Most of the computer programming systems have been designed to allow the programmers to express their algorithms in concise languages which have been subsets of the English language. These programming systems, through which most of the productive computer work in English speaking societies is done presently, were designed without the software knowledge and the hardware advances of today, they are the most limiting factors in the software development.

Iranians who have not invested in software development as largely as the English speaking societies, should not face the difficulties in a change over to more advanced and effective programming systems to the same extent that the English speaking societies do.
THE INTERACTIVE COMPUTER PROGRAMMING SYSTEM
IN FARSI

is designed so the Iranian manufacturing, scientific, engineering, governmental, business, and cultural organizations, and even high school and grammar school students can solve their problems by preparing computer programs and communicating with computers using a subset of their own language; providing them with a system which is the product of today's knowledge and technology and should make FORTRAN, COBOL, BASIC, and a number of other heavily used programming systems of scholarly interest only in Iran.

The system is composed of the following major subsystems:

* The Software Simulated Input/Output Device Program Package (SSIODPP),
* The Interactive Text Editor/String Processing Language, and
* The General Purpose Interactive Computer Programming Language.

SSIODPP is designed for transmission of and full ASCII character sets on a typical computer graphic terminal: Tektronix T4002. The techniques used in SSIODPP suggest the preliminary aspects of the design of Farsi hardware character generators and Farsi character set pattern recognition systems; it can be modified to suggest
is designed for creation, modification and display of files. It is to be used conversationally from a remote terminal so the user may monitor processing one command at a time. Nevertheless, it can be operated on nonconversational basis by submitting tasks.

provides a programming language which is machine independent (assuring transferability), relatively easy to learn and use (assuring popularity), structurally powerful in terms of problem solving (assuring effectiveness), readable in the sense that a reasonably good understanding of the program can be obtained by reading a listing of the source program (assuring the ease of documentation, and as the result, providing ease with which the programs can be extended, modified, and maintained).
PEESHTAZ  The Interactive Computer Programming System in Farsi

by

Shahram Shawn Ayromloo

A THESIS submitted to

Oregon State University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Completed September 14, 1977

Commencement June 1978
APPROVED:

Redacted for Privacy

Professor of Mathematics and Computer Science in charge of major

Redacted for Privacy

Chairman of the Department of Computer Science

Redacted for Privacy

Dean of Graduate School

Date thesis is presented September 14, 1977

Typed by Lora Wixom for Shahram Shawn Ayromloo
ACKNOWLEDGMENT

I want to express my sincere thanks to my major advisor (Dr. Harry E. Goheen) for all that he has taught me not only in Computer Science and Mathematics, but also in social, cultural and philosophical matters. Without his constant encouragement and support in the past ten years, I may have not been able to accomplish all that I have.

Also, I want to express my gratitude to my parents for all their love, trust and pride in my ability which strengthened my dedication and boosted my progress.

Finally, I want to thank the Iranian Government for the financial support, the Chairman of the Computer Science Department (Dr. Robert A. Short) for providing me with the teaching assistantship during the first three years of my doctoral work, and my other doctoral committee members (Dr. Larry C. Hunter, Mr. Thomas L. Yates and Dr. Edwin L. Anderson), and my typist (Mrs. Lora D. Wixom).
DEDICATED TO

all the Iranians,

Dr. H. E. Goheen,

and

the other citizens of the United States

who helped me
# TABLE OF CONTENTS

1. ALGORITHMS AND THEIR EXECUTING MODELS ............... 1
   1.1 Algorithms ........................................ 1
      1.1.1 The Origin of Algorithm ....................... 1
      1.1.2 The Meaning of Algorithm ...................... 2
   1.2 The Elements of Computing .............................. 3
   1.3 The Executing Models .................................. 3
      1.3.1 Finite State Machines
         (Finite Automata) ................................ 5
         1.3.1.1 Finite State Machines
            Classification .................................. 8
         1.3.1.2 Combinational Networks ...................... 8
            1.3.1.2.1 Machine Realization .................. 9
               1.3.1.2.1.1 Gate Realization ................. 17
               1.3.1.2.1.2 Branch Realization ............ 25
            1.3.1.2.2 Spatial Iterative Networks ............ 28
         1.3.1.3 Sequential Machines ......................... 31
            1.3.1.3.1 Sequential Machines
               Classification ................................ 39
               1.3.1.3.1.1 Clock Mode Sequential
                  Machines ................................... 39
               1.3.1.3.1.2 Pulse Mode Sequential
                  Machines ................................... 40
               1.3.1.3.1.3 Level Mode Sequential
                  Machines ................................... 40
            1.3.1.4 Computational Limitation .................. 41
            1.3.1.5 Conclusion ................................ 51
      Bibliography ........................................ 53
   1.3.2 Infinite Machines .................................... 54
      1.3.2.1 Turing Machine ............................... 55
      1.3.2.2 Post Machine ................................ 62
         1.3.2.2.1 Wang Version of Post
            Machine ...................................... 63
         1.3.2.2.2 Shepherdson and Sturgis
            Version of Post Machine ....................... 63
      1.3.2.3 Minsky Machine ............................... 64
      1.3.2.4 Markov Normal Algorithm ...................... 65
      1.3.2.5 Transformation of One Machine
         to Another Machine ............................... 69
      1.3.2.6 Universal Turing Machine .................... 72
      1.3.2.7 Computational Limitation .................... 72
      1.3.2.8 Conclusion ................................ 73
      Bibliography ........................................ 76
2. COMPUTER AND THE EVOLUTIONARY DEVELOPMENT OF PROGRAMMING LANGUAGES

2.1 Computer

2.1.1 Definition of Computer

2.1.2 Historical Perspective of Computer

2.1.2.1 Historical Survey of 0 Generation Technology

2.1.2.1.1 Gerbert

2.1.2.1.2 Magnus

2.1.2.1.3 Al-Kashi

2.1.2.1.4 Napier

2.1.2.1.5 Gunther

2.1.2.1.6 Schickard

2.1.2.1.7 Pascal

2.1.2.1.8 Morland

2.1.2.1.9 Leibniz

2.1.2.1.10 Roven

2.1.2.1.11 Poleni

2.1.2.1.12 Stanhope

2.1.2.1.13 Boole

2.1.2.1.14 Jacquard

2.1.2.1.15 Babbage

2.1.2.1.16 Hollerith

2.1.2.2 Historical Survey of 1st Generation and Thereafter Technology

2.1.2.2.1 Z1 and Z3

2.1.2.2.2 Atanosoff-Berry

2.1.2.2.3 MARK I

2.1.2.2.4 ENIAC

2.1.2.2.5 EDVAC

2.1.2.2.6 EDSAC

2.1.2.2.7 MADM

2.1.2.2.8 SEAC

2.1.2.2.9 ERA 1101

2.1.2.2.10 IAS

2.1.2.2.11 UNIVAC

2.1.2.2.12 WhirlWind I

2.1.2.3 Important Milestones

Bibliography

2.1.3 Classification of Computer

2.1.4 Structure of Computer

2.2 Programming Languages

2.2.1 Instruction Repertoire of Computer

2.2.2 Machine Language Programming

2.2.3 High-Level Language Machines

2.2.4 Intermediate Languages

2.2.5 Assembly Language Programming

2.2.5.1 Assembler
2.2.6 High-Level Languages............................ 140
  2.2.6.1 Compiler........................................ 142
    2.2.6.1.1 Scanner/Lexical Analyzer................. 143
    2.2.6.1.2 Parser................................. 144
    2.2.6.1.3 Optimizer.............................. 144
    2.2.6.1.4 Code Generator......................... 145
  Bibliography........................................ 146
3. THE INTERACTIVE COMPUTER PROGRAMMING SYSTEM IN Farsi

3.1 Design Justification

3.2 The Software Simulated Input/Output Device

3.2.1 Decomposition of Farsi Characters
   Into Sub-Patterns
   3.2.1.1 Reduction of Farsi Character Set
   3.2.1.2 Classification of Farsi Characters
   3.2.1.3.1 Decomposition Expressions of Farsi Characters
      3.2.1.3.1.1 Cluster 1 Decomposition
      3.2.1.3.1.2 Cluster 2 Decomposition
      3.2.1.3.1.3 Cluster 3 Decomposition
      3.2.1.3.1.4 Cluster 4 Decomposition
      3.2.1.3.1.5 Cluster 5 Decomposition
      3.2.1.3.1.6 Cluster 6 Decomposition
      3.2.1.3.1.7 Cluster 7 Decomposition
      3.2.1.3.1.8 Cluster 8 Decomposition
      3.2.1.3.1.9 Cluster 9 Decomposition
      3.2.1.3.1.10 Cluster 10 Decomposition
      3.2.1.3.1.11 Cluster 11 Decomposition
      3.2.1.3.1.12 Cluster 12 Decomposition
      3.2.1.3.1.13 Cluster 13-19 Decomposition
      3.2.1.3.1.14 Cluster 20 Decomposition
      3.2.1.3.1.15 Cluster 21 Decomposition
      3.2.1.3.1.16 Cluster 22 Decomposition
   3.2.1.3.2 The Advantages of the Proposed Decomposition Technique

3.2.2 The Internal Representation of Character Set

3.2.3 Character Transmission Algorithm

3.2.4 Software Simulated Input/Output Device Components
   3.2.4.1 Software Simulated Character Generators
   3.2.4.2 Character Generator Program Writer (CGPW)
3.2.4.3 Input/Output Driver

3.3 the Interactive Text Editor/String Processing Language of PEESHTAZ

3.3.1 Functional Description
3.3.2 Notation used in Command Statements
3.3.3 Special Terms
3.3.4 The Externally-Apparent Structure of BYGONE
3.3.5 The Externally-Apparent Structure of Text in BYGONE

3.3.6 Commands
3.3.6.1 The Structure of Commands
3.3.6.1.1 Parameters
3.3.6.1.2 Components of a Command Statement
3.3.6.1.2.1 Modes
3.3.6.1.2.2 Verbs
3.3.6.2 Data Manipulation Commands
3.3.6.2.1
3.3.6.2.2
3.3.6.2.3
3.3.6.2.4
3.3.6.2.5
3.3.6.2.6
3.3.6.2.7
3.3.6.3 File Management Commands
3.3.6.3.1 Files
3.3.6.3.1.1 Sequential Files
3.3.6.3.1.2 Random Access Files
3.3.6.3.2
3.3.6.3.3
3.3.6.3.4
3.3.6.3.5
3.3.6.3.6
3.3.6.3.7
3.3.6.3.8 System/Man Information Interchange
Commands .............................................. 337
3.3.6.4.1 System-Defined Data .......................... 337
3.3.6.4.2 System-Defined Data .......................... 339
3.3.6.4.3 System-Defined Data .......................... 341
3.3.6.4.4 System-Defined Data .......................... 342
3.3.6.4.5 System-Defined Data .......................... 343
3.3.6.4.6 System-Defined Data .......................... 344
3.3.6.4.7 System-Defined Data .......................... 345
3.3.6.4.8 System-Defined Data .......................... 346
3.3.6.4.9 System-Defined Data .......................... 347
3.3.6.4.10 System-Defined Data ......................... 348
3.3.6.7 The Internal Structure of BYGONE .......... 350
3.3.6.7.1 Text Memory Organization .................. 351
3.3.6.8 Command Processor ............................ 357
3.4 The General Purpose/Interactive
Programming Language of PEESHTAZ .................. 358
3.4.1 Data and Its Control ............................. 358
3.4.1.1 Programmer-Defined Data ....................... 360
3.4.1.2 System-Defined Data ............................ 360
3.4.1.3 Primitive-Data Structures ....................... 360
3.4.1.4 Non-Primitive Structures ....................... 361
3.4.1.5 Simple Variables ............................... 361
   3.4.1.5.1 Physical Positioning
       (Environment): INTERNAL and
       EXTERNAL Attributes ........................... 363
   3.4.1.5.2 Life Expectancy (Persistence):
       PERMANENT, SEMI-PERMANENT and
       TEMPORARY Attributes ........................ 364
3.4.1.5.3 Popularity (Scope): PUBLIC, PRIVATE and SEMI-PUBLIC Attributes..........................365
3.4.1.5.4 READ ONLY..........................365
3.4.1.5.5 INTEGER..........................366
3.4.1.5.6 REAL..........................366
3.4.1.5.7 COMPLEX..........................367
3.4.1.5.8 CHARACTER..........................367
3.4.1.5.9 LOGICAL..........................367
3.4.1.5.10 LABEL..........................368
3.4.1.5.11 PROCEDURE..........................368
3.5.1.5.12 POINTER..........................368
3.4.1.5.13 Simple Variable Declaration.....369
3.4.1.5.14 Simple Variable Descriptor.....373
3.4.1.5.15 Simple Variable Accessing......375
3.4.1.5.16 Simple Variable Initialization..........................378

3.4.1.6 Organized Group of Simple Variables..........................380

3.4.1.7 Homogeneous n-Dimensional Array (سلاسل)..........................380
3.4.1.7.1 Homogeneous Array Declaration..........................381
3.4.1.7.2 Homogeneous Array Descriptor..........................385
3.4.1.7.3 Homogeneous Array Accessing..........................388
3.4.1.7.4 Homogeneous Array Initialization..........................391

3.4.1.8 Heterogeneous n-Dimensional Array (كلمات)..........................393
3.4.1.8.1 Heterogeneous Array Declaration..........................396
3.4.1.8.2 Heterogeneous N-Dimensional Array Descriptor..........................398
3.4.1.8.3 Heterogeneous Array Accessing..........................404
3.4.1.8.4 Heterogeneous Array Initialization..........................406

3.4.2 Program Structure..........................408
3.4.2.1 Program Execution..........................410
3.4.2.2 Main Program/
Modules Communication..........................411
3.4.2.3 Insertion of a Set of
Instructions Within a Program..........................412
3.4.2.4 Procedure Structure..........................412
3.4.2.4.1 Procedure/Main Program
Information Exchange..........................416
3.4.2.4.1.1 Arguments/Formal
Parameters Dynamic
Relationships..........................419
3.4.2.4.1.1.1 Call by Name..........................419
3.4.2.4.1.1.2 Call by Value........420
3.4.2.4.1.1.3 Call by Reference........421
3.4.2.4.1.1.4 Call by Name-and-
Value........422
3.4.3 Arithmetics..........................425
3.4.3.1 Operators......................425
3.4.3.2 Operands.......................427
3.4.3.3 Constants.........................427
3.4.3.3.1 INTEGER/..........................429
3.4.3.3.2 REAL ( ) Constants........430
3.4.3.3.2.1 REAL FIXED POINT........430
3.4.3.3.2.2 REAL FLOATING POINT
........430
3.4.3.3.3 COMPLEX/.....................435
3.4.3.3.4 LOGICAL/......................436
3.4.3.3.5 CHARACTER/.....................436
3.4.3.3.6 LABEL/PROCEDURE/...........437
3.4.3.3.7 POINTER/......................437
3.4.3.3.8 Library Functions...........438
3.4.3.4 General Form and Definition
or Arithmetic Statement.................441
3.4.3.4.1 ROUNDED/.....................442
3.4.3.4.2 SIZE ERROR/....................442
3.4.3.4.3 OVERFLOW/UNDERFLOW/........442
3.4.3.4.4 Precision of Arithmetic
Results........................444
3.4.3.4.5 Generic Operations...........445
3.4.3.4.6 Type Attribute Conversion
Functions........................449
3.4.3.4.7 Operator Precedence...........449
3.4.4 Input/Output.......................450
3.4.4.1 Acquisition.......................451
3.4.4.1.1 Clause.........................454
3.4.4.1.2 Format String for Data
Acquisition.........................454
3.4.4.1.3 Clause..............................................458
3.4.4.1.4 Clause..............................................458
3.4.4.2 File.....................................................462
3.4.4.2.1 Opening.............................................462
3.4.4.2.2 Closing.............................................464
3.4.4.2.3 Manipulation........................................464
3.4.4.3 Disposition...........................................465
3.4.5 Execution Control......................................469
3.4.5.1 Conditional Statements...............................469
3.4.5.1.1 Scope Terminator of Conditional Statements......472
3.4.5.1.2 Nested Conditional Statements.....................473
3.4.5.2 Unconditional Transfer Statements.....................476
3.4.5.2.1 Unconditional GOTO................................476
3.4.5.2.2 Indexed GOTO.......................................476
3.4.5.2.2.1 Out-of-Range Index of an Indexed GOTO........477
3.4.5.2.3 Multiple-Route GOTO...............................478
3.4.5.2.3.1 Implicit Indexing of Constituents...............480
3.4.5.2.3.2 Explicit Indexing of Constituents...............480
3.4.5.2.3.3 Constituents with Several Values...............481
3.4.5.2.3.4 Constituents with Index Ranges...............482
3.4.5.2.3.5 Explicit Predicates.............................482
3.4.5.3 Return of Control from Procedures...................483
3.4.5.4 Termination of Execution.............................483
3.4.5.5 Iteration Statements................................483
3.4.5.5.1 Various Loop Structures..........................484
3.4.5.5.1.1 Test at the Beginning (WHILE Statement)......488
3.4.5.5.1.2 Test at End (REPEAT-UNTIL)....................490
3.4.5.5.1.3 Test in the Middle.............................491
3.4.5.5.2 Control Variable in Indefinite Iterations.......492
3.4.5.5.2.1 Control Variable Values........................492
3.4.5.5.2.2 Subscripted Control Variable....................492
3.4.5.5.2.3 Control Variable Changeability...............493
3.4.5.5.2.4 Control Variable Accessability.................493
3.4.5.5.2.5 Initial Value of Control Variable .......... 493
3.4.5.5.2.6 Increment/Decrement .......... 493
3.4.5.5.2.7 Termination Value .......... 494
3.4.5.5.3 Scope Terminator of Iteration Statements .......... 494
Bibliography ......................... 495

3.5 Conclusion ................................................. 498
Bibliography ................................................. 499
### LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.1</td>
<td>The Computing Environment.</td>
<td>4</td>
</tr>
<tr>
<td>1.3.1.1</td>
<td>A General Picture of a Finite State Machine.</td>
<td>5</td>
</tr>
<tr>
<td>1.3.1.1.1</td>
<td>Finite State Machines Classification</td>
<td>8</td>
</tr>
<tr>
<td>1.3.1.2.1.1</td>
<td>Different Combinational Machines with (n) Input Variables and Single Outputs.</td>
<td>10</td>
</tr>
<tr>
<td>1.3.1.2.1.2</td>
<td>The Growth of Function (2^n).</td>
<td>10</td>
</tr>
<tr>
<td>1.3.1.2.1.3</td>
<td>All the Possible Functions of a Single Variable.</td>
<td>11</td>
</tr>
<tr>
<td>1.3.1.2.1.4</td>
<td>All the Functions of Two Variables.</td>
<td>11</td>
</tr>
<tr>
<td>1.3.1.2.1.5</td>
<td>Conjunction (AND) Gate.</td>
<td>12</td>
</tr>
<tr>
<td>1.3.1.2.1.6</td>
<td>Disjunction (inclusive-OR) Gate.</td>
<td>12</td>
</tr>
<tr>
<td>1.3.1.2.1.7</td>
<td>Realization of All the Two Input Variable Machines.</td>
<td>13</td>
</tr>
<tr>
<td>1.3.1.2.1.8</td>
<td>Truth Table and the Fundamental Products (minterms) of an (n)-Input Variable Machine.</td>
<td>14</td>
</tr>
<tr>
<td>1.3.1.2.1.9</td>
<td>NAND Operation.</td>
<td>15</td>
</tr>
<tr>
<td>1.3.1.2.1.10</td>
<td>The Minimally Complete Set of Primitive Operations.</td>
<td>16</td>
</tr>
<tr>
<td>1.3.1.2.1.1.1</td>
<td>A Systematic Procedure for Realization of a Switching Function.</td>
<td>18</td>
</tr>
<tr>
<td>1.3.1.2.1.1.2</td>
<td>From AND/OR Realization to its Equivalent NAND Realization.</td>
<td>23</td>
</tr>
<tr>
<td>1.3.1.2.1.1.3</td>
<td>The NAND Realization of the Example in 1.3.1.2.1.1.</td>
<td>24</td>
</tr>
<tr>
<td>1.3.1.2.1.1.4</td>
<td>The NOR Realization of the Example in 1.3.1.2.1.1.</td>
<td>24</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>1.3.1.2.1.2.1</td>
<td>The Serial-parallel Branch Contacts of the Example in 1.3.1.2.1.1.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.2.1.2.2</td>
<td>Basic Relay Types.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.2.1.2.3</td>
<td>Relay Realization of AND, OR, and NOT Operations.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.2.2.1</td>
<td>Spatial Iterative Combinational Network.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.2.2.2</td>
<td>An Example of Iterative Combinational Network.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.1</td>
<td>General Sequential Machine.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.2</td>
<td>A Sequential Parity.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.3</td>
<td>The General Procedure for the Realization of Sequential Machines.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.4</td>
<td>Realization of a Sequential Machine (which outputs a 1 whenever it sees three successive 1's).</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.5</td>
<td>T-flip-flops and J-K-flip-flops Definition.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.6</td>
<td>The Realization of the Machine Described on Page 36, Using J-K flip-flops.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.7</td>
<td>The Realization of the Machine Described on Page 36, Using T-flip-flops.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.1.1.1</td>
<td>General Model of Clock Mode Sequential Circuit.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.3.1.3.1</td>
<td>General Model of Level Mode Sequential Machines.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.4.1</td>
<td>Regular Expressions and Their Undeterministic Transition Graphs.</td>
<td></td>
</tr>
<tr>
<td>1.3.1.4.2</td>
<td>The Derivation of Deterministic Transition Graph from an Undeterministic Transition Graph.</td>
<td></td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1.3.1.4.3</td>
<td>Design of a Minimal State Machine from a Deterministic Transition Graph.</td>
<td>47</td>
</tr>
<tr>
<td>1.3.1.4.4</td>
<td>A Machine and its Synchronizing Sequence.</td>
<td>48</td>
</tr>
<tr>
<td>1.3.1.4.5</td>
<td>A Machine and Its Homing Sequences.</td>
<td>49</td>
</tr>
<tr>
<td>1.3.1.4.6</td>
<td>A Machine and Its Preset Distinguishing Experiment.</td>
<td>50</td>
</tr>
<tr>
<td>1.3.2.1.1</td>
<td>Components of a Turing Machine.</td>
<td>55</td>
</tr>
<tr>
<td>1.3.2.1.2</td>
<td>The Turing Machine.</td>
<td>60</td>
</tr>
<tr>
<td>1.3.2.2.1</td>
<td>The Set of Instructions Within Post's System.</td>
<td>62</td>
</tr>
<tr>
<td>1.3.2.2.1.1</td>
<td>The Simplified Version of the Set of Instructions in Figure 1.3.2.2.1.</td>
<td>63</td>
</tr>
<tr>
<td>1.3.2.2.1</td>
<td>Shepherdson and Sturgis Version of Turing Machine.</td>
<td>64</td>
</tr>
<tr>
<td>1.3.2.3.1</td>
<td>Instructions on Minsky Machine.</td>
<td>64</td>
</tr>
<tr>
<td>1.3.2.4.1</td>
<td>The Normal Algorithm.</td>
<td>67</td>
</tr>
<tr>
<td>1.3.2.4.2</td>
<td>The Normal Algorithm.</td>
<td>68</td>
</tr>
<tr>
<td>1.3.2.5.1</td>
<td>A Turing Machine.</td>
<td>69</td>
</tr>
<tr>
<td>2.1.2.1.3.1</td>
<td>Al-Kashi's Instrument.</td>
<td>82</td>
</tr>
<tr>
<td>2.1.2.1.3.2</td>
<td>Al-Kashi's Instrument Setup for Finding the Sun's True Longitude.</td>
<td>83</td>
</tr>
<tr>
<td>2.1.2.1.3.3</td>
<td>Al-Kashi's Instrument Setup for Finding a Lunar Eclipse.</td>
<td>83</td>
</tr>
<tr>
<td>2.1.2.1.4.1</td>
<td>Napier's Bones.</td>
<td>84</td>
</tr>
<tr>
<td>2.1.2.1.6.1</td>
<td>Schickard Calculating Machine.</td>
<td>85</td>
</tr>
<tr>
<td>2.1.2.1.7.1</td>
<td>Pascal's Calculating Machine.</td>
<td>86</td>
</tr>
<tr>
<td>2.1.2.1.8.1</td>
<td>Morland's Adding Machine.</td>
<td>87</td>
</tr>
<tr>
<td>2.1.2.1.8.2</td>
<td>Morland's Multiplying Machine.</td>
<td>88</td>
</tr>
<tr>
<td>2.1.2.1.9.1</td>
<td>Leibniz's Calculating Machine.</td>
<td>89</td>
</tr>
<tr>
<td>2.1.2.1.14.1</td>
<td>Jacquard's Automated Loom.</td>
<td>91</td>
</tr>
</tbody>
</table>
2.1.2.1.15.1 Babbage's Difference Engine.  
2.1.2.1.15.2 Babbage's Analytical Engine.  
2.1.2.1.16.1 Hollerith's Tabulating Machines and Sorting.  
2.1.4.1 General Hardware Organization of a Computer.  
2.2.1.1 The Basic Instructions Within the Repertoire of a Typical Computer.  
2.2.1.2 The Components of the Turing Commands Encoder.  
2.2.1.3 Basic Instructions Sufficient to Express Any Algorithm.  
2.2.1.4 The Algorithm to Simulate the Multiplication Process.  
2.2.2.1 Machine Instructions from Figure 2.2.1.1 and Their Corresponding Operations Codes.  
2.2.2.2 Machine Language Program Using the Instruction Set in Figure 2.2.1.1.  
2.2.2.3 The Algorithm for the Machine Language of Figure 2.2.2.2.  
2.2.5.1 Machine Instructions Operation Codes and Their Corresponding Mnemonic Symbols.  
2.2.5.2 Symbolic Machine Language Program Equivalent to the Machine Language Program in Figure 2.2.2.2.  
2.2.5.1.1 Assembly Phase.  
2.2.5.1.2 Pass One Flowchart Outline.  
2.2.5.1.3 Data Flow for Pass One.  
2.2.5.1.4 Data Flow for Pass Two.  
2.2.5.1.5 Pass Two Flowchart Outline.  
2.2.6.1 An Example of a High-Level Language Program.  
2.2.6.1.1 The Major Components of a Typical Compiler.
3.1.1 The Major Subsystems/User Network. 151
3.2.1.1.1 The Detailed Set of Farsi Characters - 133 Symbols. 154
3.2.1.1.2 The Minimal Set of Farsi Characters - 70 Symbols. 155
3.2.1.2.1 The Clusters of the Characters in Which Certain Similarities Exist - 34 Clusters. 155
3.2.1.2.2 The Minimal Set of Clusters - 22 Clusters. 156
3.2.1.3.2.1 The Character Set of PEESHTAZ. 171
3.2.2.1 The PEESHTAZ/ASC11 Character Set Internal Codes. 173
3.2.2.2 ASC11 and PEESHTAZ Character Set Arrangement on TEKTRONIX T4002. 174
3.2.2.3 PEESHTAZ Terminal Keyboard. 175
3.2.3.1 Character Transmission Algorithm. 176
3.2.4.2.1 The General Algorithm of CGPW. 179
3.2.4.2.2 Character Generator Program Writer. 180
3.2.4.2.3 The Decomposition Expressions/The Input to CGPW. 182
3.2.4.2.4 The Software Simulated Character Generators/The Output of CGPW. 185
3.2.4.3.1 The General Algorithm of IODRV. 263
3.2.4.3.2 Input/Output Drifter. 264
3.2.4.3.3 The Output of IODRV. 275
3.3.1.1 The Modes of Operation and Manipulatable Text Components. 278
3.3.3.1 Vertical and Horizontal Pointers Within BYGONE. 289
3.1.1 The Major Subsystems/User Network. 151
3.2.1.1.1 The Detailed Set of Farsi Characters - 133 Symbols. 154
3.2.1.1.2 The Minimal Set of Farsi Characters - 70 Symbols. 155
3.2.1.2.1 The Clusters of the Characters in Which Certain Similarities Exist - 34 Clusters. 155
3.2.1.2.2 The Minimal Set of Clusters - 22 Clusters. 156
3.2.1.3.2.1 The Character Set of PEESHTAZ. 171
3.2.2.1 The PEESHTAZ/ASC11 Character Set Internal Codes. 173
3.2.2.2 ASCII and PEESHTAZ Character Set Arrangement on TEKTRONIX T4002. 174
3.2.2.3 PEESHTAZ Terminal Keyboard. 175
3.2.3.1 Character Transmission Algorithm. 176
3.2.4.2.1 The General Algorithm of CGPW. 179
3.2.4.2.2 Character Generator Program Writer. 180
3.2.4.2.3 The Decomposition Expressions/The Input to CGPW. 182
3.2.4.2.4 The Software Simulated Character Generators/The Output of CGPW. 185
3.2.4.3.1 The General Algorithm of IODRV. 263
3.2.4.3.2 Input/Output Drifter. 264
3.2.4.3.3 The Output of IODRV. 275
3.3.1.1 The Modes of Operation and Maniputable Text Components. 278
3.3.3.1 Vertical and Horizontal Pointers Within BYGONE. 289
3.3.4.1 The Environment of BYGONE 290

