CHAPTER I
INTRODUCTION

I.1 Background of the study

Computers are an integral part of our modern world. Almost every day, applications of computer affect our daily lives. Whether it be in the supermarket, bank, or airport, computers perform more and more of routine data-processing tasks that modern life requires. In government installations, computers keep track of our satellite; in the research laboratories, computers analyze data and help steer the course of research; in business, computers help steer and analyze data used to make informed business decisions; and in schools and colleges, computers are used as an aid for finishing students' assignments.

It is certainly not an exaggeration to say that applications of computers will soon be a part of most jobs. So it is important to learn about computers and to use them to solve problems. The heart of solving problems using a computer is the study of programs and programming. For some applications, it is sufficient to

apply programs that others have written. However, in many other applications (especially the more specific ones), it is necessary to write custom programs.

The goal of a program is to solve a problem. Some problems are rather simple to solve using a program, such as printing out a mailing list or keeping a set of accounting records up-to-date. However, other problems such as calculating the orbit of space satellite, tracking the level of air pollution, or unraveling the chemical structure of a new compound are more difficult and require a large amount of technical knowledge to solve using programs.

I.1.1 Communication

A natural language is a symbolic communication system that is commonly understood among a group of people. Each language has a set of symbols that stand for objects, properties, actions, abstractions, relations, and the like. A language must also have rules for combining these symbols. A speaker can communicate an idea to a listener if and only if they have a common understanding of enough symbols and rules. Communication is impaired when speaker and listener interpret a symbol differently. In this case, either speaker and or listener must use feedback to modify his or her understanding of

the symbols until commonality is actually achieved. This happens when we learn a new word or a new meaning for an old word, or correct an error in our idea of the meaning of a word.

English is for communication among people. Programs are written to be understood by both computers and people. Using a programming language requires a mutual understanding between a person and a machine. This can be more difficult to achieve than understanding between people because machines are so much more literal than human beings.

The meaning of symbols in natural language is usually defined by custom and learned by experience and feedback. In contrast, programming languages are generally defined by an authority, either an individual language designer or a committee. For a computer to `understand' a human language, we must devise a method for translating both the syntax and semantics of the language into machine code. Language designers build languages that they know how to translate, or that they believe, they can figure out how to translate.

On the other, if computers were the only audience for our programs we might write code in a language that was trivially easy to transform into machine code. But a

programmer must be able to understand what he or she is writing, and a human cannot easily work at the level of detail that machine language represents. So we use computer languages that are a compromise between the needs of the speaker (programmer) and listener (computer). Declarations, types, symbolic names, and the like are all concessions to a human's need to understand what someone has written. The concession we make for computers is that we write programs in languages that can be translated with relative ease into machine language. These languages have limited vocabulary and limited syntax. Most belong to a class called *context-free languages* (Fischer 1993:3).

To learn to use a new computer language effectively, a user must learn exactly what combinations of symbols will be accepted by a compiler and what actions will be invoked for each symbol in the language. This knowledge is the required common understanding.

1.1.2. Programming language

A programming language is a notation for expressing instructions to be carried out by a computer. It is a medium of communication between human and the machine and, often, between one human being and another.

Over the years, computers have become more and more `friendly' to human users. This means that computer is made to work more like a human and, consequently, the human is required to make less of an effort to `think like a machine.' Mainly, this increased friendliness depends on the development of software system of increasing size and complexity.

As software systems have increased in size and complexity, a large number of programming languages, systems, and methodologies have been developed in an attempt to reduce or, at least, manage this complexity. Consequently, the study of these programming languages, systems, and methodologies is crucial to those who are involved in the programming function.

I.1.2.1 Low-level programming language

In a computer, all data, whatever its form, is stored and used in binary codes, strings of 1s and 0s. When computers were first developed, the only programming language available was the primitive instruction set built into each machine, the ***machine language***, or ***machine code***.

When programmers used machine language for programming, they had to enter the binary codes for the

various instructions, a tedious process that was prone to error. Moreover, their programs were difficult to read and modify. In time ***assembly languages*** were developed to make the programmer's job easier.

Instructions in an assembly language are an easy-to-remember form called *mnemonic*. Typical instructions for addition and subtraction might look like this (Dale & Weems,1991:9):

Assembly Language	*Machine Language*
ADD	100101
SUB	010011

The only problem with assembly languages was that instructions written in them could not be executed directly by computers. So a program called an ***assembler*** was written to translate the instructions written in assembly language into machine code.

