural Programming: Project overview and proposal," Tech. Rep. CMU-CS-98-1 and CMU-HCII-98-100, Carnegie Mellon University, School of Computer Science, Pittsburgh, 1998.

[13] J.-M. Hoc and A. Nguyen-Xuan, "Language semantics, mental models and analogy," in *Psychology of Programming* (J.-M. Hoc, T. Green, R. Samurçay, and D. Gilmore, eds.), pp. 139–156, London: Academic Press, 1990.

[14] D. C. Smith, A. Cypher, and J. Spohrer, "KidSim: Programming agents without a programming language," *Communications of the ACM*, vol. 37, no. 7, pp. 54–67, 1994.

[15] T. R. G. Green and M. Petre, "When visual programs are harder to read than textual programs," in *Human-Computer Interaction: Tasks and Organisation, Proceedings of ECCE-6 (6th European Conference on Cognitive Ergonomics)* (G. van der Veer, M. Tauber, S. Bagnarola, and M. Antavolits, eds.), (Rome), 1992.

[16] F. Detienne, "Difficulties in designing with an object-oriented language: An empirical study," in *Proceedings of IFIP INTERACT'90: Human-Computer Interaction*, Applications and Case Studies: Programming, (Cambridge, England), pp. 971–976, 1990.

[17] D. Conway and L. McIver, "Seven deadly sins of introductory programming language design," in *Proceedings of Conference on Software Engineering: Education and Practice*, (Los Alamitos, CA), pp. 309–316, 1996.

[18] D. A. Watt, *Programming Language Concepts and Paradigms*. Upper Saddle River, NJ: Prentice-Hall, Inc., 1990.

[19] W. M. Waite, "Beyond LEX and YACC: How to generate the whole compiler," tech. rep., University of Colorado, Boulder, Colorado, 1993.

APPENDIX A

EBNF GRAMMAR OF TPD

EBNF Grammar of TPD (the precedence rules are eliminated in order to simplify the grammar) is as follows:

```
program =
   { definition | declaration | command } ;

definition =
   type_definition |
   function_definition ;

type_definition =
   TUR identifier type_denoter OLSUN "." ;

type_denoter =
   array_type |
   type_id_use |
   record_type |
   function_type |
   basic_type ;

basic_type =
   GERCEL_SAYI |
   DOGAL_SAYI |
   TAMSAYI |
   YAZI |
   KARAKTER |
   DOGRULUK_DEGERI |
   DOSYA |
   LISTE ;

record_type =
   KAYIT "{" field { "," field } "}" ;

field =
   type_denoter identifier ;
```

```
function_type =
   FONKSIYON "(" types [ "->" type_denoter ] ")" ;

types =
   [ type_denoter { "," type_denoter } ] ;

array_type =
   type_denoter "[" [ expression ] "]" ;

function_definition =
   FONKSIYON identifier
      "(" parameters [ "->" type_denoter ] ")" body ;

parameters =
   [ type_denoter identifier
      { "," type_denoter identifier } ] ;

body =
   "{" { definition | declaration | command } "}" ;

declaration =
   SABIT type_denoter identifier "=" expression OLSUN "." |
   identifier { "," identifier } type_denoter OLSUN "." ;

command =
   dotless_command |
   dotted_command "." ;

dot_removed_command =
   dotless_command |
   dotted_command ;

dotless_command =
   block_command |
   while_command |
   for_command |
   conditional_command ;

dotted_command =
   begin_command |
   assignment_command |
   repeat_command |
   return_command |
   break_command |
   continue_command |
   print_command |
   read_command |
   expression_command ;
```

```
begin_command =
   BASLA expression ;

assignment_command =
   expression DEGISKENINI expression YAP ;

block_command =
   "{" { definition | declaration | command } "}" ;

while_command =
   expression [ DOGRU | YANLIS ] OLDUKCA command ;

repeat_command =
      dot_removed_command
   TA_KI expression [ DOGRU | YANLIS ] OLANA_KADAR ;

for_command =
   expression ILE expression ARASINDAKI identifier ICIN
      command ;

conditional_command =
   EGER expression "{" ( case ":" command )+ "}" ;

case =
   expression ISE |
   operator expression ISE |
   DOGRUYSA |
   YANLISSA |
   HICBIRI_DEGILSE ;

return_command =
   expression DEGERINI_VER |
   FONKSIYONDAN_CIK ;

break_command =
   BIRAK ;

continue_command =
   DEVAM_ET ;

print_command =
   ( EKRANA | expression DOSYASINA )
      expression { "," expression } YAZ ;

read_command =
   ( KLAVYEDEN | expression DOSYASINDAN )
      expression { "," expression } OKU ;

expression_command =
```

```
   expression ;

expression =
   expression operator expression |
   operator expression |
   expression operator |
   expression "[" [ expression ] "]" |
   expression "(" arguments ")" |
   identifier "@" expression |
   "(" expression ")" |
   numeric_literal |
   boolean_literal |
   character_literal |
   string_aggregate |
   list_aggregate |
   array_aggregate |
   record_aggregate |
   function_aggregate |
   identifier ;

arguments =
   [ expression { "," expression } ] ;

numeric_literal =
   integer_literal |
   floating_point_literal ;

boolean_literal =
   DOGRU |
   YANLIS ;

list_aggregate =
   "[" [ expression { "," expression } ] "]" ;

array_aggregate =
   DIZI type_denoter
      "(" [ expression { "," expression } ] ")" ;

record_aggregate =
   KAYIT "{" [ field_agg { "," field_agg } ] "}" ;

field_agg =
   type_denoter identifier [ "=" expression ] ;

function_aggregate =
   FONKSIYON "(" parameters [ "->" type_denoter ] ")" body ;

(******* TERMINALS *******)
```

```
LETTER =
   "a" | ...  | "z" | "A" | ...  | "Z" ;

DIGIT =
   "0" | ...  | "9" ;

ALPHA =
   LETTER | DIGIT ;

identifier =
   LETTER { ( "_" ALPHA ) | ALPHA } ;

operator =
   "-" | "+" | "*" | "/" | "//" | "=" | "=/" | "<" | ">" |
   "<=" | ">=" | "ve" | "veya" | "değil" | ":" | "->" |
   "<-" | "bitti" | "bitmedi";

integer_literal =
   DIGIT {DIGIT} ;

floating_point_literal =
   DIGIT {DIGIT} "." DIGIT {DIGIT} ;

character_literal =
   C_CHAR_CONSTANT ;

string_aggregate =
   C_STRING_LIT ;

(******* KEYWORDS *******)

SABIT =
   "sabit" ;
KAYIT =
   "kayıt" ;
FONKSIYON =
   "fonksiyon" ;
TAMSAYI =
   "tamsayı" ;
GERCEL_SAYI =
   "gerçel" "sayı" | "gerçel" ;
DOGAL_SAYI =
   "doğal" "sayı" | "doğal" ;
YAZI =
   "yazı" ;
KARAKTER =
   "karakter" ;
DOGRULUK_DEGERI =
   "doğruluk" "değeri" | "doğruluk" ;
```

```
DOSYA =
   "dosya" ;
LISTE =
   "liste" ;
DEGISKENINI =
   "değişkenini" ;
TUR =
   "tür" ;
OLSUN =
   "olsun" ;
YAP =
   "yap" ;
DOSYASINDAN =
   "dosyasından" ;
KLAVYEDEN =
   "klavyeden" ;
OKU =
   "oku" ;
DOSYASINA =
   "dosyasına" ;
EKRANA =
   "ekrana" ;
YAZ =
   "yaz" ;
OLDUKCA =
   "oldukça" ;
BASLA =
   "başla" ;
DEGERINI_VER =
   "değerini" "ver" | "ver" ;
FONKSIYONDAN_CIK =
   "fonksiyondan" "çık" | "çık" ;
BIRAK =
   "bırak" ;
DEVAM_ET =
   "devam" "et" ;
TA_KI =
   "ta" "ki" ;
OLANA_KADAR =
   "olana" "kadar" ;
EGER =
   "eğer" ;
ISE =
   "ise" ;
HICBIRI_DEGILSE =
   "hiçbiri" "değilse" | "değilse" ;
DOGRUYSA =
   "doğruysa" ;
YANLISSA =
```

```
   "yanlışsa" ;
ILE =
   "ile" ;
ARASINDAKI =
   "arasındaki" ;
ICIN =
   "için" ;
DOGRU =
   "doğru" ;
YANLIS =
   "yanlış" ;
DIZI =
   "dizi" ;
```