3.3.5.1 The Externally-Apparent Structure of Text in BYGONE 290

3.4.3.4.4.1 The Precision of the Fixed-Point/Results of Arithmetic Operations. 444

3.4.3.4.5.1 Type Dominancy Within Generic Arithmetic Operations. 445

3.4.3.4.7.1 The Operator Precedence. 449

3.4.4.1.1 The General Form of Statement. 452

3.4.4.2.1.1 The General Form of Statement. 463

3.4.4.2.2.1 The General Form of Statement. 464

3.4.4.3.1 The General Form of Statement. 466

3.4.4.3.2 The Carriage Controls. 467

3.4.5.1.1 The Relational Operators. 470

3.4.5.1.2 The Logical Operators and Their Precedence. 470

3.4.5.1.3 The General Form of Conditional Statement. 471

3.4.5.2.1.1 The General Form of Unconditional GOTO Statement. 476

3.4.5.2.2.1.1 The General Form of Indexed GOTO Statement. 478

3.4.5.2.3.1 The General Form of Multiple-Route GOTO Statement. 479

3.4.5.5.1.1 The General Form of Test-at-Beginning Iteration Statement. 485

3.4.5.5.1.2 The General Form of Test-at-End Iteration Statement. 486

3.4.5.5.1.3 The General Form of Test-in-the-Middle Iteration Statement. 487
3.3.6.1.2.1.1 The Modes of Operations Within

3.3.6.2.1.1 The General Form of Command.

3.3.6.2.1.2 An Example of Entering Text from the Terminal.

3.3.6.2.2.1 The General Form of Command.

3.3.6.2.3.1 The General Form of Command.

3.3.6.2.4.1 The General Form of Command.

3.3.6.2.5.1 The General Form of Command.

3.3.6.2.6.1 The General Form of Command.

3.3.6.2.7.1 The General Form of Command.

3.3.6.3.1.1.1 Externally-Apparent Structure of Sequential File.

3.3.6.3.1.1.2 Internal Structure of Sequential Files.

3.3.6.3.1.2.1 Externally Apparent Structure of a Random Access File.

3.3.6.3.1.2.2 Internal Organization of a Random Access File.

3.3.6.3.1.2.3 Contents of the File Before the Modification.

3.3.6.3.1.2.4 Contents of the File After the Modification.
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.6.3.2.1</td>
<td>The General Form of Command.</td>
<td>322</td>
</tr>
<tr>
<td>3.3.6.3.3.1</td>
<td>The General Form of Command.</td>
<td>323</td>
</tr>
<tr>
<td>3.3.6.3.4.1</td>
<td>The General Form of Command.</td>
<td>324</td>
</tr>
<tr>
<td>3.3.6.3.5.1</td>
<td>The General Form of Command.</td>
<td>325</td>
</tr>
<tr>
<td>3.3.6.3.6.1</td>
<td>The General Form of Command.</td>
<td>326</td>
</tr>
<tr>
<td>3.3.6.3.7.1</td>
<td>The General Form of Command.</td>
<td>327</td>
</tr>
<tr>
<td>3.3.6.3.8.1</td>
<td>The General Form of Command.</td>
<td>328</td>
</tr>
<tr>
<td>3.3.6.3.9.1</td>
<td>The General Form of Command.</td>
<td>329</td>
</tr>
<tr>
<td>3.3.6.3.10.1</td>
<td>The General Form of Command.</td>
<td>330</td>
</tr>
<tr>
<td>3.3.6.3.11.1</td>
<td>The General Form of Command.</td>
<td>331</td>
</tr>
<tr>
<td>3.3.6.3.12.1</td>
<td>The General Form of Command.</td>
<td>332</td>
</tr>
<tr>
<td>3.3.6.3.13.1</td>
<td>The General Form of Command.</td>
<td>333</td>
</tr>
<tr>
<td>3.3.6.3.14.1</td>
<td>The General Form of Command.</td>
<td>334</td>
</tr>
<tr>
<td>3.3.6.3.15.1</td>
<td>The General Form of Command.</td>
<td>335</td>
</tr>
<tr>
<td>3.3.6.3.16.1</td>
<td>The General Form of Command.</td>
<td>336</td>
</tr>
<tr>
<td>3.3.6.4.1.1</td>
<td>The General Form of Command.</td>
<td>337</td>
</tr>
<tr>
<td>3.3.6.4.2.1</td>
<td>The General Form of Command.</td>
<td>339</td>
</tr>
<tr>
<td>3.3.6.4.3.1</td>
<td>The General Form of Command.</td>
<td>341</td>
</tr>
<tr>
<td>3.3.6.4.4.1</td>
<td>The General Form of Command.</td>
<td>342</td>
</tr>
<tr>
<td>3.3.6.4.5.1</td>
<td>The General Form of Command.</td>
<td>343</td>
</tr>
</tbody>
</table>
3.3.6.4.6.1 The General Form of Command.  
3.3.6.4.7.1 The General Form of Command.  
3.3.6.4.8.1 The General Form of Command.  
3.3.6.4.9.1 The General Form of Command.  
3.3.6.4.10.1 The General Form of Command.  
3.3.7.1.1 The Dynamic Structure of the Text Bank.  
3.3.7.1.2 A Block of Text.  
3.3.7.1.3 The Organization and Components of the Text Bank.  
3.3.7.1.4 The Dynamic Allocation of Storage.  
3.3.8.1 The Command Processor.  
3.4.1.1 Data Representation Within DARRIE.  
3.4.1.5.1 The Definition of Identifiers Within DARRIE.  
3.4.1.5.13.1 The General Form of the Declaration Statement for Simple Variables.  
3.4.1.5.13.2 The Declaration Tree of Simple Variables.  
3.4.1.5.13.3 The Definition of Simple Variable Declaration Statement.  
3.4.1.5.14.1 The General Form of a Simple Variable Descriptor.  
3.4.1.5.15.1 Some of the Major Activities of Accessing Algorithms for Simple Variables.
3.4.1.5.16.1 The General Form and the Definition of the Initialization Statement for Simple Variables. 379

3.4.1.7.1 The Definition of Arrays Within $\text{DARRIE}$. 380

3.4.1.7.1.1 The General Form of the Declaration Statement for Homogeneous Arrays. 382

3.4.1.7.1.2 The Declaration Tree for Homogeneous Arrays. 383

3.4.1.7.1.3 The Definition of the Declaration Statement for Homogeneous Arrays. 384

3.4.1.7.2.1 The Descriptor of a Homogeneous n-Dimensional Array with Fixed-Size Length. 385

3.4.1.7.2.2 Fixed-Size Length and Variable-Size Length Arrays Storage Layout. 387

3.4.1.7.4.1 The General Form of Homogeneous Array Initialization Statement. 391

3.4.1.8.1 An Example of a Structure. 393

3.4.1.8.2 The Tree Representation of the Structure in Figure 3.4.1.7.2.2. 394

3.4.1.8.3 An Array of Structures. 394

3.4.1.8.1.1 The General Form of a Heterogeneous Array ( ) Declaration. 397

3.4.1.8.1.2 The Storage Representation of a Structure Whose Elementary Data Items Are Fixed in Size. 398

3.4.1.8.2.1 The General Form of the Descriptors for Heterogeneous n-Dimensional Arrays. 399

3.4.1.8.2.2 The Complete Set of Descriptors for the Example in Figure 3.4.1.8.1. 400
3.4.1.8.2.3 The Modified Version of the Structure in Figure 3.4.1.8.1. 402
3.4.1.8.2.4 The Complete Set of Descriptors for the Structure in Figure 3.4.1.8.2.3. 403
3.4.1.8.4.1 The General Form of the Initialization of Structures. 406
3.4.1.8.4.2 The General Form of the Initialization of Arrays of Structures. 407
3.4.2.1 The Program Structure. 409
3.4.2.2.1 A Program Which Requires a Unique Set of Statements at Several Points. 411
3.4.2.3.1 The Methods by Which a Set of Instructions Can be Inserted at Different Points Within a Program. 412
3.4.2.3.2 One Method of Having Access to a Unique Set of Statements at Several Points in a Program - Method 1. 413
3.4.2.4.1.1 An Example of Programs with Their External Procedures. 417
3.4.2.4.1.4.1 The Procedure Definition. 424
3.4.3.1.1 The Repertoire of Arithmetic Operations. 425
3.4.3.3.1.1 The Definition of INTEGER/Constants. 429
3.4.3.3.2.2.1 The Definition of REAL/Constants. 434
3.4.3.4.1 The General Form and Definition of Arithmetic Expressions. 441
## LIST OF EXHIBITS

<table>
<thead>
<tr>
<th>Exhibit</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3.2.1.1</td>
<td>The Turing Machine which Converts the Stroke Representation of an Integer &amp; Number into its Binary Representation.</td>
<td>55 &amp; 58</td>
</tr>
<tr>
<td>1.3.2.1.2</td>
<td>The Turing Machine which Generates the Error Check Placements for a Hamming Error Check Encoder.</td>
<td>60</td>
</tr>
<tr>
<td>1.3.2.4.1</td>
<td>The Normal Algorithm which Converts a String of Characters (composed of a's and b's) into its reverse order.</td>
<td>67</td>
</tr>
<tr>
<td>1.3.2.4.2</td>
<td>The Normal Algorithm which Converts a String of Characters (composed of any English Letters, A-Z) into its reverse order.</td>
<td>68</td>
</tr>
<tr>
<td>3.2.4.2.2</td>
<td>Character Generator Program Writer/the Source Program.</td>
<td>180</td>
</tr>
<tr>
<td>3.2.4.2.3</td>
<td>Decomposition Expressions/the Input to CGPW.</td>
<td>182</td>
</tr>
<tr>
<td>3.2.4.2.4</td>
<td>Software Simulated Character Generators/the Output of CGPW.</td>
<td>185</td>
</tr>
<tr>
<td>3.2.4.3.1</td>
<td>Input/Output Driver/the Source Program.</td>
<td>264</td>
</tr>
<tr>
<td>3.2.4.3.2</td>
<td>Samples of the Output of IODRV</td>
<td>275</td>
</tr>
</tbody>
</table>
PREFACE

Toward the end of my senior year at Oregon State University, I began a research project the end result of which would have been the creation of an automated English-to-Farsi translator.

Early in my graduate years, Dr. Goheen (my major advisor) and I reached the conclusion that it was not feasible to pursue further the project design and implementation, and considering the vastness of our ignorance of the nature of human communication, impossible at the time. We realized that the need for a tremendous amount of memory and computer time was a limiting factor (if not a blocking one) to our progress. With the recent development in computer hardware which has resulted in cheaper, more reliable and compact circuitries, larger memories, and faster input/output devices, our ultimate goal: the design and implementation of English-Farsi-English automatic translator seems more attractive than before.

Later in my graduate years, I decided to create a subset of what we had in mind:

THE INTERACTIVE COMPUTER PROGRAMMING SYSTEM-IN FARSI.
PEESHTAZ

In this thesis I give an illustration of the computing environment and discuss the elements directly associated with my creation.
In Chapter One, I describe the operations, properties, computing limitations and procedures to create some of the models by which algorithms can be executed. In Chapter Two, I discuss the evolutionary development of languages, the means of expressing algorithms so that machines can execute them. In Chapter Three, I present the design specification of a new interactive computer programming system which uses the alphabet and grammar of Farsi and possesses features not found in existent and heavily used computer systems. This system, once implemented, will enable Iranian manufacturing, scientific, engineering, governmental, business and cultural organizations, and even high school and grammar school students, to solve their problems by preparing computer programs and communicating with the computer using a subset of their own language. Of course, the translated versions of the systems features can be adopted by other non-Farsi computer systems.

The references made throughout the text belong to four categories:

(1) references made to the footnotes, such as in the seventh line on page 1,

(2) references made to the bibliography materials which follow the referencing section, such as in the ninth line on page 1,
(3) references made to the sections of the thesis, such as in the 21st and 22nd lines on page 2, and
(4) references made to the figures within the thesis, such as in the last line on page 9.
1. ALGORITHMS AND THEIR EXECUTING MODELS

1.1 Algorithms

1.1.1 The Origin of Algorithm

It is found that the true origin of the word "algorithm" arose from the name of the Great Iranian (Persian) mathematician/astronomer: Abu-Jafar Mohammad ibn Musa Al-Khowrazmi. Al-Khowrazmi wrote the cele-

*Al-Khowrazmi (813?-835/845?), sometimes called the "Father of Algebra" was born in Khowram which was "the region corresponding roughly with Chorasmia, a north province of ancient Persia; covered valley of lower Oxus and extended across steppes west to Caspian Sea and east to Bukhara...First Russian contact in 17th Century under U.S.S.R. approximately same region known as Khorezmoblost," as stated in Webster's New Geographical Dictionary, Massachusetts: G. & C. Merrian Company, p. 6060607. Al-Khowrazmi wrote his books in Arabic language which at that time was a common language in which intellectual articles were expressed because of the extension of the rule and influence of the Moslem star and crescent over a territory reaching from India through Persia, Mesopotamia, and northern Africa, out into Spain. The language that he used is perhaps the reason for the false assumption that he was an Arabic mathematician rather than Iranian as erroneously stated in the following publications:

brated book *Ketab' al-jabr v'al-mugrabalah*\(^4\) or Hisab' *al-gabr v'al-mugrabalah* (rules of restoration and reduction,\(^4\) science of the reunion and the opposition, or science of reduction and cancellation)\(^1\) from which the word "algebra" was derived. The word "algorism" stemmed from the Latin translation of Abu-Jafar's last name, Al-Khowrazmi. The word "algorism" means the art of calculating in any particular way (using the Hindu/Persian numerals: , 1, 2, 3, 5, 6, 7, 8, 9).

1.1.2 The Meaning of Algorithm

Throughout this thesis the word algorithm means: a finite set of primitive instructions (in which no creativity: possession of intelligence or ability to innovate is conceived) to transform an input into an output. The algorithm is said to be an effective procedure if the process of transformation (of an input into an output) can be accomplished in a finite number of steps.

The lack of intelligence or ability to innovate among the set of instructions of which an algorithm is composed, assures the possibility of creation of basic machines for automatic means\(^1.3\) of processing all algorithms. However, not always can a problem be solved by an algorithm.\(^1.3.2.7\).

If it can be proved that a problem cannot be solved by an algorithm, it is said to be unsolvable.
1.2 The Elements of Computing

Figure 1.2.1 illustrates the computing environment and its major elements:

1. problems (solvable and/or unsolvable),
2. human—or something which possesses intelligence and ability to innovate; the latter not yet identified,
3. algorithms (for solvable problems),
4. languages (means of expressing algorithms, uniquely, in ways comprehensible to executing models), and
5. executing models (means of automation).

1.3 The Executing Models

The executing models provide means of automatic processing of algorithms by accepting inputs and algorithms expressed in their own languages and generating the outputs.

The executing models can be organized in two major categories:

- finite state machines
- infinite machines
Figure 1.2.1 The Computing Environment.
1.3.1 Finite State Machines (Finite Automata)

Finite state machines are of special importance because of their finite nature which assures the possibility of their physical realization, and their completely describable behavior in discrete sets without any ambiguity or approximation.

They are composed of a finite collection of digital elements (the things capable of exhibiting two-phase outputs, 0 and 1, strictly as functions of their two-value inputs, 0 and 1, at synchronous moments) interconnected with a finite number of input (independent) variables and a finite number of output (dependent) variables (produced by the machine as functions of input variables).

\[
\begin{align*}
    x_1 & \quad --- y_1 = y_1(x_1, \ldots x_n) \\
    x_2 & \quad --- y_2 = y_2(x_1, \ldots x_n) \\
    x_n & \quad --- y_m = y_m(x_1, \ldots x_n)
\end{align*}
\]

Figure 1.3.1.1 A General Picture of a Finite State Machine.

The behavior of a finite state machine is composed of a set of simple, linear events in time.* The events occur

*Combinational networks are strictly functions of their inputs and time does not play any significant role in their behavior.
at discrete moments, between which no activity that would disturb the internal behavior of the machine occurs. With the passage of time the machine perambulates through a set of "conditions" or "states" corresponding to the particular values assumed by each of the elements of which they are composed and it responds to its input changes, as well as to autonomous changes in its own variables.

If at time \( t \) input \( x_j \) is received (from the environment), an output \( o_k \) is produced which depends both on \( x_j \) and on the state in which the machine was at time \( t \). If the state in which the machine was (at time \( t \)) is determined by the history \( H(t) \) (a complete record of machine activities up to the present time, \( t \)), assuming the machine is deterministic, the following relationship exists:

\[
Y(t+1) = P(H(t),x(t))
\]

The capability of a finite state machine in storing past events must be limited (because of its finite nature). Therefore, it cannot store a complete record of the history of the machine and as a result it cannot distinguish between all possible histories. In other words, finite state machines can distinguish by their present and future behavior only among some finite number of classes of histories (internal states) of the machine.

*The machine which has one and only one starting state. If a machine has more than one starting state it is called undeterministic (Figure 1.3.1.4).
Since the machine's response $Y(t+1)$ to $x(t)$ depends only on $Q(t)$ (the internal state of the machine at time $t$), relationship (1) can be presented as follows:

\[(2) \quad Y(t+1) = F(Q(t), x(t))\]

The internal state at time $t$, $Q(t)$, depends on the history of the machine at time $t$. The history at time $t+1$ differs from the history at time $t$ only in having an extra term, $x(t)$, and nothing else; thus the internal state at time $t+1$ depends on the input at the previous time, $x(t)$, and state in which the machine was at the previous time, $Q(t)$, so the following relationship holds:

\[(3) \quad Q(t+1) = G(Q(t), x(t))\]

The two functions (2) and (3) give the complete description of the behavior of the machine from which a step-by-step procedure can be followed to determine the output and state of the machine at any future time, given the input and the initial state.
1.3.1.1 Finite State Machines Classification

Finite state machines can be categorized depending upon their structure, as presented below:

![Finite State Machines Classification Diagram]

*Linear Cascade, Cellular arrays or networks

Figure 1.3.1.1.1 Finite State Machines Classification.