The assembler was a step in the right direction, but programmers still were forced to think in terms of individual machine instructions. Eventually high-level programming languages were developed. These languages are easier to use than assembly languages or machine code because they are closer to English and other natural languages.

I.1.2.2. High-level programming languages

Rather than write a program in assembly language, a programmer can write in any of the hundreds of available computer languages. Among the many popular programming languages are BASIC, Pascal, C, LISP, PROLOG, Modula-2, COBOL, and FORTRAN. These languages allow us to express instructions to carry out major tasks and so typically require less machine language instructions. For this reason, these languages are called ***high-level*** languages, in contrast to machine language, which is a ***low-level*** language.

A programming language has many of the familiar features of natural languages like English. First, a programming language has vocabulary consisting of special words and symbols. These special words are called ***reserved words***. The following are some of the reserved words of one programming language, Pascal:

and or for not with case begin end

Pascal makes special use of the punctuation marks: " ; " and " . " It also assigns special meanings to other symbols such as " * " and " < > ". These all are examples of ***reserved symbols*** in Pascal.

In natural language, one important unit of communication is the sentence. In a computer programming language, the role of a sentence is played by the ***statement***. A statement is a sequence of words and symbols that expresses an instruction for the computer. As in natural languages, statements must be constructed according to certain rules of ***syntax*** that govern the usage and arrangement of words and symbols.

I.2. Statement of the problem

As stated above, statements in a programming language must be constructed according to certain rules of syntax that govern the usage and arrangement of words and symbols. A programmer to make a program needs an understanding of the syntax of the programming language he uses since computer does not tolerate an error. Related to this, the problem that will be solved in this thesis is:

How is the syntax of Pascal language?

I.3. Objective of the study

The objective of this study is to describe how the syntax of Pascal language is.

I.4. Significance of the study

By describing the syntax of Pascal language, the writer hopes it can be a contribution to linguistics and useful for the readers to broaden their knowledge on programming language; a language that is used to make a computer program.

Furthermore, the results are expected to contribute to the studies on syntax and other related studies.

I.5. Theoretical framework

I.5.1 Metalanguage

Programmers develop solutions to problems using a programming language. A programming language is a set of rules, symbols, and special words used to construct a program. There are rules for both ***syntax*** (grammar) and ***semantics*** (meaning).

Syntax is a formal set of rules that defines exactly what combinations of letters, numbers, and symbols can be used in a programming language. There is no place for ambiguity in the syntax of a programming language because computer cannot think;it does not ``know what we mean.'' To avoid ambiguity, syntax rules themselves must be written in a very simple, precise, formal language called

a ***metalanguage*** (Dale and Weems,1991:38). ***Metalanguage*** is the ***language*** with the prefix ***meta***, which means ``beyond.'' A metalanguage is a language that goes beyond a normal language by allowing us to speak precisely about that language. It is a language for talking about languages.

I.5.2 Extended Backus-Naur Form (EBNF)

One of the oldest computer-oriented metalanguages is the **Backus-Naur Form (BNF)**, which is named after John Backus and Peter Naur, who developed it in 1960. The original BNF formalism has since been extended and streamlined; a generally accepted version, named ***Extended*** **BNF** (Dale and Weems,1991:39;Fischer 1993:76). According to Fischer (1993:77) an EBNF grammar consist of :

- A starting symbol.
- A set of terminal symbols, which are the keywords and syntactic markers of the language being defined.
- A set of nonterminal symbols, which correspond to the syntactic categories and kinds of statements of the language.
- A series of rules, called productions, that specify how each nonterminal symbol may be expanded into a phrase containing terminals and

nonterminals. Every nonterminal has one production rule, which may contain alternatives.

I.5.3 The syntax of EBNF

The syntax for EBNF itself is not altogether standardized; several minor variations exist. Here is a commonly used version (Fischer,1993:77-78).