APPENDIX B

ENGLISH EQUIVALENTS OF TPD KEYWORDS AND PHRASES

English equivalents of predefined function names, identifiers which represent built-in types, keywords and phrases used in definitions, commands and operators are categorized and listed for the convenience of English speakers.

B.1 Types

`tamsayı`: integer
`gerçel sayı`: floating-point number
`doğal sayı`: natural number
`karakter`: character
`doğruluk değeri`: truth (boolean) value
`yazı`: text, string
`dizi`: array
`liste`: list
`kayıt`: record
`fonksiyon`: function
`dosya`: file

B.2 Operators

`X ve Y`: `X` and `Y`
`X veya Y`: `X` or `Y`
`X değil`: not `X`
`X bitti`: `X` has finished
`X bitmedi`: `X` has not finished
`X doğruysa Y değilse Z`: if `X` is true then `Y` else `Z`

B.3 Commands

`X değişkenini Y yap`: set the variable `X` to `Y`
`X dosyasından Y oku`: read `Y` from file `X`
`klavyeden X oku`: read `X` from keyboard
`X dosyasına Y yaz`: write `Y` to file `X`
`ekrana X yaz`: write `X` to the screen
`X oldukça ...`: while `X` . . .
`X doğru oldukça ...`: while `X` is true . . .

X yanlış oldukça ...: while X is false ...
X değerini ver: give the value of X
başla X: begin X
fonksiyondan çık: exit from the function
bırak: quit, break
devam et: continue
... ta ki X olana kadar: ... until X holds
... ta ki X doğru olana kadar: ... until X is true
... ta ki X yanlış olana kadar: ... until X is false
X ile Y arasındaki Z için ...: for Z between X and Y ...
eğer X {: if X {
  Y ise ...: equals to Y ...
  $<$ Y ise ...: $<$ Y ...
  doğruysa ...: is true ...
  yanlışsa ...: is false ...
  hiçbiri değilse ... }: is none of the above ... }

B.4 Declarations

sabit tamsayı X Y olsun: let constant integer X be Y
X tamsayı olsun: let X be integer
tür X tamsayı olsun: let type X be integer

B.5 Constants

doğru: true
yanlış: false

B.6 Predefined Functions

tavan: ceiling
taban: floor
yuvarla: round
mutlak: absolute
karekök: square root
aç: open
kapat: close
ilki: first (head)
gerisi: rest (tail)