1.3.1.2 Combinational Networks

Combinational networks are composed of those finite state machines whose outputs are strictly functions of their inputs and time is not normally considered to be of significance in their behaviors. The machine is specified by determining which combination of inputs produces a particular output. Thus, the dependence on time need not be explicitly indicated, since the values of all the variables are those at some single time. If the present value of the output is dependent not only on the present value of the inputs but also on the past history of the events within the machine, it is a sequential system: those
machines which introduce the full range of complexity to describe and characterize everything of which digital machines are capable. The combinational networks may be considered as a subset of sequential machines.

1.3.1.2.1 Machine Realization of Combinational Networks

There are \(2^n\) different combinational machines with \(n\) input variables and single outputs, since there are \(2^n\) different mappings from the domain of \(n\)-tuples \((2^n)\) to the range of the output function \((0,1)^2\) as illustrated in Figure 1.3.1.2.1.1.

It would be hopelessly costly to create \(2^n\) different circuitries to realize all of the combinational networks of \(n\) variables with single outputs, as \(2^n\) grows alarmingly fast for even small values of \(n\). Nevertheless, a finite state machine can be decomposed into a set of primitive components; as Turing thought algorithms could be decomposed into a set of primitive operations and so was able to create his basic machine to execute all the computable functions.
### Figure 1.3.1.2.1.1 Different Combinational Machines With \( n \) Input Variables and Single Outputs.

<table>
<thead>
<tr>
<th>( X_n \ldots X_2 X_1 )</th>
<th>( f_0(X_1, X_2, \ldots, X_n) )</th>
<th>( f_1 f_2 \ldots f_{2^n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 \ldots 0 0</td>
<td>( y_0 )</td>
<td></td>
</tr>
<tr>
<td>0 \ldots 0 1</td>
<td>( y_1 )</td>
<td></td>
</tr>
<tr>
<td>0 \ldots 0 1</td>
<td>( y_2 )</td>
<td></td>
</tr>
<tr>
<td>\ddots</td>
<td>\ddots</td>
<td>\ddots</td>
</tr>
<tr>
<td>1 \ldots 1 1</td>
<td>( y_2^{n-1} )</td>
<td></td>
</tr>
</tbody>
</table>

\( n \) variables \( 2^{2^n} \) functions

### Figure 1.3.1.2.1.2 The Growth of Function \( 2^{2^n} \).

<table>
<thead>
<tr>
<th>( n )</th>
<th>( 2^{2^n} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>256</td>
</tr>
<tr>
<td>4</td>
<td>65536</td>
</tr>
<tr>
<td>5</td>
<td>4294967296</td>
</tr>
<tr>
<td>6</td>
<td>( 2^{64} )</td>
</tr>
<tr>
<td>7</td>
<td>2^{128}</td>
</tr>
<tr>
<td>8</td>
<td>2^{256}</td>
</tr>
<tr>
<td>\ddots</td>
<td>\ddots</td>
</tr>
</tbody>
</table>

\( 2^{2^n} \)
The decomposition of the machines into a small set of primitive elements, each of which is capable of performing only a basic operation without possessing any ability to innovate or use intelligence, not only reduces the cost of realization, but also allows the execution of effective algorithms.

In seeking for the small set of primitive elements with which all of the finite state machines can be realized it is noticed that for the single variable machine the following possibilities exist:

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>$f_0$</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1.3.1.2.1.3 All the Possible Functions of a Single Variable.

The set $0, 1, \overline{\text{NOT}}$ is sufficient to express the above possibilities. For the two input variable machines, sixteen different realizations exist.

<table>
<thead>
<tr>
<th>input</th>
<th>output</th>
<th>$f_0$</th>
<th>$f_1$</th>
<th>$f_2$</th>
<th>$f_3$</th>
<th>$f_4$</th>
<th>$f_5$</th>
<th>$f_6$</th>
<th>$f_7$</th>
<th>$f_8$</th>
<th>$f_9$</th>
<th>$f_{10}$</th>
<th>$f_{11}$</th>
<th>$f_{12}$</th>
<th>$f_{13}$</th>
<th>$f_{14}$</th>
<th>$f_{15}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1</td>
<td>0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 1</td>
<td>0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 0</td>
<td>0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 1</td>
<td>0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.3.1.2.1.4 All the Functions of Two Variables.
The functions \( f_0 \) and \( f_{15} \) are the constants (0,1) which were included in the set of operations capable of representing all the single variable machines. Figure 1.3.1.2.1.3

The other functions could all be realized by introducing the additional conjunction and disjunction gates as illustrated in Figures 1.3.1.2.1.5 and 1.3.1.2.1.6.

![Conjunction (AND) Gate.](image)

![Disjunction (inclusive-OR) Gate.](image)

The set \( \{0,1,\text{NOT, AND, OR}\} \) is sufficient to realize all of the functions of two input variables (Figure 1.3.1.2.1.7) using the perfect induction method.

The AND and OR operations may be extended to have \( n \) input variables instead of only two.

\[
\text{AND} (P_1, \ldots, P_n) = 1 \text{ if every } P \text{ is true}
\]

\[
\text{OR} (P_1, \ldots, P_n) = 1 \text{ if one or more } P's \text{ are true}
\]

With the above extension and the confirmation of the associativity of the operations AND and OR below:

\[
(p_1 \text{ AND } p_2) \text{ AND } p_3 = p_1 \text{ AND } (p_2 \text{ AND } p_3) = p_1 \text{ AND } p_2 \text{ AND } p_3
\]

\[
(p_1 \text{ OR } p_2) \text{ OR } p_3 = p_1 \text{ OR } (p_2 \text{ OR } p_3) = p_1 \text{ OR } p_2 \text{ OR } p_3
\]

and the consideration that any \( n \)-variable machine can be
Figure 1.3.1.2.1.7 Realization of All the Two Input Variable Machines.
expressed in terms of its fundamental products (minterms), the set $0, 1, \text{NOT, AND, OR}$ becomes sufficient to express any finite state machine of one or more input variables.

Since $(0) \text{ AND } (p_1, p_2, \ldots, p_n) = 0$ and $(1) \text{ OR } (p_1, p_2, \ldots, p_n) = 1$, the set of primitive operations capable of representing finite state machines reduces to \text{NOT, AND, OR}. Also \text{NOT} and \text{AND} can do all that can be done by \text{OR} (as was found by De Morgan):

$p \text{ OR } q = \overline{p} \text{ AND } \overline{q} \text{ or } \overline{p} \text{ AND } q = \overline{p} \text{ OR } q$

Hence, $\{\text{NOT, AND}\}$ is a minimally complete set (the set in which the elimination of one or more elements would lead to an incomplete set to realize all of the finite state machines) containing all that is needed to realize any finite state machine.

<table>
<thead>
<tr>
<th>$p_1$, $p_2$, $\ldots$, $p_{n-2}$, $p_{n-1}$, $p_n$</th>
<th>$f(p_1, p_2, \ldots, p_n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 $\ldots$ 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 $\ldots$ 0 0 1</td>
<td>1</td>
</tr>
<tr>
<td>0 0 $\ldots$ 0 1 0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 $\ldots$ 0 1 1</td>
<td>0</td>
</tr>
<tr>
<td>0 0 $\ldots$ 1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>0 0 $\ldots$ 1 0 1</td>
<td>0</td>
</tr>
<tr>
<td>0 0 $\ldots$ 1 1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 $\ldots$ 1 1 1</td>
<td>1</td>
</tr>
</tbody>
</table>

$f(p_1, p_2, \ldots, p_n) = \overline{p_1}p_2 \ldots \overline{p_{n-1}}p_{n-1} \overline{p_1}p_2 \ldots p_{n-1} \overline{p_n} + \overline{p_1}p_2 \ldots$

$p_{n-2}p_{n-1}p_n + \ldots + p_1p_2 \ldots p_{n-2}p_{n-1}p_n$

Figure 1.3.1.2.1.8 Truth Table and the Fundamental Products (minterms) of an n-Input Variable Machine.
The set could be reduced further, yet remain sufficient to represent any machine, when the Sheffer stroke (NAND) operator is considered:

\[
\begin{array}{c|c|c}
 p & q & p \text{NAND} q \\
 \\
 0 & 0 & 1 \\
 0 & 1 & 1 \\
 1 & 0 & 1 \\
 1 & 1 & 0 \\
\end{array}
\]

Figure 1.3.1.2.1.9 NAND Operation.

Since NOT and AND are logically complete and they can be represented by the NAND operator alone, as follows:

\[
p \text{NAND} p = \overline{p \text{AND} p} = \overline{p}
\]
\[
p \text{AND} q = \overline{p \text{NAND} q}
\]

Thus all of the finite state machines, regardless of their complexity, can be realized by a single primitive operator, NAND.
The same holds for the operation NOR defined as follows:

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
<th>pNORq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\{\text{AND, NOT}\} \\
\{\text{OR, NOT}\} \\
\{\text{NAND or NOR}\}
\]

Figure 1.3.1.2.1.10 The Minimally Complete Set of Primitive Operations.
1.3.1.2.1.1 Gate Realization

A gate network is made up of interconnected nodes (digital elements) wherein each node monitors several inputs and produces an output (representing the truth value of the particular combination of values appearing on the inputs).

For a function realizable by a combinational network, there always exists a systematic procedure for its gate realization, as shown in Figure 1.3.1.2.1.1.1.

As an example the function \( f(w,x,y,z) \), which is presented in decimal sum of product notation \( ^2,11 \) as \( \Sigma(0,4,5,7,8,9,13,15) \) is realized.

The following truth table (one method by which a machine can be represented) shows the machine outputs (0 or 1) for all the combinations of the input variables \( 2^4 \) with combinations (rows) arranged in formal binary ordering. It is derived from the decimal sum of product notation \( \Sigma(0,4,5,7,8,9,13,15) \) where the decimal numbers refer to those rows for which the machine outputs are 1's.
Figure 1.3.1.2.1.1.1 A Systematic Procedure for Realization of a Switching Function.
Step 1: Truth Table

<table>
<thead>
<tr>
<th>wxyz</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1</td>
</tr>
<tr>
<td>0001</td>
<td>0</td>
</tr>
<tr>
<td>0010</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>1</td>
</tr>
<tr>
<td>0101</td>
<td>1</td>
</tr>
<tr>
<td>0110</td>
<td>0</td>
</tr>
<tr>
<td>0111</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>1</td>
</tr>
<tr>
<td>1001</td>
<td>1</td>
</tr>
<tr>
<td>1010</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>0</td>
</tr>
<tr>
<td>1100</td>
<td>0</td>
</tr>
<tr>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>0</td>
</tr>
<tr>
<td>1111</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 2: Karnaugh Map

<table>
<thead>
<tr>
<th>yz</th>
<th>wx 00 01 11 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>01</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>11</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>

Step 3: Minimized Sum of Product

\[ f(w,x,y,z) = xz + wx'y' + w'y'z' \]

The minimized sum of product form is derived from the above Karnaugh map which shows where the adjacencies (the possibilities of combining two or more, \( T=AX+AX' = A \), minterms) exist.
Step 4: Multi-level AND/OR Realization

Using OR/AND network the same function can be realized as follows:
Step 2: Karnaugh Map

\[
\begin{array}{cccc}
00 & 01 & 11 & 10 \\
00 & 0 & 0 & 0 \\
01 & 0 & 0 & 0 \\
11 & 0 & 0 & 0 \\
10 & 0 & 0 & 0 \\
\end{array}
\]

Step 3: Minimized Product of Sum

\[f(w,x,y,z) = (x+y')(y'+z)(w'+x'+z)(w+x+z')\]

Step 4: Multi-level OR/AND
Two Level OR/AND

\[
\begin{align*}
x & \quad \text{OR} \\
y & \quad \text{OR} \\
y' & \quad \text{OR} \\
Z & \quad \text{AND} \\
w' & \quad \text{OR} \\
x' & \quad \text{OR} \\
w & \quad \text{OR} \\
Z & \\
\end{align*}
\]

The sum of product representation can also be realized by NAND or NOR networks quite easily. From the fundamental theorem of \( \bar{x} = x \), the following structural interpretation can be observed:

\[
\begin{align*}
\bar{x} & \equiv \overline{\overline{x}} \\
\end{align*}
\]

which means that into any lead, two NOT gates can be inserted without changing the terminal behavior. Also, from the definition of NAND and DeMorgan's Theorem the following can be observed: \(^{5,12}\)

\[
A \text{ NAND } B = \overline{AB} = \overline{A} + \overline{B}
\]

Given the AND/OR realization in Figure 1.3.1.2.1.1.2(a) its equivalent NAND realization can be immediately obtained as in Figure 1.3.1.2.1.1.2(b) and (c).
Figure 1.3.1.2.1.1.2 From AND/OR Realization to its Equivalent NAND Realization.

Considering the $\bar{A} = A \text{ NAND } A$ because of the following truth table:

<table>
<thead>
<tr>
<th>A</th>
<th>$\bar{A}$</th>
<th>A NAND A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

the AND/OR realization of the running example can be converted into its NAND realization as in Figure 1.3.1.2.1.1.3 and similarly converted into NOR realization as in Figure 1.3.1.2.1.1.4.
Figure 1.3.1.2.1.1.3 The NAND Realization of the Example in 1.3.1.2.1.1.

Figure 1.3.1.2.1.1.4 The NOR Realization of the Example in 1.3.1.2.1.1.
1.3.1.2.1.2 Branch Realization

The same example can be realized by serial-parallel* branch contacts as shown in Figure 1.3.1.2.1.2.1.

The transmission in branch networks is accomplished by a set of appropriately connected relays. A relay consists of an electromagnet and a set of switch contacts. When the relay coil is energized, the switch operates. There are three basic types of relays\(^8\), shown in Figure 1.3.1.2.1.2.2.

The NO relay is normally open and the contacts close when the coil is energized. The NC relay is normally closed and the contacts open when the coil is energized. The TRANSFER relay is a combination of NO and NC contacts, sharing a common movable switch.

*An example of non-serial-parallel network is the even parity function (f=E(0,3,5,6,9,10,12,15)) on four variable).

---

The following is an example of a bridge network which realizes the function f = \((w+y)(w+v+z)(x+v+y)(x+z)\).\(^5\)
Figure 1.3.1.2.1.2.1 The Serial-parallel Branch Contacts of the Example in 1.3.1.2.1.1.

Figure 1.3.1.2.1.2.2 Basic Relay Types.2, p. 581

Figure 1.3.1.2.1.2.2 shows the relay circuits which realize AND, OR, and NOT functions (the elements from which all switching functions can be constructed).
"The branch network has unquestionably seen a long decline in interest, simply because most networks nowadays are of the other variety. Probably, however, with respect to cleverness and intuition, and sheer perseverance, the branch network has provided the most challenge to clever network theorists and thinkers. Very sophisticated analysts, from graph theory specialists to matrix theory sophisticates have dabbled with the branch networks, with very few general results regarding minimality resulting from all these efforts."
1.3.1.2.2 Spatial Iterative Networks

Spatial iterative networks represent the class of combinational networks in which an enforced degree of commonality among the variables exists.

The signal generated to the left of cell $i$, $y_i$, contains complete information about the first $i-1$ variables so that cell $i$ can examine variable $x_i$ and generate a sufficient set of similar signals to pass along to cell $i+1$.

Figure 1.3.1.2.2.2 illustrates the process of realization of a spatial iterative network generating three different outputs on an arbitrary number of variables. The outputs correspond to the number of true variables in modulo 3.
\begin{tabular}{|c|c|c|}
\hline
$y_1$ & $y_2$ & \text{state} \\
\hline
0 & 0 & 0 \mod 3 \\
0 & 1 & 1 \mod 3 \\
1 & 0 & 2 \mod 3 \\
1 & 1 & \text{don't care*} \\
\hline
\end{tabular}

\begin{tabular}{|c|c|c|c|c|}
\hline
$x_i$ & $y_1$ & $y_2$ & $T_1$ & $T_2$ \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & * & * \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 0 & 0 \\
1 & 1 & 1 & * & * \\
\hline
\end{tabular}

$Y_1 = x_i y_2 + \overline{x}_i y_1$

$Y_2 = x_i \overline{y}_1 \overline{y}_2 + \overline{x}_i y_2$

Figure 1.3.1.2.2.2 An Example of Iterative Combinational Network.
In some cases certain of the input combinations are not considered of any significance — either because the external environment is such that they never happen, or because if they do happen their outputs do not influence the overall behavior of the system. Whenever the output does not have any effect it can be assigned either a 0 or a 1 and thus is a don't care. Don't cares provide a certain extra degree of freedom that the designer can use in simplifying the networks.
1.3.1.3 Sequential Machines

Instead of spatial distribution of machine components (as in spatial iterative combinational networks) one cell capable of generating the appropriate output at successive time instances and properly delaying the outputs to coincide with the next input can be used. Figure 1.3.1.3.1 illustrates the general model for sequential machines consisting of separate combinational logic and memory.

Where \( x_1, \ldots, x_n \) are the external inputs (inputs).

\( y_1, \ldots, y_p \) are the internal inputs (states).

\( Y_1, \ldots, Y_p \) are the next internal inputs (next states), and

\( Z_1, \ldots, Z_m \) are the external outputs (outputs).

![General Sequential Machine](image)

**Figure 1.3.1.3.1 General Sequential Machine.**

The memory stores (delays the passage of) information about the past events. It receives the next internal states \( (Y_1, Y_2, \ldots, Y_p) \) and provides the internal inputs
(y₁, y₂, ..., yᵢ). The memory may be time dependent as in clock mode and pulse mode circuits or time independent as in level mode sequential circuits.

The combinational logic section receives the external inputs (x₁, x₂, ..., xₙ) and the internal inputs (y₁, y₂, ..., yᵢ) and generates the external outputs (Z₁, Z₂, ..., Zₘ) and the set of next internal states (Y₁, Y₂, ..., Yᵢ). Its behavior is not time dependent and can be illustrated by the following equations:

\[ Zᵢ = fᵢ(x₁, x₂, ..., xₙ, y₁, y₂, ..., yᵢ) (i=1,2,...,m) \]

\[ Yᵢ = Qⱼ(x₁, x₂, ..., xₙ, y₁, y₂, ..., yᵢ) (j=1,2,...,p) \]

There are \( 2^{(m+p)} \) sequential machines. The count reduces to the count for single output combinational networks when \( p=0 \) and \( m=1 \).

The parity machine designed earlier can now be constructed in terms of the new model, using delay memory represented by \( \mathfrak{Q} \), as in Figure 1.3.1.3.2.
In Figure 1.3.1.3.4 the systematic procedure for the realization of sequential machines and delays as the memory elements are used to realize a sequential machine which outputs a 1 whenever it detects three successive 1's on its inputs.

* $\oplus$ is the exclusive-or operation defined as:

<table>
<thead>
<tr>
<th>$x$</th>
<th>$y$</th>
<th>$x \oplus y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Step 1...</td>
<td>The state graph is designed</td>
<td></td>
</tr>
<tr>
<td>Step 2...</td>
<td>The state assignment table is designed.</td>
<td></td>
</tr>
<tr>
<td>Step 3...</td>
<td>The flowtable is produced</td>
<td></td>
</tr>
<tr>
<td>Step 4...</td>
<td>The truth table is produced after the choice for the type of memory elements is made.</td>
<td></td>
</tr>
<tr>
<td>Step 5...</td>
<td>The minimization procedure is employed to derive the minimized form of excitation equations and the output function.</td>
<td></td>
</tr>
<tr>
<td>Step 6...</td>
<td>The circuit is designed as a result of step 5.</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1.3.1.3.3 The General Procedure for the Realization of Sequential Machines.
Step 1. State Graph

Step 2. State Assignment Table

<table>
<thead>
<tr>
<th></th>
<th>Y1</th>
<th>Y2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 3. Flow Table

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A/0</td>
</tr>
<tr>
<td>B</td>
<td>A/0</td>
</tr>
<tr>
<td>C</td>
<td>A/0</td>
</tr>
<tr>
<td>D</td>
<td>A/0</td>
</tr>
</tbody>
</table>

Step 4. Truth Table

<table>
<thead>
<tr>
<th>x</th>
<th>Y1</th>
<th>Y2</th>
<th>Present state</th>
<th>next state</th>
<th>Y1</th>
<th>Y2</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>C</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>A</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>B</td>
<td>C</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>C</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>D</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Step 5. Excitation Equations

\[ Y_1 = x'y_1y_2 = xy_1 \]
\[ Y_2 = x'y_1y_2 = xy_1 \]
\[ Z = xy_1 \]

Step 6. AND/OR Realization

Figure 1.3.1.3.4 Realization of a Sequential Machine (which outputs a 1 whenever it sees three successive 1's).
Flip flops (bistable multivibrators) are some of the most commonly used memory elements from which the memory of sequential machines are formed. The machine realized in Figures 1.3.1.3.6 (using J-K flip-flops) and 1.3.1.3.7 (using T flip-flops) recognizes all the strings which end with 11100, 11010, or 11000.

Figure 1.3.1.3.5 T-flip-flops and J-K flip-flops Definition.
1.3.1.3.1 Sequential Machines Classification

![Sequential Machines Diagram]

Clock Mode ---- Pulse Mode ---- Level Mode

1.3.1.3.1.1 Clock Mode Sequential Machines

A clock mode sequential circuit is shown in Figure 1.3.1.3.1.1.1 using $r$ J-K-flip-flops as clocked flip-flops forming the machine memory.²

Figure 1.3.1.3.1.1.1 General Model of Clock Mode Sequential Circuit.
The clock pulses arrive at $t_v \ (v=1,2,3\ldots)$ times changing the state variables by triggering the flip-flops. The values of the J and K variables change at any time in response to changes in the input variables, but the state is changed only at the clock pulse. The values of J and K at time of clock pulse determine the transition state. The excitation and the output variables adjust to the change in state as well as any input changes, reaching new stable levels before the next clock pulse. Of course, the clock pulses which trigger state changes must be of sufficient duration to allow for the system to stabilize after each pulse and before the following pulse.

1.3.1.3.1.2 Pulse Mode Sequential Machines

Pulse mode sequential machines do not use a separate clock line (unlike clock mode sequential machines). Every state change coincides with an input pulse (as in a clock-mode sequential machine) but, pulses which trigger state changes appear on the circuit input lines.²

1.3.1.3.1.3 Level Mode Sequential Machines

Both clock mode and pulse mode sequential machines require pulses to control state transition and thus require timing control. Level mode sequential machines use all inputs as levels: the signals which remain at either value (0,1) for arbitrary periods and change at arbitrary
times. If only one input may change at a time and no changes can occur until the outputs and state transition are both stabilized, then the machine is said to operate in its fundamental mode.

Figure 1.3.1.3.1.3.1 General Model of Level Mode Sequential Machines.

1.3.1.4 Computational Limitation

It turns out that the kinds of problems that cannot be solved by finite state machines are those which violate the basic assumption implicit within their definition (the mapping of a finite set of symbols into another finite set of symbols)—the finiteness assumption.

Indeed the response to an input must ultimately become periodic or otherwise it cannot be recognized by the machine. Because the machine progresses from an initial state, through a succession of states (in accordance with
its behavior) and if the input is longer than the number of states within the machine, the machine must eventually arrive at a state it has previously visited and thus the behavior must follow a periodic pattern.