- **The *starting symbol*** must be defined. One nonterminal is designated as the starting symbol.
- **Terminal *symbols*** will be written in ***boldface*** and enclosed in 'single quotation mark'.
- **Nonterminal *symbols*** will be written in regular type and enclosed in <angle brackets>.
- **Production rules**. The nonterminal being defined is written at the left, followed by a " ::= " sign (which we will pronounce as ``is defined as''). After this is the string, with options, which defines the nonterminal. The definition extends up to but does not include the " . '' that marks the end of the production. When a nonterminal is *expanded* it is replaced by this defining phrase. Blank spaces between the " ::= `` and the " . '' are ignored.
- **Alternatives** are separated by vertical bars. Parentheses may be used to indicate grouping. For example, the rule ***s*** ::= **(*a* | *bc*) *d*** indicates that

an ` *s* ' may be replaced by an ` *ad* ' or a ` *bcd* '.

- An *optional syntactic element* is a something-or-nothing alternative- it may be included or not included as needs demand. This is indicated by enclosing the optional element in square brackets, as follows: *s*::=[*a*]*d*. This formula indicates that an ' *s* ' may be replaced by an ` *ad* ' or simply by a ` *d* '.
- An *unspecified number of repetitions* (zero or more) of a syntactic unit is indicated by enclosing the unit in curly brackets. For example, the rule *s*::={*a*}*d*. Indicates that an ` *s* ' may be replaced by a ` *d* ', an ` *ad* ', an ` *aad* ', or a string of any number of ` *a* 's followed by a single ` *d* '. A frequently occurring pattern is the following: *s*::=*t*{*t*}. This means that ` *s* ' may be replaced by *one or more* copies of ` *t* '.
- *Recursive production rules* are permitted. For example, this rule is directly recursive because its right side contains a reference to itself: *s*::=*asz*|*w*. This expands into a single ` *w* ', surrounded on the left and right by any number of matched pairs of ` *a* ' and ` *z* ': *awz*, *aawzz*, *aaawzzz*, etc.
- *Tail recursion* is a special kind of recursion in which the recursive reference is the last symbol in the string. Tail recursion has the

same effect as a loop. This production is tail recursive: *s::=as | b*. This expands into a string of any number of ` *a* 's followed by a ` *b* '.

- **Mutually recursive rules** are also permitted. For example, this pair of rules is mutually recursive because each rule refers to the other: *s::=at | b t::=bs | a*. A single ` *s* ' could expand into any of the following: *b, aa, abb, abaa, ababb, ababaa,* etc.
- ***Combinations of alternatives, optional elements, recursions, and repetitions*** often occur in a production, as follows: *s::={a | b}[c] d*. This rule indicates that an ` *s* ' may be replaced by any of the following:*d, ad, bd, cd, acd, bcd, aad, abd, aacd, abcd, bd, bad, bbd, bcd, bacd, bbcd,* and many more.

For example, an integer number in Pascal must be at least one digit, may or may not be more than one digit, and may or may not have a sign in front of it. The EBNF definition of an integer number in Pascal is

<integer>::= <unsigned-integer> | <sign> <unsigned-integer>
<sign>::= + | -
<unsigned-integer>::= <digit> | <digit> <unsigned-integer>
<digit>::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

The first line of the definition reads:''An integer is defined as an unsigned integer or a sign followed by

an unsigned integer.'' This line contains nonterminal symbol that must be defined. In the second line, the nonterminal symbol <sign> is defined as a plus sign or a minus sign, both of which are terminal symbols. The third line defines the nonterminal symbol <unsigned-integer> as either a <digit> or a <digit> followed by another <unsigned-integer>. The self-reference in the definition is a round about way of saying that an <unsigned integer> can be a series of one or more <digit>s. In the last line, <digit> is defined as any one of the numeric characters 0 through 9.

1.6. Method of the study

1.6.1 The research design

The method of research in this study is the explorative descriptive. By doing this research the writer wants to get as many data as possible, so they can help to solve the problem that will be discussed.

1.6.2. Technique of data collection

The technique of data collection used in this study is library research, since Pascal is not a spoken language, but a written one. The user of this language only use it to create a computer program. So the only way

to collect the data is to find the computer programs which are written in Pascal.

1.6.3. Technique of data analysis

The technique of data analysis used in this study is descriptive analysis. The writer will analyze the program part by part, statement by statement, then formulate the syntax of the statement using the EBNF.

CHAPTER II

GENERAL DESCRIPTION OF THE OBJECT OF THE STUDY