An arbitrary long input consisting of repeated symbols cannot be recognized by any finite state machine since the output can never become periodic. The set of palindromes (the set of sequences that read the same forwards and backwards) in the alphabet consisting of 0 and 1, is not recognizable by a finite state machine. A well defined set of palindromes, $1^k 0^1 1^k$ for arbitrary values of $k$ and 1 cannot be recognized by a fixed n-state machine because of the memory limitation. Another example where the output does not become periodic within the machine's bound is the multiplication of $2^y$ (represented by a 1 followed by $y$ 0's) by $2^n$ (represented by a 1 followed by $n$ 0's). Because the output (outputing $y+n$ 0's before a 1) does not become periodic within the machine's bound and if it does the machine would output an incorrect number of 0's and/or will not print the 1.

The regular expressions are defined to be the only inputs recognizable by finite state machines. If $A = (a_1, a_2, ..., a_n)$ is defined to be a finite alphabet, then the regular expressions over the alphabet $A$ are defined as:
(1) each member of the alphabet, the null string (\(\lambda\)), or the empty set \(\phi\),

(2) if \(P\) and \(Q\) are regular then so are \(P+Q\), \(PQ\), and \(P^*\) (their union, concatenation, and closure)*

(3) any finite number of combinations of 1 and 2.

Regular expressions can always be expressed in terms of their deterministic transition graph as shown in

*If \(R\) and \(S\) are expressions, their union symbolized by \(R+S\) is the set of all expressions which are members of either \(R\) or \(S\); that is

\[
R+S = \{x | x \in R \text{ or } x \in S\}
\]

The concatenation of \(R\) and \(S\) is the set of all expressions which are members of both \(R\) and \(S\); that is

\[
(R)(S) = \{x | x \in R \text{ and } x \in S\}
\]

The closure of \(R\), symbolized by \(R^*\), means the set of expression obtained by concatenating any number of \(R\)'s together, including none; that is

\[
R^* = +R+RR+RRR+RRRR+\ldots
\]

**Deterministic transition graphs (state diagrams) describe deterministic machines and consist of a vertex for each state and a directed path labeled 0 or 1 emanating from each vertex for every input 0 or 1.
Figure 1.3.1.4.1. An undeterministic graph* can always be transformed into its equivalent deterministic transition graph systematically. Figure 1.3.1.4.2 From a deterministic transition graph the machine to realize it can always be constructed. Figure 1.3.1.4.3 Vice versa, given a machine, different expressions recognized by it can be derived. Figures 1.3.1.4.4 through 1.3.1.4.6

*In undeterministic transition graphs the prime objective is to study and classify sets of sequences and thus some of the restrictions imposed on deterministic transition graphs are removed. Each such graph is composed as a set of vertices and various directed paths connecting them. At least one vertex is assigned to be the starting vertex and at least one vertex is assigned to be the accepting vertex.
Regular Expression  |  Transition Graph
---------------------|---------------------
\((11+0)1*0\)        |\(A\)\(\rightarrow\)\(B\)\(\rightarrow\)\(C\)\(\rightarrow\)\(D\)
\((11)^*(00)^*101\)   |\(A\)\(\rightarrow\)\(B\)\(\rightarrow\)\(C\)\(\rightarrow\)\(E\)\(\rightarrow\)\(F\)
\((1(00)^*1+01*0)^*\) |\(A\)\(\rightarrow\)\(B\)\(\rightarrow\)\(C\)\(\rightarrow\)\(D\)\(\rightarrow\)\(E\)
\(0(11+0(00+1)^*)^*\)|\(A\)\(\rightarrow\)\(B\)\(\rightarrow\)\(C\)\(\rightarrow\)\(F\)
\((0^*01+1)(1+00^*01+11)^*\) |\(A\)\(\rightarrow\)\(B\)\(\rightarrow\)\(C\)

Figure 1.3.1.4.1 Regular Expressions and Their Undeterministic Transition Graphs.
An undeterministic transition graph can always be transformed into its equivalent deterministic transition graph, systematically.

Regular Expression

\[ R = (0*01+1)(1+00*01+11)^* \]

Undeterministic Transition Graph

Transition Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>AB</td>
<td>-</td>
</tr>
<tr>
<td>B</td>
<td>-</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>BC</td>
</tr>
</tbody>
</table>

Successor Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>AB</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>BC</td>
</tr>
<tr>
<td>BC</td>
<td>A</td>
<td>BC</td>
</tr>
<tr>
<td>A</td>
<td>AB</td>
<td>φ</td>
</tr>
<tr>
<td>φ</td>
<td>φ</td>
<td>φ</td>
</tr>
</tbody>
</table>

Deterministic Transition Graph

Figure 1.3.1.4.2 The Derivation of Deterministic Transition Graph from an Underdeterministic Transition Graph.
From a deterministic transition graph the machine to realize it can always be constructed, as illustrated in Figure 1.3.1.4.3.
Vice Versa, given a machine, different expressions recognized by it can be derived as shown in Figures 1.3.1.4.4 to 1.3.1.4.6.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B/0</td>
<td>C/0</td>
</tr>
<tr>
<td>B</td>
<td>B/0</td>
<td>D/0</td>
</tr>
<tr>
<td>C</td>
<td>D/0</td>
<td>A/0</td>
</tr>
<tr>
<td>D</td>
<td>E/1</td>
<td>C/0</td>
</tr>
<tr>
<td>E</td>
<td>D/1</td>
<td>C/1</td>
</tr>
</tbody>
</table>

Synchronizing Sequence: (the shortest path to a singleton in a maximal ambiguity tree)

The Synchronizing Sequence is 0101 and it is indeed unique

Figure 1.3.1.4.4 A Machine and its Synchronizing Sequence.
Homing sequence \( (ABCDE) \) (the sequence which uniquely determines the final state of the machine from the machine's responses to the sequence, regardless of the initial state)

The shortest path is 11011 sequence.

Figure 1.3.1.4.5 A Machine and Its Homing Sequences.
Machine's Flow Table

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A/0</td>
<td>C/0</td>
</tr>
<tr>
<td>B</td>
<td>C/0</td>
<td>D/0</td>
</tr>
<tr>
<td>C</td>
<td>A/0</td>
<td>E/0</td>
</tr>
<tr>
<td>D</td>
<td>D/0</td>
<td>A/0</td>
</tr>
<tr>
<td>E</td>
<td>E/0</td>
<td>C/1</td>
</tr>
</tbody>
</table>

Preset Distinguishing Experiment

(to determine a sequence to which the machine's output is different for each initial state)

The shortest distinguishing experiment is 101011.

Figure 1.3.1.4.6 A Machine and Its Preset Distinguishing Experiment.
1.3.1.5 Conclusion

Finite state machines are of great importance because of their finite nature which assures their physical realizability and their completely describable behavior in discrete steps.

Although there are $2^2^n$ such machines only within the combinational networks of n variables with single output (the smaller class of finite state machines) a small set of elements capable of performing primitive operations is sufficient to realize all of them, regardless of their complexity. Indeed only one type of gate, NAND or NOR, is a sufficient connective by itself to even form different components of the most important class of digital systems: digital computers. Although, most problems are more naturally formulated in terms of AND-OR gates. The sequential machines impose the necessity to store information about previous events which can be maintained by the memory elements which themselves can be realized by the sufficient connectives and thus do not require any elements other than the sufficient connectives mentioned.

The computational limitations of finite state machines which are due to their finiteness are not a barrier in designing general machines executing certain classes of algorithms or designing general purpose machines executing
certain classes of algorithms or designing general purpose machines executing any algorithms (the ones which do not violate the basic assumption implicit in the definition of finite state machines: Finiteness).

Although the states of machines are kept finite in number, a tremendously large number of states may exist within practical machines. For instance, the memory of a medium scale digital computer which may be as large as 64000 words (24 bits within each word) can provide $2^{1536000}$ different states, which would make the finiteness of the machine elements seem to be not much of a practical difficulty.
BIBLIOGRAPHY


12 Short, Robert A. unpublished lecture notes for switching theory and automata courses, Department of Electrical and Computer Engineering, Oregon State University, 1974.

1.3.2 Infinite Machines

Infinite machines are the hypothetical (not in existence within this universe) tools for examining the train of thoughts which cannot develop in the framework of the finite state machines. Even to begin developing a general purpose instruction-obeying machine to realize all effective procedures (by accepting the rules of operations and the input to be processed), such as the "universal machine," capabilities are required beyond those of finite state machines, e.g. multiplying pairs of arbitrary large numbers, detecting syntactical errors in arbitrary expressions and others are essential.

The Turing machine with its finite state computing unit and its finite input is an ideal form of infinite machine to consider. Its memory, the tape where the input, intermediate and final outputs are stored may be thought of as not being infinite, but potentially infinite. The machine may begin with a finite segment of tape and as the ends are encountered, the additional tape is made available. Thus at any point in the computation, the memory is finite, but with the capability of expansion whenever necessary (large enough memory to solve any problem—provided that the problem can be solved).
1.3.2.1 Turing Machine

The Turing machine is a finite state computing machine composed of a tape potentially infinite in both directions and divided into a two-way infinite sequence of cells, where input data, intermediate output formed during a computation and the final output are stored, a read/write head (for inputting/outputting symbols onto/from the tape and moving to an adjacent square), a finite number of distinct internal states (configurations) each of which capable of determining the behavior of the machine in conjunction with the symbol of the tape read/written and an alphabet (a finite set of distinct elements to be input and/or output).

![Figure 1.3.2.1.1 Components of a Turing Machine.](image)

The quintuple list is a finite (nonempty) and unambiguous set (no two quintuples with the same two first symbols) of quintuples of the following forms:

(1-1) \( Q_i S_j Q_l R S_k \)  
(1-2) \( Q_i S_j Q_l L S_k \)
where \( Q_i \)'s represent the internal machine states, \( Q_1 \)'s represent the internal machine states and the HALT condition, and \( S \)'s represent the members to the set of alphabet recognizable by the machine.

Form (1-1) indicates that if the machine is in state \( Q_i \) and symbol \( S_j \) is read, then symbol \( S_j \) must be replaced by symbol \( S_k \) and the head must be moved one cell to the right and the machine must either enter into state \( Q_1 \) or it must halt.

Form (1-2) is different from (1-1) only in the direction in which the head must move: leftward instead of rightward.

A complete configuration is a set of three things:

1. a state of the machine
2. the number of a cell, and
3. a tape.

It is generally indicated by a listing of alphabet letters with the state inserted to the immediate left of the letter at which the machine is operating. A Turing machine operates by mapping a complete configuration onto its successor by using the machine configuration (performing one quintuple at one time).
The following Turing machine converts the stroke representation of an integer number into its binary representation.

Alphabet ::= \{1, 0, /, \equiv\}

Turing Table:

<table>
<thead>
<tr>
<th></th>
<th>/</th>
<th>1</th>
<th>0</th>
<th>\equiv</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2L\equiv</td>
<td></td>
<td>1R\equiv</td>
<td>4L</td>
</tr>
<tr>
<td>2</td>
<td>2L0</td>
<td>3R1</td>
<td>2L\equiv</td>
<td>3R1</td>
</tr>
<tr>
<td>3</td>
<td>3R1</td>
<td>3RO</td>
<td>1R\equiv</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5R1</td>
<td>5RO</td>
<td>4L</td>
<td>4 STOP</td>
</tr>
</tbody>
</table>

Exhibit 1.3.2.1.1 illustrates the quintuple list, the initial configuration and the execution process for converting the input string: ///// into its binary representation: 110.
Exhibit 1.3.2.1.1.

**ALPHA2ET:**

(1'!/F)

**QUINTUPLE LIST:**

(1./,'4L,F)

(1,1,4,1,1)

(11E91,R,F)

(2,1,2,1,0)

(200,31,R,1)

(2,302,1)

(2,7,Z,L,F)

(3,103,P,1)

(3,0,3,8,0)

(3,E,1,R,S)

(4,1,5,P11)

(4,0,4,R,O)

(4,T,4,1.)

(4,94,S.)

**INITIAL CONFIGURATION:**

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**OUTPUT:**

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**EXECUTION PROCESS:**

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**QUINTUPLE LIST:**

```
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**STOP**
The following Turing Machine generates the error check placements for a Hamming error check encoder.

Alphabet ::= \{0,1,*,·,=,↑,%,\} 

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>0</th>
<th>*</th>
<th>·</th>
<th>↑</th>
<th>%</th>
<th>Ë</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1R1</td>
<td>1RO</td>
<td>2R=</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3L*</td>
<td>3L*</td>
<td>2L%</td>
<td>2L*</td>
<td>2L*</td>
<td>3L%</td>
<td>2L%</td>
</tr>
<tr>
<td>3</td>
<td>3L1</td>
<td>3L0</td>
<td>4R</td>
<td></td>
<td></td>
<td>3L↑</td>
<td>3L%</td>
</tr>
<tr>
<td>4</td>
<td>5R</td>
<td>6R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5R1</td>
<td>5R0</td>
<td>2L*</td>
<td>5R*</td>
<td>5R*</td>
<td>5R↑</td>
<td>5R%</td>
</tr>
<tr>
<td>6</td>
<td>6R1</td>
<td>6R0</td>
<td>2L*</td>
<td>6R*</td>
<td>6R*</td>
<td>6R↑</td>
<td>6R%</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td>2STOP</td>
<td>7R1</td>
<td>7R0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8R1</td>
<td>8R0</td>
<td>8STOP</td>
<td>8R1</td>
<td>8R0</td>
<td>8R↑</td>
<td></td>
</tr>
</tbody>
</table>

Exhibit 1.3.2.1.2 illustrates the quintuple list, the initial configuration and the execution process for determining the placements of the Hamming error checks within the following binary string: 1010110011001111.
1.3.2.2 Post Machine

Post's 1936 independent formulation from Turing's, yet simultaneous and similar observation, is that he considered the tape to be two-way infinite and the alphabet to be composed of only two symbols: *(mark) and h (blank), instead of a finite number of symbols. Also, instead of a finite number of internal states, he considered a small number (n) of the instructions with the properties listed in Figure 1.42.

* mark the square being scanned
h erase the mark from the square being scanned
R move one square to the right
L move one square to the left
(h→R→* L) if the square being scanned is blank move to the right, if the square being scanned is marked move to the left.

Figure 1.3.2.2.1 The Set of Instructions Within Post's System.
1.3.2.2.1 Wang Version of Post Machine

The set of instructions in Figure 1.42 could be simplified to the set in Figure 1.43 as suggested by Wang:

* mark the square being scanned
R move one square to the right
L move one square to the left
If scanned square is blank proceed to next instruction, if scanned square is marked proceed to instruction whose number is specified.

Figure 1.3.2.2.1.1 The Simplified Version of the Set of Instructions in Figure 1.3.2.2.1.

1.3.2.2.2 Shepherdson and Sturgis Version of Post Machine

Shepherdson and Sturgis \(^9\), \(^{1963}\) were able to prove that the conditional statement in the set given in Figure 1.42 can be changed to the one given below if instructions with *'s are replaced by h (resulting in a dual set of instructions).

If scanned square is marked proceed to next instruction, if scanned square is blank proceed to instruction i.
They also designed another form of Turing machine with one tape, finite but unbounded, two symbols h, *, and operations in Figure 1.3.2.2.2.1.

\[
\begin{align*}
\text{print h at right-hand end,} \\
\text{print * at right-hand end,} \\
\text{scan and delete left-hand symbol, if *} \\
\text{proceed to the instruction whose number is specified, and if h, proceed to next instruction.}
\end{align*}
\]

Figure 1.3.2.2.2.1 Shepherdson and Sturgis Version of Turing Machine.

1.3.2.3 Minsky Machine

Minsky's\(^5,1960\) machine is another form of a simple executing machine composed of three registers (\(n_1, n_2\) and \(n_3\)) capable of storing non-negative integers and the set of instructions in Figure 1.3.2.3.1

\[
\begin{align*}
\text{n}_1 &\leftarrow \text{n}_1 + 1 \quad \text{increase n}_1 \text{ by one,} \\
\text{n}_1 &\leftarrow \text{n}_1 - 1 \quad \text{decrease n}_1 \text{ by one,} \\
\text{if n}_1 \neq 0 & \quad \text{proceed to the next instruction, and} \\
\text{if n}_1 = 0 & \quad \text{proceed to the j\textsuperscript{th} instruction.}
\end{align*}
\]

Figure 1.3.2.3.1 Instructions on Minsky Machine.
where \( i = 1, 2, 3 \) represents the register number and \( j = 1, \ldots, k \) represents the sequence number of the instructions.

1.3.2.4 Markov Normal Algorithms

For a given algorithm there exists a finite list of possible types of elementary steps, where to each type a pair of words in the alphabet of the algorithm corresponds.

If arrow (\( \rightarrow \)) is not a letter of the alphabet (in which the algorithm is expressed), the substitution of the word B for the first occurrence of the word A in any word to be transformed is shown as: \( A \rightarrow B \), (if A is not a subword of the word being transformed the machine goes to the next substitution of the list).

The essential constituent of an algorithm can be expressed by a finite list of such substitution formulas. The positions of the substitution formulas in the list determine their order in the sense that the first one in the list of substitutions applicable (a substitution formula is applicable if and only of the left-hand side of it is a subword of the word to be transformed) to a given word is chosen and the corresponding substitution to the first occurrence of the left-hand member of the word under consideration is applied. The process of successive substitutions according to the list of substitutions
continues* until either a concluding formula (A* B) or a word to which none of the substitution formulas is applicable is obtained.

The following normal algorithm converts a string of characters (composed of a's and b's) into its reverse order.

Alphabet:

List of substitution

1A:A1
1B:B1
2A:A2
2B:B2
*2:B*
*1:A*
*A:1
*3:2
*:*
:* *

Exhibit 1.3.2.4.1 illustrates the initial string and the execution process.

"After applying a substitution formula, the process is repeated from the top of the substitution list. Only if a substitution is not applicable is the next substitution in the list examined for applicability and applied if it is applicable."
Exhibit 1.3.2.4.1

ALPHABET: (AB12* )

INPUT: BABAA

EXECUTION PROCESS:

C: C) BABAA
1: 10) *BABAA
2: 8) 2ABAA
3: 3) A2BAA
4: 4) AB2AA
5: 3) ABA2A
6: 3) ABA2A
7: 10) *ABAA2
8: 7) 1BAA2
9: 2) B1AA2
10: 1) BA1A2
11: 1) BAA12
12: 10) *BAA12
13: 8) 2AA12
14: 3) A2A12
15: 3) AA212
16: 10) *AA212
17: 7) 1A212
18: 1) A1212
19: 10) *A1212
20: 7) 11212
31: 10) #11212
22: 6) A*1212
23: 6) AA*212
24: 5) AAB*12
25: 6) AABA*2
26: 5) AABAB*
27: 9) AABAB

< STOP >

OUTPUT: AABAB
Exhibit 1.3.2.4.2

Exhibit 1.3.2.4.2 illustrates the substitution list of the normal algorithm which can convert a string of characters (composed of a's, b's, c's, ..., z's) into its reverse order. The substitution list is generated by NAGEN program.
1.3.2.5 Transformation of One Machine to Another Machine

Since the enumeration of different ways of describing algorithms with a finite set of tools exists, there must be a method of transforming one machine to another equivalent one.

The following formulation allows the transformation of a Turing Machine into a right-left automation.  

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>h</th>
<th>s</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁</td>
<td>2Rh</td>
<td>2Ra</td>
<td>3LS</td>
</tr>
<tr>
<td>q₂</td>
<td>2RS</td>
<td>2RS</td>
<td>Stop</td>
</tr>
<tr>
<td>q₃</td>
<td>1Ra</td>
<td>1Rh</td>
<td>Stop</td>
</tr>
</tbody>
</table>

Figure 1.3.2.5.1 A Turing Machine.

Let q₁ correspond to r₁
q₂ correspond to r₂
q₃ correspond to r₃

Then the following substitutions correspond to the right-moving commands:

\[ r₁ \rightarrow ahr₂ \]
\[ r₁ \rightarrow h+ar₂ \]
\[ r₂ \rightarrow a+sr₂ \]
\[ r₂ \rightarrow h+sr₂ \]
\[ r₃ \rightarrow a+ar₁ \]
\[ r₃ \rightarrow h+hr₁ \]
The following set corresponds to the only left-moving command of the Turing machine:

\[
\begin{align*}
0 & \quad r_{1}s \rightarrow l_{3}s \\
0 & \quad s_{1}l_{3} \rightarrow s_{1}l_{3} \\
0 & \quad h_{1}l_{3} \rightarrow h_{1}l_{3} \\
0 & \quad a_{1}l_{3} \rightarrow a_{1}l_{3}
\end{align*}
\]

Given the input \( q_{1} \text{haaash} \) then the Turing machine and right-left automation would have to go through the following processes in order to generate the output.

<table>
<thead>
<tr>
<th>Turing Machine</th>
<th>Right-left automata</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_{1} \text{haaash} )</td>
<td>( q_{1} \text{haaash} )</td>
</tr>
<tr>
<td>( aq_{2} \text{aash} )</td>
<td>( r_{1} \text{haaash} )</td>
</tr>
<tr>
<td>( a sq_{2} \text{aash} )</td>
<td>( ar_{2} \text{aash} )</td>
</tr>
<tr>
<td>( , )</td>
<td>( asr_{2} \text{aash} )</td>
</tr>
<tr>
<td>( , )</td>
<td>( , )</td>
</tr>
<tr>
<td>( asssq_{2} \text{sh} )</td>
<td>( assr_{2} \text{ash} )</td>
</tr>
<tr>
<td>( assssh \text{stop} )</td>
<td>( asssr_{2} \text{sh} )</td>
</tr>
<tr>
<td></td>
<td>( assssh \text{stop} )</td>
</tr>
</tbody>
</table>

Another example to demonstrate the use of left-mover substitutions would be the input: \( q_{1} \text{sh} \)
<table>
<thead>
<tr>
<th>Turing machine</th>
<th>Right-left automata</th>
</tr>
</thead>
<tbody>
<tr>
<td>sq₁ sh</td>
<td>sr₁ sh</td>
</tr>
<tr>
<td>q₃ ssh</td>
<td>0</td>
</tr>
<tr>
<td>stop</td>
<td>sl₃ sh</td>
</tr>
<tr>
<td></td>
<td>sl₃ sh</td>
</tr>
<tr>
<td></td>
<td>r₃ ssh stop</td>
</tr>
</tbody>
</table>
1.3.2.6 Universal Turing Machine, 1967

The input of a stopping Turing machine is a function of its input and the Turing machine defines the function. If the algorithms are described in uniform finite terms, a universal Turing machine which, when supplied with the description of an algorithm and its input to be processed must exist to perform the same computations as described by the algorithm described.

1.3.2.7 Computational Limitation

Relaxing the finiteness restriction imposed on the elements of a machine does increase the machine's capabilities, nevertheless it does not provide the means of mechanizing the processing of all the functions within this universe.

There exist general problems which are not solvable algorithmically. There are an enumerable infinity \( \aleph_0 \) of algorithms so at most there are an enumerable infinity of functions, \( f \), from natural numbers to natural numbers which are computable (there exists an algorithm which for each \( n \) will, if started on \( n \), compute the value of \( f(n) \)). Since there are \( 2^{\aleph_0} \) such functions and \( 2^{\aleph_0} > \aleph_0 \) there must be non-computable functions from natural numbers to natural numbers.

For example a decision procedure to enable one to
determine where a Turing machine halts, given any machine \( T \) and tape \( t \), does not exist.\(^5\), 1961

Another unsolvable problem is to determine whether a machine ever prints the symbol \( s_0 \) when started on an input \( t \). If the machine (which does not use symbol \( s_0 \) or if it does it can always be made not to use it) is altered so that it would print the symbol \( s_0 \) before halting, the "printing" problem reduces to the "halting" problem which itself is unsolvable.

1.3.2.8 Conclusion

Turing machines are more capable than finite automata, but are not completely finite devices because of having access to an infinite memory. Thus, the concept of a finite automaton is more appropriate as far as practicality is concerned.

There are two major differences between Turing machines and finite automata if the input and output sequences are thought of as printed on successive squares of two tapes: \(^9\), 1963

(1) a finite automaton would leave the input unchanged and would print the output on a separate output tape;

(2) instead of mapping one cell to the right, left, or not move at all, a finite automaton
would move one cell to the right on both input and output tapes at each move (a Turing machine operates its own tape; it can reverse and alter it).

Clearly there exist machines of intermediate nature (between Turing and infinite machines).\textsuperscript{2,6,8,9, 1959}

For example if an automaton is made capable of erasing, the recognition of palindromes is made possible. With two tape automatons which are capable of stopping one tape and moving on the other, the correspondence decision problem (proposed to be unsolvable by Post)\textsuperscript{7, 1946} can be solved. DeLeeuw and his co-workers\textsuperscript{1} and Robin\textsuperscript{8} showed that probabilistic Turing machines can often do things more economically than non-probabilistic ones.

Because a primitive machine such as Turing's can be used to evaluate all of the computable functions, any algorithm regardless of complexity can be broken down into basic components. Thus automated means of solving problems (all of the solvable ones) can be established if

(1) a bound on the problem's memory requirements is imposed, and

(2) facilities to accomplish the basic tasks (the primitive operations forming the algorithms to solve the problems) are provided.
Let the automated means of solving those problems which meet the above condition (1) and belong to the class of computable functions be achieved through a machine called a computer.
BIBLIOGRAPHY


2. COMPUTER AND THE EVOLUTIONARY DEVELOPMENT OF PROGRAMMING LANGUAGES

2.1 Computer

2.1.1 Definition of Computer

A computer is a machine to automate processing of information by accepting a logically self-contained set of instructions (a program) and following the instructions without any human intervention to obtain a specific result.

2.1.2 Historical Perspective of Computer

"There is of course never an initial point for any history prior to which nothing of relevance happened and subsequent to which it did. It seems to be the nature of man's intellectual activity that in most fields one can always find by sufficiently diligent search a more or less unending regression back in time of early efforts to study a problem or at least to give it some very tentative dimensions." 6

So it is in the field of Computer Science.

Although ENIAC/Electronic Numerical Integrator and Calculator (the prototype of most of today's computers) did not come into existence until 1947, many devices whose evolutionary developments contributed to the achievement of today's machines did exist.
In general the history of computers can be divided into "generations" which depend mostly on the technology of the time.

**0 generation technology**
- wheels
- gears
- metal punched cards
- mechanical devices

**1st generation technology**
- relays
- vacuum tubes
- acoustic delay lines
- CRT memories
- magnetic tape
- magnetic drum

**2nd generation technology**
- transistors
- magnetic disk
- magnetic core memory
- line printers

**3rd generation technology**
- integrated circuits
- Hybrid, MOS, MSI
- high speed I/O devices
- impact printers, CRT, etc.
4th generation technology

LSI

Bubble memories

thin film memories

Lasar memories

Josephson junctions
2.1.2.1 Historical Survey of 0 Generation Technology

2.1.2.1.1 Gerbert

Gerbert d' Aurillac, Pope Sylvester II (945-1003), an Italian mathematician, designed an abacus composed of 27 columns combined in groups of three. From left to right they were marked C (hundred), D (ten), S or M (one), other columns marked with other letters were for higher decimal units. The numbers were expressed by counters carrying symbols equivalent to Moslems' numerals.

2.1.2.1.2 Magnus

Spaniard Magnus (who lived about the same time as Gerbert) designed a calculator which was shaped like a human head. The figures appeared in the mouth of the head where the teeth should be.

2.1.2.1.3 Al-Kashi

Some of the contributions made toward the automation of computation are due to the special-purpose instruments invented by Jamshid ben Masoud ben Mahmoud Ghiyath ed-Din Al-Kashi (1393-1449), an Iranian astronomer and mathematician of the fifteenth century.

Al-Kashi made contribution to the problems of the summation of the fourth powers of the natural numbers, trigonometric computation, and approximation. The first
use of a decimal fraction and computation of the value of $2\pi$ to sixteen decimal places are also his achievements.

Al-Kashi's "plate of conjunctions" was designed to determine the exact time at which two planets would be in conjunction (the time at which two planets would have the same longitude). His lunar eclipse device was designed to simplify the calculation of the important times associated with lunar eclipses. He also created a planetary computer which was used to determine the longitudes of the sun, the moon, and the visible planets.

Figure 2.1.2.1.3.1 Al-Kashi's Instrument.
Figure 2.1.2.1.3.2 Al-Kashi's Instrument Setup for Finding the Sun's True Longitude.

Figure 2.1.2.1.3.3 Al-Kashi's Instrument Setup for Finding a Lunar Eclipse.
John Napier (1550-1617) a Scottish mathematician realized the necessity of mechanizing multiplication, divisions, and evaluation of square and cubical expansions (because of their time consuming processes). He conceived the idea that all numbers could be written in exponential form and then created logarithm tables providing the means for faster computation. He also invented a calculator called Napier's bones; combined by his algorithmic tables made a significant contribution to computation by mechanical means.
2.1.2.1.5 Gunther⁴,⁵,⁷

Edmund Gunther (1581-1626), an English mathematician, plotted Napier's logarithms on a device called a slide roll allowing the multiplication and division to be performed by the addition and subtraction of lengths by means of a pair of dividers. William Oughtred improved Gunther's instrument.

2.1.2.1.6 Schickard⁴,⁵,⁶,⁷,¹²

Wilhelm Schickard (1592-1635), a German professor of astronomy, mathematics, and Hebrew in Tübingen, sketched and described a machine capable of performing addition, subtraction and partially automatically multiplication and division in 1623.

Figure 2.1.2.1.6.1 Schickard Calculating Machine⁶,p.120
Blaise Pascal (1623-1662), a French scientist, contributed significantly to the world of science in his 39 short years of life. By the age of 16 he wrote a paper that was claimed as the beginning of modern geometry. When he was 20 (in 1642) he built a calculating machine capable of performing additions and subtraction (not as advanced as Schickard's in performing multiplication and division).

Figure 2.1.2.1.7.1 Pascal's Calculating Machine. 6, p.120
Sir Samuel Morland (1625-1695), an English inventor, designed three calculating machines: a trigonometrical calculating machine (in 1663), a calculator similar to Pascal's (in 1666), and a machine for multiplication and division.

Figure 2.1.2.1.8.1 Morland's Adding Machine. 7, p. 34
Gotfried Leibniz (1646-1716), a German universalist, built a calculating machine in 1671 which was superior to Pascal's and was the first general purpose calculating device able to meet the major needs of mathematicians and bookkeepers. The binary number system so important in present computers was also envisioned by Leibniz.
2.1.2.1.10 Roven

Grillet de Roven, a French inventor, designed a mechanical calculator, in 1678, with a carrying mechanism.

2.1.2.1.11 Poleni

Giovanne Poleni (1683-1761), an Italian Mathematician, physicist, ancient historian and archeologist, designed a calculating machine, in 1709, capable of being programmed by the arrangement of levers.

2.1.2.1.12 Stanhope

Charles Stanhope (1753-1816), an English inventor, invented three calculating machines: a multiplying machine, in 1775, in which Leibniz' stepped-reckoner device was used, a multiplying machine, in 1777, in which a cam plate for
the use of variable number of teeth was used, and an
improved version of Morland's machine for working problems
in inductive logic.

2.1.2.1.13 Boole^{4,5,6,7,12}

George Boole (1815-1864), an English logician and
mathematician, was the principal developer of the algebra
of logic: the expression and manipulation of logic by
mathematical symbols.

2.1.2.1.14 Jacquard^{4,5,6,7,12}

Joseph Marie Jacquard a Lyons textile mill worker
introduced his automated loom in 1805. The loom's activi-
ties were controlled by a device which accepted patterns
by means of punched cards.
Charles Babbage (1792-1871), an English scientist, designed the Difference Engine Machine, a machine to provide automatic means of producing mathematical tables. The machine was composed of a number of striking clocks in a row, each of which having one hour hand. If the hand of one clock was turned to a number on the dial, the clock struck that number of times, moving the hand of the clock on its right one number per strike.

To generate a table containing the cubes of integers, the machine had to be programmed as follows:

\[
\begin{array}{cccccc}
0 & 0 & 0 & 0 & 0
\end{array}
\]
the third clock was set to 1, causing the fourth wheel to be set to 1:

```
0 0 1 1
```

the second clock was set to 6, causing the third wheel to be set to seven and the first wheel to be set to 8:

```
0 6 7 8
```

the first clock was set to 6, causing the clocks to be set as follows:

```
6 12 19 27
```

then the right most clock (circled) represented the cubes of natural numbers.
After the creation of the Difference Engine, Babbage proceeded with the design of Analytical Engine: "a machine that could bite its own tail," as he stated himself. The machine was to be capable of using its output for retrieving further information from the memory, the retrieved
information, in turn, was to be fed back into the machine. It was to be composed of:

- **Memory:** 1000 counting wheels, each representing 50 digits
- **Control:** Jacquard punch cards, branching ability (back up and go forward)
- **Arithmetic:**
  - 1 second add/subtract
  - 1 minute 50 by 50 multiplication
- **Input:** punched cards
- **Output:** punched cards and printed copy
- **Table Storage:** random access to Jacquard punch cards, i.e., rang a bell and displayed number of desired card on counting wheels and then the operator supplied the card
- **Programming:** Lady Ada Augusta Lovelace was the first programmer, working with Babbage.

Figure 2.1.2.1.15.2 Babbage's Analytical Engine. 12, p. 68

Although Babbage never produced a complete prototype of the Analytical Engine, he gave birth to the concept
of the modern computer.

2.1.2.1.16 Hollerith

Herman Hollerith (1860-1929), an American Census Bureau, designed a tabulating machine (1884-1890) capable of sensing the holes in cards (similar to Jacquard's) through electrical contacts.

Figure 2.1.2.1.16.1 Hollerith's Tabulating Machines and Sorting.
2.1.2.2 Historical Survey of 1st Generation and Thereafter Technology

2.1.2.2.1 Z1 and Z3

1938 Z1 Computer (the first program-controlled computer)
   Designer:
   Konrad Zuse - German

1941 Z3 Computer (the first program-controlled computer in Germany, designed before Z4)
   Designer:
   Konrad Zuse - German

Memory:
   64, 22 words of relay memory
   floating point 2 7 14
      sign exp mantissa

Control:
   decoded instructions in command register
   8 bit operation code
   no means of conditional branching

Arithmetic:
   floating-point addition, subtraction, multiplication, and division
   Complementation

I/O:
   input: punched tape and keyboard
   output: display lights
2.1.2.2.2 Atanosoff-Berry

1938-1942 Atanasoff-Berry Computer

Designer: Iowa State University Agricultural Experiment Station

Memory:
- capacitor drum rotating at 60 Hz
- punch cards for intermediate results

Control:
- unknown

Arithmetic:
- binary arithmetic
- solve 30 simultaneous linear equations with 30 variables

I/O:
- punch card input

2.1.2.2.3 MARK I

1939-1944 Harvard MARK I Calculator

Designers:
- Howard Aiken and IBM - American

Memory:
- 60 constant registers each of which composed of 24 ten position switches
- 72 storage registers each of which composed of 24 electromechanical counter wheels

Control:
- Sequence mechanism composed of a sprocket drum and a perforated paper tape
Arithmetic:

addition  0.3 seconds 
multiplication  1 second 
division  5 seconds 
functions:  $\log x$, $10^x$, and $\sin x$

I/O:

input - punched cards, switches 
output - punched cards and electronic typewriter

2.1.2.2.4 ENIAC

1943 ENIAC/Electronic Numerical Integrator and Computer

Designers:

Eckert, Mauchly - American

Memory:

decade rings 
flip flops

Control:

parallel operations 
means of conditional branching 
discrimination of the sign of a number 
plug board-variable wiring 
30 minute-day to plug up a program

Arithmetic:

0.2 millisecond addition 
2.8 millisecond multiplication (1000 times faster than MARK I)
I/O:

Card reader
Card punch

2.1.2.2.5 EDVAC

1946 EDVAC/Electronic Discrete Variable Automatic Computer

Designer:

Eckert, Mauchly, and Von Neuman

Memory:

binary
mercury delay lines
1024, 44-bit words/128 thermostatically controlled acoustic delay lines

Control:

serial and synchronous operations
stored program concept
4-address instructions
2 operands, result, next instruction

Arithmetic:

864 microseconds addition
2900 microseconds multiplication

subtraction was done directly, not through complement

I/O:

paper tapes, teletype writers, and punched cards
2.1.2.2.6 EDSAC

1947 EDSAC/Electronic Delay Storage Automatic Computer

Designer:

Maurice Wilkes, Cambridge University

Memory:

1024 binary, 17 bit words (one being a sign bit)
mercury delay

Control:

single address instructions

I/O:

paper tape input and teletype writer output

2.1.2.2.7 MADM

1948 MADM/Manchester University Computer Prototype

Designer:

Manchester University

Memory:

128, 40 bit Williams Tube (CRT)
1024 40 bit words Magnetic Drum Storage

Control:

indexed address
B tube (Williams Tube) as storage for index (8)
decrement B tube capability
2.1.2.2.8 SEAC

1946-50 SEAC/Standards Electronic Automatic Computer

Memory:
44 bit words
mercury delay
Williams tubes

Control:
4 address instructions

Arithmetic:
serial

I/O:
flexowriter

Hardware:
pulse circuitry
reduced power
modular design

2.1.2.2.9 ERA 1101

194 ERA 1101

Designer:
Engineering Research Associates

Memory:
magnetic drum
data arranged for quick access
1946 IAS/Princeton Institute for Advance Studies

Designer:

Von Neuman

Memory:

hierarchical: 40 selectrons (Williams Tubes) as main memory and magnetic wire as secondary memory

40 bit data word stored with 1 bit in each selectron with 4000 words total

separate instructions to read/write main and secondary memory

Control:

stored program concept

data and instructions held in same storage medium

branching, conditional and unconditional

matrix decoding of operation codes

program counter

decode register

instruction register

2 instructions/word

Arithmetic:

parallel

fixed point chosen for simplicity in arithmetic logic and accuracy of data

accumulator register

arithmetic register
2's complement
carry save addition

I/O:

secondary storage treated as I/O device

2.1.2.2.11 UNIVAC

1951 UNIVAC/Universal Automatic Computer
Designers:
Eckert, Mauchly and ERA
Memory:
1000, 12-decimal-digit words
12 delay lines as input/output registers
Control:
stored program concept
45 instructions/6-bit operation codes
self-checked circuits
parity checks
duplex processor
I/O:
magnetic tape and line printer

2.1.2.2.12 Whirlwind I

1951 Whirlwind I
Memory:
2048 16 bit registers
1 μsec cycle time
Control:

32 instructions
20K instructions/sec
single address
distributed control with separate control for operations, storage, arithmetic, and I/O.

2.1.2.3 Important Milestones

1949  Index Registers - Manchester University MADM
1950  Core Memory-Forrester MIT/WhirlWind I
1952  Interrupts-IBM 1103
1953  Microprogramming-Wilkes, EDSAC
1953  FORTRAN
1955  Indirect Addressing-IBM 704
1957  Moving Head Disk-IBM 305
1957  Duplexed Computer-SAGE
1958  I/O Processor-IBM 709
1960  Instruction lookahead-IBM 7090
1961  Multiprocessor-Burroughs D825
1961  Instruction lookahead-IBM STRETCH
1962  Paging-Manchester Univ. ATLAS
1964  Instruction Stack-CDC 6600
1965  Pipeline-IBM 360/91
1968  Array Processing-ILLIAC IV

The developments of EDVAC at the University of Pennsylvania and the EDSAC at Cambridge, became the basis
for the present digital computer design. Spreads of computers have increased to the point where the CDC 3300/OS-3 computer system at Oregon State University, a medium scale system, has the following configuration:

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core Memory</td>
<td>98304 24-bit words with 1.6 microsecond cycle time</td>
</tr>
<tr>
<td>Tape Drive (4)</td>
<td>150 inches/second with 200, 556, or 800 bits/inch density</td>
</tr>
<tr>
<td>Mass Storage Disk File</td>
<td>268435456 with maximum seek time of 100 milliseconds</td>
</tr>
<tr>
<td>Disk Drive (4)</td>
<td>28 000 000 characters/drive</td>
</tr>
<tr>
<td>Teletype (250)</td>
<td></td>
</tr>
<tr>
<td>CRT (16)</td>
<td></td>
</tr>
<tr>
<td>Card Reader</td>
<td>1200 136-character lines/minute</td>
</tr>
<tr>
<td>Card Punch</td>
<td>250 80-column cards/minute</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


2.1.3 Classification of Computer

Depending on the application in which a computer is used, the following classification may be established:

(1) The Special Purpose Computer, which is designed for a specific class of problems such as the missile or elevator traffic control computers, and

(2) The General Purpose Computer, which is hardly limited in the number of applications for which it can be used (it is capable of simulating a universal Turing machine).

The means by which quantities and variables are presented within the computer can also be a basis for its classification, such as:

(1) The Digital Computer, which is capable of expressing quantities and variables within it in terms of numbers: discrete (discontinuous) data,

(2) The Analog Computer, which uses physical analogies to represent quantities and variables. It translates physical conditions such as flow, temperature, pressure and angular positions into related mechanical and electrical quantities. The information processed by this class of computers is of continuous nature (such as the speedometer of a car: a simple
special purpose analog computer), and

(3) The Hybrid Computer, which combines the features of digital and analog computers in order to maximize the overall computational economy and efficiency.

Throughout this thesis, the general purpose digital computers (referred to simply as computers) are considered.

2.1.4 Structure of Computer

A typical computer, from hardware point of view, is composed of

(1) Memory,
(2) Processor,
(3) Control,
(4) Input, and
(5) Output
sub-systems.
Figure 2.1.4.1 General Hardware Organization of a Computer.
2.2 Programming Languages

2.2.1 Instruction Repertoire of Computer

The repertoire of the machine can be composed of a set of instructions to perform a few primitive operations (such as the instructions in Figure 2.2.1.1).

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>(AR) + (MEM)</td>
</tr>
<tr>
<td>2.</td>
<td>(AR) + CONSTANT</td>
</tr>
<tr>
<td>3.</td>
<td>(MEM) + (AR)</td>
</tr>
<tr>
<td>4.</td>
<td>(AR) + (AR) + (MEM)</td>
</tr>
<tr>
<td>5.</td>
<td>(AR) + (AR) - (MEM)</td>
</tr>
<tr>
<td>6.</td>
<td>(AR) + (AR) * (MEM)</td>
</tr>
<tr>
<td>7.</td>
<td>(AR) + (AR) / (MEM)</td>
</tr>
<tr>
<td>8.</td>
<td>(PC) + MEM if (AR) = 0</td>
</tr>
<tr>
<td>9.</td>
<td>(PC) + MEM if (AR) &lt; 0</td>
</tr>
<tr>
<td>10.</td>
<td>(PC) + MEM unconditionally</td>
</tr>
<tr>
<td>11.</td>
<td>(PC) + (PC) dynamic halt</td>
</tr>
</tbody>
</table>

where

AR = accumulator register
MEM = specified memory location
PC = program counter (contains the address of next instruction to be executed)
(...) = contents of ...

Figure 2.2.1.1 The Basic Instructions Within the Repertoire of a Typical Computer.

The instruction set in Figure 2.2.1.1 is sufficient to provide means of solving those problems in which the bounds on their memory requirements do not exceed the total available memory and belong to the class of computable functions. It suffices to show that the instructions within the set in Figure 2.2.1.1 are capable of simulating all that can be done by a Turing machine.
In order to simulate a Turing machine, the following Turing machine components and behavior must be provided:

1. a Turing tape, where the input data, intermediate output formed during the course of computation and final output are stored; of course with the restriction that the tape length is finite;
2. a read/write head, for inputting/outputting symbols onto/from the tape and moving to an adjacent square;
3. a finite number of distinct internal states (configurations) each of which are capable of determining the behavior of the machine in conjunction with the symbol of the tape read/write, and
4. an alphabet, a finite set of distinct elements to be input and/or output.

To provide the Turing tape, a portion of the computer memory whose size determines the maximum possible length of the Turing tape is used. Then the tape symbols can be stored in successive memory locations of that portion of the memory.

The existence of the read/write head can be simulated by a pointer, stored in a memory location: \( Q \) which always points to the memory location in which the current tape symbol is held. The head movement can be kept track of
by altering the contents of Q appropriately: increasing its contents by one for a right move, and decreasing its contents by one in the case of a left move. The symbols of the tape could be read or written onto by the appropriate collection of the instructions in Figure 2.2.1.1 and also the use of Q.

For example, the following action of a Turing machine (the partial task accomplished by a quintuple which involves read/write head activities):

if the symbol $s_3$ is read then replace it by symbol $s_0$ and move one cell to the right; otherwise leave the symbol unchanged and move one cell to the left

can be simulated as follows:

<table>
<thead>
<tr>
<th>Computer Memory</th>
<th>Turing Tape</th>
</tr>
</thead>
<tbody>
<tr>
<td>addresses</td>
<td>$s_0 s_1 s_2 s_3 s_4 s_5 s_6$</td>
</tr>
<tr>
<td>Turing tape pointer</td>
<td>read/write head</td>
</tr>
<tr>
<td>$a_0$</td>
<td>$s_0$</td>
</tr>
<tr>
<td>$a_1$</td>
<td>$s_1$</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$s_2$</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$s_3$</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$s_4$</td>
</tr>
<tr>
<td>$a_5$</td>
<td>$s_5$</td>
</tr>
<tr>
<td>$a_6$</td>
<td>$s_6$</td>
</tr>
<tr>
<td></td>
<td>$a_3$</td>
</tr>
</tbody>
</table>

$Q$


0 \( (AR) + ((Q)) \) load the symbol pointed by Q

1 \( (AR) + (AR) - (MEM) \) determine if the loaded symbol is \( s_3 \) (stored in \( MEM_0 \))

2 \( (PC) + 4 \) if \( (AR) = 0 \)

3 \( (PC) + 10 \)

4 \( (AR) + (MEM) \) replace \( s_3 \) by \( s_0 \) (stored in \( MEM_1 \))

5 \( ((Q)) + (AR) \)

6 \( (AR) + (MEM) \) move right; a 1 is stored in MEM

7 \( (AR) + (AR) + (Q) \)

8 \( (Q) + (AR) \)

9 \( (PC) + 12 \) transfer control to location 13 to continue on with the other operations, leave the symbol unchanged and move one cell to the left

10 \( (AR) + (Q) \)

11 \( (AR) + (AR) - (MEM) \)

12 \( (Q) + (AR) \)

13 Continue with the other operations in sequence

The means for keeping track of the state in which the machine operates (before and after processing a quintuple) can be provided by a memory location whose contents are always set to the current state number, \( Q \).

To make the simulator completely automated, in addition to the above design specifications, a command translator to decode the Turing table commands into the elementary tasks is also required which may be designed as follows:
Figure 2.2.1.2 The Components of the Turing Commands Encoder.
Indeed the set composed of the operations 1, 2, 3, 4, 8, 9 of Figure 2.2.1.1 and another instruction (to complement the (AR)) can accomplish all that can be done by the set in Figure 2.2.1.1 and thus is sufficient to express any algorithm.

1. \( (AR) \rightarrow (MEM) \)
2. \( (AR) \rightarrow \text{CONSTANT} \)
3. \( (MEM) \rightarrow (AR) \)
4. \( (AR) \rightarrow (AR) + (MEM) \)
5. \( (PC) \rightarrow \text{MEM if } (AR) = 0 \)
6. \( (PC) \rightarrow \text{MEM if } (AR) < 0 \)
7. \( (AR) \rightarrow (AR)' \)

Figure 2.2.1.3 Basic Instructions Sufficient to Express Any Algorithm.

Operation \( (AR) \rightarrow (AR) - (MEM_i) \) of Figure 2.2.1.1 can now be simulated by the following operations:

\( (MEM_i) \rightarrow (AR) \) store the first operand

\( (AR) \rightarrow (MEM_i) \) load the second operand

\( (AR) \rightarrow (AR)' \) complement the second operand

\( (AR) \rightarrow (AR) + (MEM_i) \) add the first operand and the complemented representation of the second operand.

The following ordered set of instructions implement the algorithm in Figure 2.2.1.4 which simulates the multiplication operation: \( (AR) \rightarrow (AR) \times (MEM_i) \).
Figure 2.2.1.4 The Algorithm to Simulate the Multiplication Process.
0. (MEMJ) ← (AR)
1. (AR) ← 0
2. (FLAG) ← (AR)
3. (MEMR) ← (AR)
4. (AR) ← (MEMI)
5. (MEMK) ← (AR)
6. (AR) ← 1
7. (AR) ← (AR)'
8. (MEMN) ← (AR) store -1 in location MEMN
9. (AR) ← (MEMJ)

10. (PC) ← 39 if (AR) < 0 the first operand is negative
11. (PC) ← 46 if (AR) = 0 the first operand is zero; the result is zero
12. (AR) ← (MEMK)
13. (PC) ← 28 if (AR) < 0 the second operand is negative
14. (PC) ← 45 if (AR) = 0 the second operand is zero; the result is zero
15. (AR) ← (MEMN) begin to add successively
16. (AR) ← (AR) + (MEMJ)
17. (MEMJ) ← (AR)
18. (PC) ← 23 if (AR) < 0
19. (AR) ← (MEMR)
20. (AR) ← (AR) + (MEMK)
21. (MEMR) + (AR)
22. (PC) + 15
23. (AR) + (FLAG) the successive addition is completed; test the FLAG for the sign
24. (PC) + 45 if (AR) = 0
25. (AR) + (MEMR) complement the result; one of the operands has been negative.
26. (AR) + (AR)'
27. (PC) + 46
28. (AR) + (FLAG) the second operand is negative; check to see if the first operand is also negative
29. (PC) + 36 if (AR) = 0
30. (AR) + 0 both operands are negative; clear the FLAG and complement the second operand
31. (FLAG) + (AR)
32. (AR) + (MEMK)
33. (AR) + (AR)'
34. (MEMK) + (AR)
35. (PC) + 15
36. (AR) + 1 only the second operand is negative; set the FLAG and complement the second operand
37. (FLAG) + (AR)
38. (PC) + 32
39. (AR) + (MEMJ) the first operand is negative; complement the first operand and set the FLAG

40. (AR) + (AR)'

41. (MEMJ)+ (AR)

42. (AR) + 1

43. (FLAG)+ (AR)

44. (PC) + 12

45. (AR) + (MEMR)

46. EXIT
The following instructions follow an algorithm similar to that in Figure 2.2.1.4 which uses successive subtractions instead of successive additions, in order to simulate the process of division: \((AR)÷(AR)÷(MEMI)\).

0. \((MEMJ) + (AR)\)
1. \((AR) + 0\)
2. \((\text{FLAG}) + (AR)\)
3. \((\text{MEMR}) + (AR)\)
4. \((AR) + (MEMI)\)
5. \((\text{MEMK}) + (AR)\)
6. \((AR) + (MEMJ)\)
7. \((\text{PC}) + 38\text{ if } (AR) < 0\)
8. \((\text{PC}) + 43\text{ if } (AR) = 0\)
9. \((AR) + (\text{MEMK})\)
10. \((\text{PC}) + 30\text{ if } (AR) < 0\)
11. \((\text{PC}) + 43\text{ if } (AR) = 0\text{ if the second operand is 0}
     \text{the result is 0}\)
12. \((AR) + (\text{MEMI})\)
13. \((AR) + (AR)'\)
14. \((\text{MEMK}) + (AR)\)
15. \((AR) + (MEMJ)\)
16. \((AR) + (AR) + (\text{MEMK})\)
17. \((\text{MEMJ}) + (AR)\)
18. \((\text{PC}) + 23\text{ if } AR < 0\)
19. \((AR) + 1\)
20. \( (AR) \leftarrow (AR) + (MEMR) \)
21. \( (MEMR) \leftarrow (AR) \)
22. \( (PC) \leftarrow 15 \)
23. \( (AR) \leftarrow (FLAG) \)
24. \( (PC) \leftarrow 28 \) if \((AR) = 0\) the division is completed check the flag
25. \( (AR) \leftarrow (MEMR) \)
26. \( (AR) \leftarrow (AR)' \)
27. \( (PC) \leftarrow 43 \)
28. \( (AR) \leftarrow (MEMR) \)
29. \( (PC) \leftarrow 43 \)
30. \( (AR) \leftarrow (FLAG) \) the second operand is negative; check to see if the first operand is also negative
31. \( (PC) \leftarrow 34 \) if \((AR) = 0\)
32. \( (AR) \leftarrow 0 \)
33. \( (FLAG) \leftarrow (AR) \)
34. \( (AR) \leftarrow (MEMK) \)
35. \( (AR) \leftarrow (AR)' \)
36. \( (MEMK) \leftarrow (AR) \)
37. \( (PC) \leftarrow 12 \)
38. \( (AR) \leftarrow (AR)' \)
39. \( (MEMJ) \leftarrow (AR) \)
40. \( (AR) \leftarrow 1 \)
41. \( (FLAG) \leftarrow (AR) \)
42. \( (PC) \leftarrow 9 \)
43. EXIT
operation \((PC)+\text{MEM}\) unconditionally can be replaced by \\
\((AR)+0\)
\[(PC)+\text{MEM}\] if \((AR) = 0\)
and the following instructions can be substituted for
\((PC)+\text{(PC)}\), the dynamic halting
\[n \ (AR)+0\]
\[n+1 \ (PC)+n\] if \((AR) = 0\)

Hence, the instruction set in Figure 2.2.1.2 is
equivalent to that in Figure 2.2.1.1 in the sense that they
both can express all effective algorithms. Nevertheless,
the instruction set 2.2.1.1 is more efficient than the
instruction set of Figure 2.2.1.2 if the speed with which
the algorithm is executed and the number of locations in
which the program instructions must reside are the basis
of such judgement; although more circuitry and thus higher
cost of design is required for the direct (hardware)
interpretation of the instruction set of Figure 2.2.1.1.

Still further reduction may be imposed upon the
instruction set of Figure 2.2.1.2.
2.2.2 Machine Language Programming

The machine instructions are represented by different bit configurations (operation codes) as may be arranged in one of the many possible codes, in Figure 2.2.2.1.

<table>
<thead>
<tr>
<th>Operation Code</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 001</td>
<td>(AR) + (MEM)</td>
</tr>
<tr>
<td>000 010</td>
<td>(AR) + CONSTANT</td>
</tr>
<tr>
<td>000 011</td>
<td>(MEM) + (AR)</td>
</tr>
<tr>
<td>000 100</td>
<td>(AR) + (AR) + (MEM)</td>
</tr>
<tr>
<td>000 101</td>
<td>(AR) + (AR) - (MEM)</td>
</tr>
<tr>
<td>000 110</td>
<td>(AR) + (AR) * (MEM)</td>
</tr>
<tr>
<td>000 111</td>
<td>(AR) + (AR) / (MEM)</td>
</tr>
<tr>
<td>001 000</td>
<td>(PC) + MEM if (AR) = 0</td>
</tr>
<tr>
<td>001 001</td>
<td>(PC) + MEM if (AR) &lt; 0</td>
</tr>
<tr>
<td>001 010</td>
<td>(PC) + MEM unconditionally</td>
</tr>
<tr>
<td>001 011</td>
<td>(PC) + (PEC) (dynamic halt)</td>
</tr>
</tbody>
</table>

Figure 2.2.2.1 Machine Instructions from Figure 2.2.1.1 and Their Corresponding Operations Codes.

Whether the human problem is stated in vague and inexact form or in a self-contained formulation, it must be broken into elementary task components, each of which is performable by a command of the instruction set, if it is to be executed algorithmically.
The elementary tasks must be expressed in the language of the machine (chain of zeroes and ones). Figure 2.2.2.2 illustrates a machine language program which uses the instructions in Figure 2.2.1.1 to raise the contents of a memory location 748 to the power of the positive integer stored in location 758 and store the result in location 768. The program follows the method of successive multiplication, illustrated in Figure 2.2.2.3.
<table>
<thead>
<tr>
<th>Location</th>
<th>Operation Code</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 000 000</td>
<td>000 010</td>
<td>000 000 001</td>
</tr>
<tr>
<td>000 000 001</td>
<td>000 011</td>
<td>000 111 011</td>
</tr>
<tr>
<td>000 000 010</td>
<td>000 010</td>
<td>000 111 101</td>
</tr>
<tr>
<td>000 000 011</td>
<td>000 011</td>
<td>000 111 010</td>
</tr>
<tr>
<td>000 000 100</td>
<td>000 010</td>
<td>000 111 100</td>
</tr>
<tr>
<td>000 000 101</td>
<td>000 011</td>
<td>000 111 110</td>
</tr>
<tr>
<td>000 000 110</td>
<td>000 010</td>
<td>000 111 010</td>
</tr>
<tr>
<td>000 000 111</td>
<td>000 101</td>
<td>000 111 011</td>
</tr>
<tr>
<td>000 001 000</td>
<td>001 000</td>
<td>000 001 101</td>
</tr>
<tr>
<td>000 001 001</td>
<td>000 011</td>
<td>000 111 010</td>
</tr>
<tr>
<td>000 001 101</td>
<td>000 101</td>
<td>000 111 110</td>
</tr>
<tr>
<td>000 001 011</td>
<td>000 110</td>
<td>000 111 100</td>
</tr>
<tr>
<td>000 001 100</td>
<td>001 010</td>
<td>000 000 101</td>
</tr>
<tr>
<td>000 001 101</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2.2.2  Machine Language Program Using the Instruction Set in Figure 2.2.1.1.
**Figure 2.2.2.3** The Algorithm for the Machine Language of Figure 2.2.2.2.
There are obvious disadvantages in expressing algorithms in machine language.

(1) Working with instructions in a long string of binary digits or even in a string of octal or even hexadecimal digits is a burden on the experienced programmer;

(2) inserting or deleting an instruction may cause the location of all the succeeding instructions and constants to change because usually the instructions must reside in consecutive memory locations (without unused ones) and each instruction which refers to any of the succeeding instructions or data must also be changed;

(3) there are difficulties in documentation and thus in the maintenance and extension due to the illegibility of the source program;

(4) there are difficulties with the transferability due to the dependence of machine language programs on the details of the machine for which they are originally developed.

The above difficulties would be magnified when large machine language programs are dealt with. Two possible ways of circumventing the problems caused by the gap between the languages of machines and the languages of human problems seem attractive:
(1) designing high-level language machines; i.e. machines whose means of communication are closely related to the problems encountered, and

(2) designing intermediate languages between languages of machines and the problems encountered.

2.2.3 High-Level Language Machines

To design high-level language machines would mean to develop machines which would communicate by means of a high-level language rather than communicating by a set of primitive commands (similar to that given in Figure 2.7). Although this would reduce the difficulties with the ordinary machine language programming there are at least two drawbacks opposed to it:

(1) the insufficiencies and the application orientation of the high-level languages in existence, and

(2) the high cost of circuitry.

Nevertheless, it is to be the ultimate goal as far as the future of computers is concerned and, as more satisfactory high-level languages are designed and the cost of circuitry decreases, the idea becomes more and more attractive and feasible.
2.2.4 Intermediate Languages

The second possibility, expressing the algorithms in intermediate steps between problems and machine by means of a high-level programming language, seems to be the most effective technique to date. Particularly, when the improvement of high-level languages is the step that must be taken before achievement of the ultimate goal: The development of high-level language machines.

2.2.5 Assembly Language Programming

In searching for a more convenient language than machine language to express algorithms, assembly languages (symbolic machine languages) were developed. These languages were designed to eliminate the difficulties associated with machine languages. They closely resemble the language of machines, except the operation codes and operands are given symbolic (mnemonic) names so that working with strings of zeroes and ones and absolute addresses is eliminated. A set of symbols can be associated with the operation codes in Figure 2.2.2.1 as arranged in Figure 2.2.5.1.

Using the mnemonic symbols in Figure 2.2.5.1 the machine language program in Figure 2.2.2.2 can be written as in Figure 2.2.5.2.
<table>
<thead>
<tr>
<th>operation code</th>
<th>symbolic operation code</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 001</td>
<td>LOAD</td>
</tr>
<tr>
<td>000 010</td>
<td>ENTER</td>
</tr>
<tr>
<td>000 011</td>
<td>STORE</td>
</tr>
<tr>
<td>000 100</td>
<td>ADD</td>
</tr>
<tr>
<td>000 101</td>
<td>SUBTRACT</td>
</tr>
<tr>
<td>000 110</td>
<td>MULTIPLY</td>
</tr>
<tr>
<td>000 111</td>
<td>DIVIDE</td>
</tr>
<tr>
<td>001 000</td>
<td>JUMP = 0</td>
</tr>
<tr>
<td>001 001</td>
<td>JUMP &lt; 0</td>
</tr>
<tr>
<td>001 010</td>
<td>JUMP</td>
</tr>
<tr>
<td>001 011</td>
<td>HALT</td>
</tr>
</tbody>
</table>

Figure 2.2.5.1  Machine Instructions Operation Codes and Their Corresponding Mnemonic Symbols.
<table>
<thead>
<tr>
<th>location</th>
<th>operation code</th>
<th>operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>ENTRY</td>
<td>BEGIN</td>
</tr>
<tr>
<td>POWER</td>
<td>DEC</td>
<td>5</td>
</tr>
<tr>
<td>RESULT</td>
<td>DEC</td>
<td>10</td>
</tr>
<tr>
<td>MINUSONE</td>
<td>DEC</td>
<td>-1</td>
</tr>
<tr>
<td>COUNTER</td>
<td>BSS</td>
<td>1</td>
</tr>
<tr>
<td>BEGIN</td>
<td>LOAD</td>
<td>NUMBER</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>RESULT</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>POWER</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>COUNTER</td>
</tr>
<tr>
<td>LOOP</td>
<td>LOAD</td>
<td>COUNTER</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>MINUSONE</td>
</tr>
<tr>
<td></td>
<td>JUMP = 0</td>
<td>END</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>COUNTER</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>RESULT</td>
</tr>
<tr>
<td></td>
<td>MULT</td>
<td>NUMBER</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>RESULT</td>
</tr>
<tr>
<td></td>
<td>JUMP</td>
<td>LOOP</td>
</tr>
<tr>
<td>END</td>
<td>HALT</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2.2.5.2  Symbolic Machine Language Program Equivalent to the Machine Language Program in Figure 2.2.2.2.

The above assembly language program is composed of two types of instructions: machine instructions (LOAD, STORE, ADD, JUMP = 0, MULT, and HALT) and instructions which give information to the assembler such as the address of the first executable statement (ENTRY), constants (DEC), request for reservation of memory location (BSS), and indication as to the physical end of program (END).
2.2.5.1 Assembler

Programs written in assembly languages impose the necessity of going through a transition phase before they can be executed. The programs which translate symbolic machine languages into their equivalent machine language programs are called assemblers.

![Assembly Phase Diagram](image)

Figure 2.2.5.1.1 Assembly Phase.

The assembler like most other forms of language translators/processors produces an executable version of the source program by associating the mnemonic labels and operands with machine addresses, replacing the mnemonic operation codes by their machine operation codes, and connecting the program to the operating system for servicing the pseudo operations and other request.

The assembly phase is usually accomplished in two passes over the source program. The assembler cannot associate a symbolic address with its machine address or
evaluate a symbolic expression until it has seen the
definition for each symbol and in an assembly language
program the definition of a symbol does not have to precede
its use.

The major task of the first pass is to associate the
mnemonic labels with machine addresses so that the
mnemonic operands and expressions can be associated with
their machine addresses in the second pass. To do this,
a symbol table is constructed as follows:

A location counter (LC) is set to base address (the
memory location at which the first instruction of the
object program will be loaded. This address could be
altered by relocatable loading*, initially. Then all the
input lines (source programs) are read, one at a time.
As the assembly proceeds the location counter is updated,
i.e. incremented by an appropriate amount, which depends

*In most operating systems, the task of loading a machine
language program into the memory of the machine is accom-
plished by a program called loader. A relocatable loader
performs the following tasks, in addition to the basic
loading operation:

(1) the object program can be displaced during load-
ing and the modification of the addresses of
the instructions are done by the loader,
(2) more than one object program can be loaded in
the memory and memory conflict is prevented by
the loader, and
(3) library programs may be automatically obtained
and placed in the user's memory area by this
kind of loader.
on the particular memory requirement of the statement just processed. The mnemonic labels are placed in the symbol table along with their machine addresses (indicated by the location counter). If it is desired to check for doubly defined symbols, some means of table lookup procedure must be employed to determine if the symbol to be placed in the symbol table is already there. In general the major tasks of pass one are accomplished using an algorithm similar to the one in Figure 2.2.5.1.2.
Exit from pass one

Read one statement of the source program

End of the source program

true

false

Exit from pass one

the symbol is doubly defined - set the appropriate flag on

the symbol is not doubly defined - location field

blank

non-blank

look up through SYMTAB to see if the mnemonic label exists

SYMTAB is the symbol table

found

not found

place the mnemonic symbol and value of location counter into SYMTAB - update SYMTAB length

update location counter by the appropriate amount based on the operation code

Figure 2.2.5.1.2 Pass One Flowchart Outline.
OPTAB: A table which contains the mnemonic operation codes, their corresponding machine operation code, type, and length.

Figure 2.2.5.1.3 Data Flow for Pass One.

Figure 2.2.5.1.4 Data Flow for Pass Two.
set the location counter to the desired base, to the location at which the first object program instruction will be loaded

read one statement of the source program

exit from pass 2

true

end of source program?

false

mnemonic operation code

pseudo operation

not pseudo operation

obtain the machine operation code; get the type of the instruction; evaluate the operand field; assemble the instruction

output the assembled instruction onto the object program file, or load it directly into memory for immediate run

output the source listing and the assembled instructions or the syntactical error messages

update the location counter by the appropriate amount based upon the operation code

Figure 2.2.5.1.5 Pass Two Flowchart Outline.
In order for pass two to assemble machine instructions, the machine operation code for each instruction has to be determined in addition to the machine address of its operand (obtainable from SYMTAB of pass one). The machine operation codes can be obtained from OPTAB and the access to the type of each operation code allows the proper evaluation of the expressions in the operand field and determination of the machine instruction fields into which the resultant value is to be placed.

Figure 2.2.5.1.5 illustrates the general procedure followed by pass two of most assemblers.

The preliminary step toward the development of completely symbolic machine language programming was the work done at MIT on "WhirlWind" in 1952, and by Rochester. Since that day a large number of assembly languages have been developed.

The design of assembly languages was an accomplishment at that time, since it circumvented the disadvantages (1) and (2) but difficulties in documentation and thus in maintenance and extension due to illegibility of the source program (to a large extent), and difficulties with the transferability due to the dependence of symbolic machine language programs on the details of a machine for which it was originally developed remained to be con-
sidered. Also, for solving problems which required a small number of runs, programming in symbolic machine languages did not seem effective due to the length of elapsed time from inception of the problem to its solution.

2.2.6 High-Level Languages

All the difficulties with assembly language programming lead to the development of high-level languages: the languages, usually subsets of the natural language English, chosen so that the programmer can express any effective algorithm. The programmer does not have to deal with the details of machine language programming or with the computer system on which the program is to run and can enjoy the following features inherent in most high-level languages:

1. ease of learning
2. ease of use
3. more structuring power in terms of problem solving, since the programmer does not have to express the algorithms in terms of a few basic machine instructions.
4. ease of debugging
5. ease of documentation, and
6. transferability (running a program developed on one machine or another dissimilar one).
The significance of some of the preceding advantages is demonstrated in the example in Figure 2.2.6.1, in which the same task as the machine and symbolic language programs in Figures 2.2.2.2 and 2.2.5.1 is accomplished.

A program in a high-level language is constructed by synthesizing a sequence of statements from the set of valid statements within a language. Different languages differ in syntax and semantics and are designed for different purposes.

\[
\begin{align*}
X &= \text{a non-zero positive integer} \\
Y &= \text{a non-zero positive integer} \\
R &= X^{**}Y
\end{align*}
\]

Figure 2.2.6.1 An Example of a High-Level Language Program.

One of the earliest works in language development is the 1952 description of the SHORT CODE\(^9\) for UNIVAC suggested by Dr. John Mauchly in 1949 and programmed by R. Logan, W. Schmitt, and A. Tonik. Dr. Mauchly's "suggestion was, in effect, to have a program which would accept algebraic equations as originally written..."\(^9\)

The basic principle was to use two-character codes to represent the variables and two-digit numbers to represent about 30 operations provided. For example the equation A=B+C can be written in SHORT CODE as:
The primitive systems, SHORT CODE AND SPEED CODING, were followed by others which answered a need at that time or have contributed to the design of programming languages.

The two years 1958 and 1959 are the most significant years during the period, in which the development of high-level languages has continued. To date, the most productive work is done using FORTRAN and COBOL, and ALGOL, the languages which were developed during those years, and the recent languages involve the direct and indirect influence of the work done in that period.  

2.2.6.1 Compiler

The programs written in high-level languages must undergo translation before they can be executed on computers. The translation is accomplished usually by compilers, the programs which transform programs written in a high-level language to their equivalent ones in low-level (assembly/machine) languages.

Figure 2.2.6.1.1 shows the major components of which a typical compiler is composed and the tasks carried out by it.
2.2.6.1.1 Scanner/Lexical Analyzer

The scanner scans the source program statements and builds (recognizes) the basic elements (tokens) which are identifiers, keywords, punctuations, operators, and other distinguishable symbols.
2.2.6.1.2 Parser

The parser recognizes the phrases (syntactic construction), the phase of translation known as syntax analysis, and interprets the meaning of constructions, the phase of translation known as semantic analysis.

2.2.6.1.3 Optimizer

The optimizer imposes certain efficiency criteria such as better use of temporary storage areas (the machine register) which can result in reduced number of loads and stores, the use of shorter and faster instructions, better management of storage allocation, and in general, taking advantage of machine dependent properties, reduce the memory space and the execution time of the object program. Some optimizers may even modify the source program (without disturbing the outcome of the program) in order to generate a more optimal version of the object program. For example changing "A = X**2" to "A = X*X," or "GO TO (100,100),N" to "IF (N-L)100,100,200," and

"DIMENSION A(100,100)

::

DO 10 I = 1,100

DO 10 J = 1,100
10 A(I,J) = 0.0" to
"DIMENSION A(100,100), SYSVAR* (10000)
EQUIVALENCE (A, SYSVAR)
.
.
DO 10 I = 1,10000
10 SYSVAR (I) = 0.0"

Such optimizations are to be the subject of future investigations in PEESHTAZ.

2.2.6.1.4 Code Generator

The code generator combines the information received from the parser and the optimization techniques to complete the translation process by generating assembly, machine or partially assembled instructions.

Compilation is much more involved than assembly and there are a number of references in which detailed description can be found.

*SYSVAR an identifier which does not appear in the user's program.
BIBLIOGRAPHY


3. \textsc{PEESHTAZ} the Interactive Computer Programming System in Farsi

3.1 Design Justification

Most programming systems have been designed to allow the programmers to express algorithms in concise languages which have been subsets of the English Language. Programming languages facilitate programming, should be machine independent, and provide means of describing algorithms so that information contained in them can be passed from one man to another and from one generation to the next one. To a great extent the growth of computer use and as a result, the computer industry, is dependent upon the means of man/machine communication. To help spread out the usage of the computer among Farsi speaking scientific, engineering, business and government organizations, \textsc{PEESHTAZ} THE INTERACTIVE PROGRAMMING SYSTEM -- IN FARSI is created.

The system uses the alphabet and grammar of Farsi and includes the useful features of already existent programming languages and operating systems with additional newborn *\textsc{PEESHTAZ} is a Farsi word which means: the pioneer, the leader, or the utmost extremity in front, the farthest of all.*
facilities for achievement of simplicity and generality in a programming system (without the deficiencies found in already existent and heavily used systems).

The following methods of creating a new system are all combined to design PEESHTAZ.

1. Cleaning up the features and adding new facilities to an existing language.
2. Combining the known useful features of several systems into one, and
3. Developing a system to meet the requirements of a certain problem area.

The programming language within PEESHTAZ is developed to create a language that is machine independent (assuring transferability), relatively easy to learn and use (assuring popularity), structurally powerful in terms of problem solving (assuring effectiveness), readable in the sense that a reasonably good understanding of the program can be obtained by reading a listing of the source program (assuring the ease of documentation, and as the result, providing ease with which the programs can be extended, modified, and maintained). Its interactive nature allows the possibility of a close man/machine interaction--the programmer can benefit from the advanced computer technology of today (a feature not found in the
languages in which most productive work is done*).

The system is designed for the use of Farsi speaking people so that all Iranian people (including high school and even elementary school students) have potentially the same access to computers that English speaking people now have.

Of course, the system meets design specifications, makes use of the knowledge and technology of today. Its goal is to make FORTRAN, COBOL, BASIC, and a number of other programming systems of scholarly interest only among Farsi-speaking people.

PEESHTAZ is composed of three major subsystems:

(1) The software simulated input/output device,

(2) ** text editor/string processing BYGONE language, and

(3) *** general purpose/interactive DARRIE Computer programming language.

*This is due to the large investment that English speaking societies have made in the earlier days (when FORTRAN and COBOL were designed). Farsi speaking communities should not have to pay for such economic pressure.

** is a Farsi word which means: the person in charge of the management (collection, manipulation and maintenance) of files.

*** is a Farsi word which refers to the ancient (old) Iranian language spoken in Darius I (the great king of Achaemenian Dynasty) period - the period in which Persians made the greatest empire of the ancient world.
Figure 3.1.1 The Major Subsystems/User Network.
3.2 The Software Simulated Input/Output Device of PEESHTAZ

There are many I/O devices such as teletypes, line printers, and CRT's which provide man/machine transmission of characters. These devices use the English characters, numerals, and a set of special symbols as their standard character set and input/output characters from left to right (the normal means of reading/writing in Latin oriented natural languages).

The software simulated input/output device program package (SSIODPP) provides facilities for transmission of and the full ASCII character set on a typical computer graphic terminal: TEKTRONIX T4002. The techniques used in PSSIODPP suggest the preliminary aspects of the design of Farsi hardware character generators* and Farsi character set pattern recognition systems.

*Model 38 5-level communication terminal, made by Midgulf Corporation (West Babylon, N.Y.) can only input/output a subset of character set and cannot transmit ASCII character set.
3.2.1 Decomposition of Farsi Characters Into Sub-Patterns

In order to reduce the cost of hardware realization of Farsi character set and/or to reduce the memory and programming time requirements for the software simulation of Farsi character generators the following procedure is created:

(1) a minimal set Figure 3.2.1.1.2 is driven from the detailed Farsi character set; Figure 3.2.1.1.1 the minimal set is composed of as few symbols as possible while maintaining readability,

(2) the characters with certain similarities are classified into clusters, Figures 3.2.1.1.3 and 3.2.1.1.4

(3) the common subpatterns among the members of each cluster are identified, Figure 3.2.1.1.3

(4) each character is identified by the subpatterns common among the members of the cluster to which it belongs, and then the Figure 3.2.1.3.1 through 3.2.1.3.1.17

(5) automatic means of combining the appropriate subpatterns to form the entire character set is designed. Figure 3.2.3
3.2.1.1 Reduction of Farsi Character Set

Figure 3.2.1.1.2 shows a reduced set of Farsi characters composed of 32 letters with a few letters having up to four variations depending upon their positions of appearance (at the beginning of a word and attached at the left end, in the middle of a word and attached at both ends, at the end and not attached, and at the end and attached from the right end) in words. The minimal set is driven from the detailed Farsi character set in Figure 3.2.1.1.1 and is designed to generate Farsi manuscript efficiently - using as few symbols as possible while maintaining readability.

Figure 3.2.1.1.1 The Detailed Set of Farsi Characters - 133 Symbols.
3.2.1.2 Classification of Farsi Characters

Certain similarities among the members of the minimal set allow further simplification in generation of the characters. Figure 3.2.1.1.3 illustrates the clusters of the characters with common subpatterns.
3.2.1.3.1 Decomposition Expressions of Farsi Characters

Let $S_m^n$ represent the primitive subpattern $m$ or the number $m$ of the cluster $n$ and let $S_m^n := S_{m_1}^{n_1} \cdot S_{m_2}^{n_2} \cdot \ldots \cdot S_{m_k}^{n_k}$.

$S_{m_3}^{n_3} \cdot \ldots \cdot S_{m_k}^{n_k}$ mean that the subpattern or member $S_m^n$ is composed of primitive subpatterns or members $S_{m_1}^{n_1}$, $S_{m_2}^{n_2}$, $S_{m_3}^{n_3}$, ..., and $S_{m_k}^{n_k}$. Where the $n_1$ and $n_j$ may or may not be the same, but $m_i$ and $m_j$ (for $i \neq j$) must be distinct.
3.2.1.3.1.1 Cluster 1 Decomposition

Cluster 1

member 1 2 3 4 5 6 7 8 9 10 11

class 1 2 3 4 5 6 7 8 9 10 11

Let $S_1^{p_1} ::= \cdots$, $S_1^{p_2} ::= \cdots$, $S_1^{p_3} ::= \cdots$, $S_1^{p_4} ::= \cdots$, $S_1^{p_5} ::= \cdots$,

$s_1^{p_6} ::= \cdots$, $s_1^{p_7} ::= \cdots$, $s_1^{p_8} ::= \cdots$, and $p_1^{p_9}$ be the set of sub-patterns, then

$s_1^{p_5} ::= s_1^{p_3} \cdot s_1^{p_1}$

$s_1^{p_9} ::= s_1^{p_4} \cdot s_1^{p_5}$

$s_1^{p_6} ::= s_1^{p_5} \cdot s_1^{p_9}$

$s_1^{p_1} ::= s_1^{p_5} \cdot s_1^{p_2}$

$s_1^{p_2} ::= s_1^{p_6} \cdot s_1^{p_2}$

$s_1^{p_10} ::= s_1^{p_6} \cdot s_1^{p_1}$

$s_1^{p_7} ::= s_1^{p_7} \cdot s_1^{p_10}$

$s_1^{p_3} ::= s_1^{p_7} \cdot s_1^{p_2}$
\[ S_8^1 ::= S_{p8}^1 . S_7^1 \]
\[ S_4^1 ::= S_8^1 . S_{p2}^1 \]
\[ S_{11}^1 ::= S_{p9}^1 . S_{p1}^1 \]

3.2.1.3.1.2 Cluster 2 Decomposition

Cluster 2

<table>
<thead>
<tr>
<th>Member</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>(\alpha)</td>
<td>(\beta)</td>
<td>(\gamma)</td>
<td>(\delta)</td>
<td>(\epsilon)</td>
<td>(\zeta)</td>
<td>(\eta)</td>
<td>(\theta)</td>
</tr>
</tbody>
</table>

Let \( S_{p1}^2 ::= \), \( S_{p2}^2 ::= \), \( S_{p3}^2 ::= \) and \( S_{p1}^1 \) \((i=3,4,5,6,7,8)\)

be the set of subpatterns,

then:

\[ S_{p1}^7 ::= S_{p1}^2 . S_{p2}^2 \]
\[ S_{p1}^3 ::= S_{p1}^2 . S_{p3}^2 \]
\[ S_{p1}^5 ::= S_{p1}^1 . S_{p7}^2 \]
\[ S_{p1}^4 ::= S_{p6}^1 . S_{p1}^2 . S_{p3}^2 \]
\[ S_{p3}^5 ::= S_{p3}^1 . S_{s3}^2 \]
\[ S_{p1}^1 ::= S_{p3}^1 . S_{s3}^2 \]
\[ S_2^2 := S_4^1 \cdot S_5^1 \cdot S_1^1 \cdot S_6^2 \cdot S_2^2 \]
\[ S_2^1 := S_4^1 \cdot S_5^1 \cdot S_1^1 \]

3.2.1.3.1.3 Cluster 3 Decomposition

Cluster 3

member 1 2 3 4 5 6

Character

Let \( S_1^3 \) be the set of subpatterns, then:

\[ \cdot \cdot S_3^3 := S_1^3 \cdot S_3^3 \]
\[ \cdot \cdot S_4^3 := S_8^1 \cdot S_3^3 \]
\[ \cup S_1^3 := S_1^3 \cdot S_4^3 \]
\[ \cup S_2^3 := S_8^1 \cdot S_1^3 \]
\[ \cdot S_5^3 := S_1^3 \cdot S_2^3 \]
\[ \cdot S_6^3 := S_6^1 \cdot S_3^1 \cdot S_2^3 \]
3.2.1.3.1.4 Cluster 4 Decomposition

Cluster 4

<table>
<thead>
<tr>
<th>member</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Let \( S_4 \) be the set of subpatterns, then:

\[
S_4 := S_{p_1}^{4} \cdot S_{p_2}^{4} \cdot S_{p_6}^{1} \cdot S_{p_7}^{1} \cdot S_{p_8}^{1} \cdot S_{p_3}^{1}
\]

3.2.1.3.1.5 Cluster 5 Decomposition

Cluster 5

<table>
<thead>
<tr>
<th>member</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Let \( S_5 \) be the set of subpatterns, then:

\[
S_5 := S_{p_1}^{5} \cdot S_{p_8}^{1} \cdot S_{p_3}^{2}
\]
\[ S_3^5 ::= S_{p_1}^5 \]
\[ S_1^5 ::= S_{p_2}^5 \cdot S_{p_3}^2 \]
\[ S_4^5 ::= S_{p_8}^1 \cdot S_3^5 \]
\[ S_2^5 ::= S_{p_4}^5 \cdot S_{p_3}^2 \]

3.2.1.3.1.6 Cluster 6 Decomposition

Cluster 6

<table>
<thead>
<tr>
<th>member</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>( \sigma \cdot \mu \cdot \nu \cdot \tau \cdot \xi \cdot \zeta )</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Let \( S_{p_1}^6 ::= \sigma \), \( S_{p_2}^6 ::= \tau \), \( S_{p_3}^2 \), and \( S_{p_8}^1 \) be the set of subpatterns,

then:

\[ S_3^6 ::= S_{p_1}^6 \]
\[ S_4^6 ::= S_{p_8}^1 \cdot S_3^5 \]
\[ S_2^6 ::= S_{p_4}^5 \cdot S_{p_3}^2 \]
\[ S_1^6 ::= S_3^5 \cdot S_{p_3}^2 \]
3.2.1.3.1.7 Cluster 7 Decomposition

Cluster 7

<table>
<thead>
<tr>
<th>member</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

Let $S^7_1 := \overline{3}$, $S^7_2 := \overline{1}$, and $S^7_3 := \overline{4}$ be the set of subpatterns,

then:

$\overline{3} S^7_3 := S^7_1$

$\overline{1} S^7_1 := S^7_3 . S^1_2$

$\overline{4} S^7_4 := S^7_2 . S^3_3$

$\overline{1} S^7_2 := S^7_4 . S^1_2$

3.2.1.3.1.8 Cluster 8 Decomposition

Cluster 8

<table>
<thead>
<tr>
<th>member</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Let $S^8_1 := \overline{1}$ and $S^8_2 := \overline{2}$ be the set of subpatterns,

then:

$\overline{1} S^8_2 := S^8_1$

$\overline{2} S^8_1 := S^8_2 . S^8_2$
3.2.1.3.1.9 Cluster 9 Decomposition

Cluster 9
member 1 2 3
close 1 1 1

Let $S_9^{p_1} := 1$, $S_9^{p_2} := \cup$, and $S_9^{p_3} \vdash$ be the set of subpatterns,
then:

$S_9 := S_9^{p_1}$

$\cup S_1 := S_9^{p_1} \cup S_9^{p_2}$

$\vdash S_2 := S_9^{p_1} \vdash S_9^{p_3}$

3.2.1.3.1.10 Cluster 10 Decomposition

Cluster 10
member 1 2 3
character 1 1 1

Let $S_{10}^{p_1} := \rho$, $S_{10}^{p_6} := S_{10}^{p_7}$, and $S_{10}^{p_8}$ be the set of sub-patterns,
then:
3.2.1.3.1.11 Cluster 11 Decomposition

Cluster 11
member 1 2
character > >

Let \( S_{11}^{10} := \) and \( S_{11}^{10} \) be the set of subpatterns, then:

\[ S_{11}^{10} := S_{p_1}^{10} \]

\[ S_{p_1}^{10} := S_{p_7}^{10} \cdot S_{p_1}^{10} \]

\[ S_{p_1}^{10} := S_{p_6}^{10} \cdot S_{p_8}^{10} \cdot S_{p_2}^{10} \]

3.2.1.3.1.12 Cluster 12 Decomposition

Cluster 12
member 1 2 3 4
character > > >

Let \( S_{12}^{11} := \), \( S_{12}^{11} := \), \( S_{12}^{11} := \), \( S_{p_6}^{11} := \), \( S_{p_2}^{11} := \), \( S_{p_2}^{11} := \), and \( S_{p_7}^{11} := \) be the set of subpatterns,
then:

\[ S_{12}^{12} ::= S_{p_6}^1 \cdot S_{p_1}^{12} \cdot S_{p_2}^{12} \]
\[ S_{12}^{12} ::= S_{p_7}^1 \cdot S_{p_2}^{12} \]
\[ S_{12}^{12} ::= S_{p_6}^1 \cdot S_{p_7}^1 \cdot S_{p_1}^{12} \cdot S_{p_3}^{12} \]

3.2.1.3.1.13 Cluster 13-19 Decomposition

Clusters 13 through 19 are singletons and thus are not decomposable into contributing subpatterns.

3.2.1.3.1.14 Cluster 20 Decomposition

Cluster 20

<table>
<thead>
<tr>
<th>member</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>character</td>
<td>( _ )</td>
<td>( _ )</td>
<td>( _ )</td>
<td>( _ )</td>
<td>( _ )</td>
<td>( _ )</td>
</tr>
</tbody>
</table>

Let \( S_{p_1}^{20} := \_ \), \( S_{p_2}^{20} := \_ \), \( S_{p_3}^{20} := \_ \), \( S_{p_4}^{20} := c \), \( S_{p_5}^{20} := \_ \), and \( S_{p_6}^{20} := \_ \) be the set of subpatterns,

then:

\[ S_{20}^{20} ::= S_{p_1}^{20} \cdot S_{p_2}^{20} \cdot S_{p_3}^{20} \cdot S_{p_4}^{20} \cdot S_{p_5}^{20} \cdot S_{p_6}^{20} \]
1 S_{20}^{20} := S_{p_1}^{20}

\hat{1} S_{20}^{20} := S_{p_1}^{20} \cdot S_{p_2}^{20}

\hat{\hat{1}} S_{3}^{20} := S_{p_5}^{20} \cdot S_{p_1}^{20}

\hat{\hat{\hat{1}}} S_{4}^{20} := S_{p_6}^{20} \cdot S_{p_1}^{20}

\ddagger S_{5}^{20} := S_{p_1}^{20} \cdot S_{p_3}^{20}

\ddagger \ddagger S_{6}^{20} := S_{1}^{20} \cdot S_{p_4}^{20}

3.2.1.3.1.15 Cluster 21 Decomposition

Cluster 21

member 1 2

character \Delta \Lambda

Let S_{p_1}^{21} := \Lambda and S_{p_2}^{21} := \omega be the set of subpatterns,

then:

\Lambda S_{2}^{21} := S_{p_1}^{21}

\Delta S_{1}^{21} := S_{p_1}^{21} \cdot S_{p_2}^{21}

3.2.1.3.1.16 Cluster 22 Decomposition

Cluster 22 is a singleton.
3.2.1.3.2 The Advantages of the Proposed Decomposition Technique

The proposed decomposition technique has the following obvious advantages:

(1) the reduction of the amount of storage required for the character matrices, in the software simulation of Farsi characters,
(2) the reduction of the cost of design and hardware realization of Farsi character generators, and
(3) the simplification of the process of maintenance (extension, modification, and trouble shooting) of the software simulated input/output device and/or the hardware.

If the characters are to be represented in a 10 x 16 matrix, to generate the members of Cluster 1, individually, without the use of proposed technique, the coordinates of 64 points must reside within the memory of the machine, because:
However using the proposed technique only 17 coordinates would suffice to generate all the members of Cluster 1, because:
which results in over 73% improvement, still some other clusters use the primitive subpatterns of this cluster. Of course the proposed method imposes the necessity for some means of concatenating the appropriate primitive sub-patterns together which require invocation of a set of subroutines.

If a cost is associated with each coordinate to be realized hardware wise, then the advantage of the proposed
method is obvious, as far as designing the hardware character generators are concerned.
### Letters

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Symbol</td>
</tr>
<tr>
<td>Arabic</td>
<td></td>
</tr>
<tr>
<td>Hebrew</td>
<td></td>
</tr>
<tr>
<td>Greek</td>
<td></td>
</tr>
<tr>
<td>Cyrillic</td>
<td></td>
</tr>
<tr>
<td>Armenian</td>
<td></td>
</tr>
<tr>
<td>Kazakh</td>
<td></td>
</tr>
<tr>
<td>Mongolian</td>
<td></td>
</tr>
<tr>
<td>Thai</td>
<td></td>
</tr>
<tr>
<td>Khmer</td>
<td></td>
</tr>
<tr>
<td>Tibetan</td>
<td></td>
</tr>
<tr>
<td>Vietnamese</td>
<td></td>
</tr>
<tr>
<td>Lao</td>
<td></td>
</tr>
<tr>
<td>Khmer</td>
<td></td>
</tr>
<tr>
<td>Tibetan</td>
<td></td>
</tr>
<tr>
<td>Vietnamese</td>
<td></td>
</tr>
<tr>
<td>Lao</td>
<td></td>
</tr>
</tbody>
</table>

### Digits

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

### Special/Control Characters

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control mode</td>
<td></td>
</tr>
<tr>
<td>Text Editor</td>
<td></td>
</tr>
<tr>
<td>OS-3</td>
<td>OS-3</td>
</tr>
<tr>
<td>ASCII</td>
<td>ASCII</td>
</tr>
<tr>
<td>SCAPE</td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td></td>
</tr>
<tr>
<td>SPACE</td>
<td></td>
</tr>
<tr>
<td>Horizontal Pointer (HPTR)</td>
<td></td>
</tr>
<tr>
<td>Vertical Pointer</td>
<td></td>
</tr>
<tr>
<td>Whole Text Mode (WTXT)</td>
<td></td>
</tr>
<tr>
<td>Line of Text Mode (LTXT)</td>
<td></td>
</tr>
<tr>
<td>HPTR and WTXT</td>
<td></td>
</tr>
<tr>
<td>HPTR and LTXT</td>
<td></td>
</tr>
<tr>
<td>Character Deleter</td>
<td></td>
</tr>
<tr>
<td>Line Deleter</td>
<td></td>
</tr>
<tr>
<td>Origin</td>
<td></td>
</tr>
<tr>
<td>Horizontal Tab</td>
<td></td>
</tr>
<tr>
<td>Vertical Tab</td>
<td></td>
</tr>
<tr>
<td>Forward Space</td>
<td></td>
</tr>
<tr>
<td>Backward Space</td>
<td></td>
</tr>
<tr>
<td>Search End of File Forward</td>
<td></td>
</tr>
<tr>
<td>Search End of File Backward</td>
<td></td>
</tr>
<tr>
<td>Rewind</td>
<td></td>
</tr>
<tr>
<td>Set at End of Data</td>
<td></td>
</tr>
<tr>
<td>Decimal Point</td>
<td>/</td>
</tr>
<tr>
<td>Double Quote</td>
<td>&quot;</td>
</tr>
<tr>
<td>Exclamation Mark</td>
<td>!</td>
</tr>
<tr>
<td>Question Mark</td>
<td>?</td>
</tr>
<tr>
<td>Colon</td>
<td>:</td>
</tr>
<tr>
<td>Percent</td>
<td>%</td>
</tr>
<tr>
<td>Right Parenthesis</td>
<td>)</td>
</tr>
<tr>
<td>Left Parenthesis</td>
<td>(</td>
</tr>
<tr>
<td>Right Bracket</td>
<td>]</td>
</tr>
<tr>
<td>Left Bracket</td>
<td>[</td>
</tr>
<tr>
<td>Right Brace</td>
<td></td>
</tr>
<tr>
<td>Left Brace</td>
<td></td>
</tr>
<tr>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td></td>
</tr>
<tr>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>Exclusive OR</td>
<td>#</td>
</tr>
<tr>
<td>Equal Sign</td>
<td>-</td>
</tr>
<tr>
<td>Not Equal Sign</td>
<td>#</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
</tr>
<tr>
<td>Addition Operator</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction Operator</td>
<td>-</td>
</tr>
<tr>
<td>Multiplication Operator</td>
<td>*</td>
</tr>
<tr>
<td>Division Operator</td>
<td>/</td>
</tr>
<tr>
<td>Exponentiation Operator</td>
<td>^</td>
</tr>
<tr>
<td>Comma</td>
<td>,</td>
</tr>
</tbody>
</table>

**Figure 3.2.1.3.2.1**  The Character Set of PEESHTAE
3.2.2 The Internal Representation of PEESHTAZ

Character Set

The alphabet of PEESHTAZ is represented by 9-bit codes with lower 8 bits (bits 0-7) containing the information part and the bit 8 used as the parity bit. The codes are selected in such a way that there is no conflict with the ASCII representation. The additional one bit is required to allow room for the inclusion of 32 letters, 10 numerals, and 50 special/control characters of PEESHTAZ. A combination of the ASCII and PEESHTAZ character set representation and also the collating sequence of PEESHTAZ characters are represented in Figure 3.2.2.1.
<table>
<thead>
<tr>
<th>Decimal</th>
<th>0000</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NUL</td>
<td>DLE</td>
<td>SP</td>
<td>@</td>
<td>P</td>
<td>p</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>s</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>u</td>
<td>e</td>
<td>u</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACK</td>
<td>SYN</td>
<td>$</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BEL</td>
<td>ETB</td>
<td>?</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>VT</td>
<td>FSC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>\</td>
<td>N</td>
<td>n</td>
<td>~</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>0</td>
<td>O</td>
<td>-</td>
<td>0</td>
<td>DEL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2.2.1 The ASCII Character Set Internal Codes.
Figure 3.2.2.2 ASCII and Character Set Arrangement on TEXTRONIX T-1002.
Figure 3.2.2.3 Terminal Keyboard.
3.2.3 Character Transmission Algorithm

The transmission of PEESHTAZ/ASCII characters is simulated using both the alphabetic and the vector plotting modes of TEKTRONIX T4002, as illustrated in Figure 3.10.

![Diagram of Character Transmission Algorithm]

Figure 3.2.3.1 Character Transmission Algorithm.
The above algorithm indicates that there are two modes of character transmission within the terminal: ASCI mode and - mode. The terminal is placed in ASCI mode by a press on ASCI (a key of the terminal keyboard) after which ASCI characters are transmitted until ASCII is pressed on again. Thus a press on ASCI converts the terminal into a regular Tektronix T4002 until it is pressed again which converts the terminal into that of PEESHTAZ.

3.2.4 Software Simulated Input/Output Device

Components

All the components of the software simulated input/output device are organized into software simulated input/output device program package (SSIODPP).

SSIDPP is composed of the following subsystems:
(1) Software simulated character generators,
(2) input/output driver (IODRV), and
(3) a set of auxiliary subsystems.

3.2.4.1 Software Simulated Character Generators

The set of character generator routines (Exhibit 3.2.4.2.3 provide IODRV with the necessary information about character matrices. All of the character
generator routines are written and documented by character Generator Program Writer (CGPW), Exhibit 3.2.4.2.1.

3.2.4.2 Character Generator Program Writer (CGPW)

CGPW automates the implementation process of all the software-simulated character generators.

It contains the appropriate information in the decomposition expressions and the coordinates of the vectors of which the primitive subpatterns are composed, to write and document the required sub-programs.

CGPW follows the general algorithm illustrated in Figure 3.2.4.2.1. The source program, the input, and the output (the software simulated character generators) are given in Exhibits 3.2.4.2.1, 3.2.4.2.2, and 3.2.4.2.3, respectively.
produce a special listing of all or those desired

prepare two segments of storage (PRMTVN and PRMTVP) to hold the character identifications (the cluster and member numbers), and the coordinates of the vectors of which the character is composed (for the documentation)

get a decomposition expression

is there a special listing of the routines generated required? Yes

end of data?

Yes

end of data?

No

write the program identification and its documentation part

generate the remaining portion of the routine: the appropriate calls to the routine that moves the beam

write the program identification, its documentation, and the part that makes appropriate calls to the routines that generate primitive subpatterns

get the information in regard to the movement of the beam on the screen of the terminal (to draw the character)

get the coordinates of the vectors of which the character is composed

place the coordinates of the vectors into the appropriate section of PRMTVP and mark those cells of the character matrix which correspond to the coordinates of the vectors

set those cells of the character matrix which correspond to the coordinates of the vectors of which the primitive is composed, and place the coordinates of the vectors into the appropriate section of PRMTVP

ERROR in the decomposition expression?

No

Abnormal stop

Yes

Figure 3.2.4.2.1 The General Algorithm of CGPW.
Exhibit 3.2.4.2.2 Character Generator Program Writer/

The Source Program

```plaintext
PROGRAM CG3

INTEGER 
  N,
  K,
  M,
  L,
  M,
  N,
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
  L,
  INTEGER
  K,
  M,
```
Exhibit 3.2.4.2.3 Decomposition Expressions/the Input to CGPW
Exhibit 3.2.4.2.4 Software Simulated Character Generators/

The Output of CGPW

SUBROUTINE S01P1

\*
\* THIS ROUTINE GENERATES MEMBER \( P1 \) OF CLUSTER \( Q1 \) OF THE \*
\* ALPHABET. \*
\*
\*------------------------------------------------------------------------*
\*| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
\*------------------------------------------------------------------------*
\*| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
\*------------------------------------------------------------------------*
\*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \* |
\*------------------------------------------------------------------------*
\*| 0123456789 |
\*------------------------------------------------------------------------*
\*
\*------------------------------------------------------------------------*
\* CALL \( \text{VDPV} \) \( 5, 9, 1 \) \*
\* CALL \( \text{VDPV} \) \( 9, 7, 1 \) \*
\* CALL \( \text{VDPV} \) \( 6, 7, 1 \) \*
\* RETURN \*
\* END \*
\*------------------------------------------------------------------------*

SUBROUTINE S01P2

\*
\* THIS ROUTINE GENERATES MEMBER \( P2 \) OF CLUSTER \( Q2 \) OF THE \*
\* ALPHABET. \*
\*
\*------------------------------------------------------------------------*
\*| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
\*------------------------------------------------------------------------*
\*| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
\*------------------------------------------------------------------------*
\*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \*| \* |
\*------------------------------------------------------------------------*
\*| 0123456789 |
\*------------------------------------------------------------------------*
\*
\*------------------------------------------------------------------------*
\* CALL \( \text{VDPV} \) \( 6, 7, 0 \) \*
\* CALL \( \text{VDPV} \) \( 8, 7, 1 \) \*
\* CALL \( \text{VDPV} \) \( 8, 6, 1 \) \*
\* CALL \( \text{VDPV} \) \( 1, 9, 1 \) \*
\* RETURN \*
\* END \*
\*------------------------------------------------------------------------*
SUBROUTINE S0103

THIS ROUTINE GENERATES MEMBER (23) OF CLUSTER (21) OF THE
ALPHABET.

CALL VODRV (A, 5, 0)
CALL VODRV (A, 5, 1)
RETURN
END

SUBROUTINE S0104

THIS ROUTINE GENERATES MEMBER (24) OF CLUSTER (21) OF THE
ALPHABET.

CALL VODRV (A, 5, 0)
CALL VODRV (A, 5, 1)
RETURN
END
**SUBROUTINE S104**

This routine generates member (15) of cluster (01) of the alphabet.

```
CALL VOORV (7, 4, 1)
RETURN
END
```

**SUBROUTINE S01P6**

This routine generates member (06) of cluster (11) of the alphabet.

```
CALL VOORV (4, 13, 0)
CALL VOORV (9, 13, 1)
RETURN
END
```
SUBROUTINE S01R7

* THIS ROUTINE GENERATES MEMBER (P7) OF CLUSTER (011) OF THE ALPHABET.

CALL LDRV (6,13,2)
CALL VDRV (6,13,1)
RETURN
END

SUBROUTINE S01R9

* THIS ROUTINE GENERATES MEMBER (P9) OF CLUSTER (011) OF THE ALPHABET.

CALL LDRV (7,14,2)
CALL VDRV (7,14,1)
RETURN
END
**SUBROUTINE SCIPE**

*THIS ROUTINE GENERATES ELEMENTS OF A THOSE OF THE ALPHA.*

```
CALL VOPPY (7, 11, 1)
CALL VOPPY (7, 11, 2)
CALL VOPPY (7, 11, 3)
CALL VOPPY (9, 11, 4)
CALL VOPPY (9, 9, 1)
CALL VOPPY (9, 9, 2)
END
```
SUBROUTINE SC103

**THIS ROUTINE GENERATES MEMBER LIST OF CLUSTER C11 OF THE ALPHABET.**

CALL SC1D3
CALL SC1D1
RETURN
END
**SUBROUTINE S6179**

This routine generates member (09) of cluster (01) of the alphabet.

CALL SIG8
CALL SIG9
RETURN
END

**SUBROUTINE S6106**

This routine generates member (16) of cluster (01) of the alphabet.

CALL SIG8
CALL SIG9
RETURN
END
SUBROUTINE SC181

* THIS ROUTINE GENERATES MEMBER (111) OF CLUSTER (111) OF THE ALPHABET.

CALL SC166
CALL SP122
RETURN
END

SUBROUTINE SC171

* THIS ROUTINE GENERATES MEMBER (111) OF CLUSTER (111) OF THE ALPHABET.

CALL SC166
CALL SP122
RETURN
END
SUBROUTINE SU11C

THIS ROUTINE GENERATES MEMBER (161) OF CLUSTER (111) OF THE ALPHABET.

CALL SU1IP
CALL SIIP1
RETURN
END

SUBROUTINE SU117

THIS ROUTINE GENERATES MEMBER (187) OF CLUSTER (111) OF THE ALPHABET.

CALL SI1IP7
CALL 50117
RETURN
END
SUBROUTINE S316?

THIS ROUTINE GENERATES MEMBER 131 OF CLUSTER 011 OF THE ALPHABET.

CALL S3167
CALL S01P2
RETURN

CALL S01P2
RETURN

SUBROUTINE S3108

THIS ROUTINE GENERATES MEMBER 131 OF CLUSTER 011 OF THE ALPHABET.

CALL S3108
CALL S01C7
RETURN
END
**SUBROUTINE S3114**

---

*This routine generates member (14) of cluster (Q1) of the alphabet.*

---

```
CALL S7129
CALL S7122
RETURN
END
```

---

**SUBROUTINE S3111**

---

*This routine generates member (11) of cluster (Q1) of the alphabet.*

---

```
CALL S0110
CALL S31101
RETURN
END
```
SUBROUTINE S02P1

THIS ROUTINE GENERATES MEMBER (P1) OF CLUSTER (Q2) OF THE ALPHABET.

CALL VPORV (3, 8, 0)
CALL VPORV (6, 10, 1)
CALL VPORV (9, 7, 1)
CALL VPORV (4, 7, 1)
RETURN
END

SUBROUTINE S02P2

THIS ROUTINE GENERATES MEMBER (P2) OF CLUSTER (Q2) OF THE ALPHABET.

CALL VPORV (4, 7, 0)
CALL VPORV (3, 7, 1)
RETURN
END
SUBROUTINE S02P3

* THIS ROUTINE GENERATES MEMBER (P3) OF CLUSTER (02) OF THE ALPHABET.

CALL VDRV (3, 7, 1)
CALL VDRV (2, 2, 1)
CALL VDRV (1, 1, 1)
CALL VDRV (9, 4, 1)
CALL VDRV (1, 7, 1)
RETURN
END

SUBROUTINE S02P7

* THIS ROUTINE GENERATES MEMBER (P7) OF CLUSTER (02) OF THE ALPHABET.

CALL S02P1
CALL S32P2
RETURN
END
SUBROUTINE SC217

This routine generates member (17) of cluster (07) of the alphabet.

CALL SC217
CALL SC207
RETURN
END

SUBROUTINE SC279

This routine generates member (19) of cluster (02) of the alphabet.

CALL SC179
CALL SC207
RETURN
END
SUBROUTINE SC21

THESE ROUTINE GENERATES MEMBER (141) OF CLUSTER (221) OF THE
ALPHABET.

CALL SC1PR
CALL SC2PR
RETURN
END

SUBROUTINE SC23

THESE ROUTINE GENERATES MEMBER (151) OF CLUSTER (231) OF THE
ALPHABET.

CALL SC1PR
CALL SC2PR
RETURN
END
SUBROUTINE 021

*************************************************************************
* THIS ROUTINE GENERATES MEMBER (21) OF CLUSTER (021) OF THE
* ALPHABET.
*************************************************************************

CALL S0103
CALL S0104
CALL S0105
CALL S0106
CALL S1202
RETURN
END

SUBROUTINE 020

*************************************************************************
* THIS ROUTINE GENERATES MEMBER (00) OF CLUSTER (020) OF THE
* ALPHABET.
*************************************************************************

CALL S0103
CALL S0104
CALL S0105
CALL S0106
CALL S1202
RETURN
END
SUBROUTINE SG222

This routine generates member (32) of cluster (32) of the alphabet.

CALL SG184
CALL SG185
CALL SG201
END

SUBROUTINE SG251

This routine generates member (31) of cluster (31) of the alphabet.

CALL VIP4 (5, 7, 5)
CALL VIP4 (5, 4, 1)
CALL VIP4 (5, 6, 1)
CALL VIP4 (5, 3, 1)
CALL VIP4 (5, 4, 1)
CALL VIP4 (5, 7, 1)
RETURN
END
SUBROUTINE S0302

**THIS ROUTINE GENERATES MEMBER #21 OF CLUSTER #31 OF THE ALPHABET.**

CALL VPRV (5, 14, 9)
CALL VPRV (6, 7, 1)
CALL VPRV (4, 7, 1)
RETURN
END

SUBROUTINE S0307

**THIS ROUTINE GENERATES MEMBER #71 OF CLUSTER #331 OF THE ALPHABET.**

CALL VPRV (5, 7, 2)
CALL VPRV (6, 7, 1)
CALL VPRV (3, 7, 1)
CALL VPRV (2, 7, 1)
RETURN
END
SUBROUTINE SC304

THIS ROUTINE GENERATES MEMBER (94) OF CLUSTER (33) OF THE ALPHABET.

CALL VDORV ( 5, 7, 8)
CALL VDORV ( 6, 6, 1)
CALL VDORV ( 7, 3, 1)
CALL VDORV ( 2, 3, 1)
CALL VDORV ( 1, 4, 1)
CALL VDORV ( 6, 7, 1)
RETURN
END

SUBROUTINE SD303

THIS ROUTINE GENERATES MEMBER (131) OF CLUSTER (63) OF THE ALPHABET.

CALL VDORV ( 6, 1)
CALL VDORV ( 7, 1)
RETURN
END
SUBROUTINE 50304

This routine generates member (34) of cluster (33) of the alphabet.

CALL 50101
CALL 50303
RETURN
END

SUBROUTINE 50301

This routine generates member (31) of cluster (33) of the alphabet.

CALL 50301
CALL 50304
RETURN
END
SUBROUTINE 50302

**THIS ROUTINE GENERATES MEMBER (12) OF CLUSTER (63) OF THE ALPHABET.**

CALL S01P3
CALL S03P1
RETURN
END

SUBROUTINE 50305

**THIS ROUTINE GENERATES MEMBER (151) OF CLUSTER (63) OF THE ALPHABET.**

CALL S01P5
CALL S03P2
RETURN
END
SUBROUTINE SU3'ROL

This routine generates member (1) of cluster (2) of the alphabet.

CALL 531P6
CALL 532P1
CALL 532P2
RETURN

END

SUBROUTINE SU4'P1

This routine generates member (01) of cluster (24) of the alphabet.

CALL VDPY (5, 9, 1)
CALL VDPY (5, 7, 1)
CALL VDPY (7, 9, 1)
CALL VDPY (7, 5, 1)
CALL VDPY (5, 7, 1)
CALL VDPY (5, 9, 1)
RETURN

END
SUBROUTINE SC482


CALL VOTRY (5, 8, 6)
CALL VOTRY (4, 7, 1)
CALL VOTRY (3, 7, 1)
END

SUBROUTINE SC483

* THIS ROUTINE GENERATES MEMBER [43] OF CLUSTER [23] OF THE ALPHABET.

CALL SC481
CALL SC482
END
SUBROUTINE S045

THIS ROUTINE GENERATES MEMBER (14) OF CLUSTER (14) OF THE ALPHABET.

CALL S0156
CALL S0157
CALL S0158
CALL S3159
RETURN
END

SUBROUTINE S0411

THIS ROUTINE GENERATES MEMBER (11) OF CLUSTER (14) OF THE ALPHABET.

CALL S0116
CALL S0117
CALL S0118
CALL S31119
RETURN
END
SUBROUTINE S3432

This routine generates member (32) of cluster (34) of the alphabet.

CALL SJ1P6
CALL SJ2P7
CALL SJ3P8
CALL SJ4P1
RETURN
END

SUBROUTINE S35P1

This routine generates member (31) of cluster (35) of the alphabet.

CALL VPDP2 (3, 0)
CALL VPDP5 (7, 10, 0)
CALL VPDP5 (9, 11, 1)
CALL VPDP5 (4, 9, 1)
CALL VPDP5 (1, 7, 3)
CALL VPDP5 (3, 7, 1)
RETURN
END
SUBROUTINE S5303

THIS ROUTINE GENERATES MEMBER (31) OF CLUSTER (05) OF THE
ALPHABET.

CALL S5513
RETURN
END

SUBROUTINE S5511

THIS ROUTINE GENERATES MEMBER (31) OF CLUSTER (05) OF THE
ALPHABET.

CALL S5503
CALL S5523
RETURN
END
SUBROUTINE 50502

* THIS ROUTINE GENERATES MEMBER (12) OF CLUSTER (05) OF THE
  ALPHABET.

CALL 50504
CALL 50503
RETURN
END

SUBROUTINE 50504

* THIS ROUTINE GENERATES MEMBER (14) OF CLUSTER (05) OF THE
  ALPHABET.

CALL 50504
CALL 50503
RETURN
END

CALL 50504
CALL 50503
RETURN
END
SUBROUTINE SG=P1

This routine generates member (P1) of cluster (051) of the alphabet.

CALL VP9QV (5, 7, 0)
CALL VPORV (7, 7, 1)
CALL VPOPV (4, 11, 1)
CALL VPDRV (5, 12, 1)
CALL VPORV (9, 11, 1)
CALL VPOPV (8, 10, 1)
CALL VPOPV (5, 7, 1)
CALL VPOPV (3, 7, 1)
RETURN
END

SUBROUTINE SG=P2

This routine generates member (P2) of cluster (061) of the alphabet.

CALL SO501
RETURN
END
SUBROUTINE SDH14

This routine generates member (14) of cluster (54) of the alphabet.

CALL SD1P9
CALL SD6C3
RETURN
END

SUBROUTINE SD6E1

This routine generates member (21) of cluster (64) of the alphabet.

CALL SD6E3
CALL SD6D3
RETURN
END
Subroutine S602

This routine generates member 121 of Cluster 161 of the alphabet.

CALL S3604
CALL S1203
RETURN
END

Subroutine S601

This routine generates member 141 of Cluster 107 of the alphabet.

CALL VOPRV (5, 13, 7)
CALL VOPRV (5, 12, 7)
CALL VOPRV (5, 7, 7)
RETURN
END
SUBROUTINE S3799

This routine generates member 1921 of cluster (271) of the alphabet.

CALL V909V (9,17, 1)
CALL V909V (1,11, 1)
END

SUBROUTINE S6733

This routine generates member 1331 of cluster (971) of the alphabet.

CALL S6733
RETURN
END
SUBROUTINE S3711

This routine generates member (31) of cluster (37) of the alphabet.

CALL S3713
CALL S3712
RETURN
END

SUBROUTINE S6704

This routine generates member (34) of cluster (37) of the alphabet.

CALL S6703
CALL S6702
RETURN
END
SUBROUTINE 53712

THIS ROUTINE GENERATES MEMBER (12) OF CLUSTER (37) OF THE
SLP1-M13FT.

CALL 53714
CALL 53102
RETURN
END

SUBROUTINE 50142

THIS ROUTINE GENERATES MEMBER (12) OF CLUSTER (37) OF THE
SLP1-M13FT.

CALL 50142
CALL 50202
RETURN
END
SUBROUTINE SC#02

* THIS ROUTINE GENERATES MEMBER (P2) OF CLUSTER (O2) OF THE
  ALPHABET.

CALL VORV (4, 7, 1)
CALL VORV (1, 5, 1)
CALL VORV (3, 5, 1)
RETURN
END

SUBROUTINE SC#01

* THIS ROUTINE GENERATES MEMBER (P1) OF CLUSTER (O1) OF THE
  ALPHABET.

CALL SCOPI
RETURN
END
SUBROUTINE S301

* THIS ROUTINE GENERATES MEMBER (LL) OF CLUSTER (09) OF THE
* ALPHABET.

CALL S0302
CALL S0302
RETURN
END

SUBROUTINE S3091

* THIS ROUTINE GENERATES MEMBER (PL) OF CLUSTER (39) OF THE
* ALPHABET.

CALL V009 (3, 15, 0)
CALL V009 (3, 16, 1)
RETURN
END
SUBROUTINE 5092

**This routine generates member (02) of cluster (193) of the alphabet.**

CALL VPROV (9, 7, 0)
CALL VPROV (9, 6, 1)
CALL VPROV (7, 6, 1)
CALL VPROV (3, 6, 1)
RETURN
END

SUBROUTINE 5093

**This routine generates member (03) of cluster (193) of the alphabet.**

CALL VPROV (9, 7, 0)
CALL VPROV (7, 7, 1)
RETURN
END
SUBROUTINE S0904

**THIS ROUTINE GENERATES MEMBER 341 OF CLUSTER 099 OF THE ALPHABET.**

CALL S09P1
RETURN
END

SUBROUTINE S0901

**THIS ROUTINE GENERATES MEMBER 1011 OF CLUSTER 1091 OF THE ALPHABET.**

CALL S09P1
RETURN
END
SUBROUTINE SG202

CALL SGPM1
CALL SGPM3
RETURN
END

SUBROUTINE S10P1

CALL UOO1 ( 9, 3, 1)
CALL UOO1 ( 9, 7, 1)
CALL UOO1 ( 4, 3, 1)
CALL UOO1 ( 4, 7, 1)
RETURN
END
SUBROUTINE S101

* THIS ROUTINE GENERATES MEMBER (01) OF CLUSTER (101) OF THE
  ALPHABET.

CALL S1P1
RETURN
END

SUBROUTINE S102

* THIS ROUTINE GENERATES MEMBER (32) OF CLUSTER (11) OF THE
  ALPHABET.

CALL S1P7
CALL S1P1
RETURN
END
SUBROUTINE S1213

THIS ROUTINE GENERATES MEMBER (3) OF CLUSTER (11) OF THE
ALPHABET.

CALL S1210
CALL S1212
RETURN
END

SUBROUTINE S1111

THIS ROUTINE GENERATES MEMBER (P1) OF CLUSTER (11) OF THE
ALPHABET.

CALL V301 (5, 10, 2)
CALL V301 (5, 9, 1)
CALL V301 (5, 8, 1)
CALL V301 (5, 7, 1)
CALL V301 (4, 7, 1)
RETURN
END
SUBROUTINE $3111$

THIS ROUTINE GENERATES TEMP (II) OF CLUSTER (II) OF THE ALPHABET.

CALL $3111$
RETURN
END

SUBROUTINE $3112$

THIS ROUTINE GENERATES TEMP (II) OF CLUSTER (II) OF THE ALPHABET.

CALL $3112$
CALL $3111$
RETURN
END
SUBROUTINE S1291

\[\text{THIS ROUTINE GENERATES MEMBER (01) OF CLUSTER (12) OF THE}\]
\[\text{ALPHABET.}\]

CALL VOOPV ( 9, 7, 7)
CALL VPORV ( 7, 7, 1)
CALL VPORV ( 5, 7, 1)
CALL VPORV ( 7, 9, 1)
CALL VPORV ( 8, 9, 1)
CALL VPORV ( 5, 8, 1)
RETURN
END

SUBROUTINE S1292

\[\text{THIS ROUTINE GENERATES MEMBER (P2) OF CLUSTER (12) OF THE}\]
\[\text{ALPHABET.}\]

CALL VOOPV ( 7, 7, 0)
CALL VPORV ( 4, 7, 1)
RETURN
END
SUBROUTINE S12P3

* THIS ROUTINE GENERATES MEMBER (03) OF CLUSTER (12) OF THE ALPHABET.

CALL VPOPV ( 4, 7, 0)
CALL VPOPV ( 4, 3, 1)
CALL VPOPV ( 4, 5, 1)
CALL VPOPV ( 2, 2, 1)
CALL VPOPV ( 1, 1, 1)
RETURN
END

SUBROUTINE S12P2

* THIS ROUTINE GENERATES MEMBER (02) OF CLUSTER (12) OF THE ALPHABET.

CALL E12P1
CALL E12P2
RETURN
END
SUBROUTINE S1231

* THIS ROUTINE GENERATES MEMBER 111 OF CLUSTER 111 OF THE
ALPHABET.

CALL S1202
CALL S0102
RETURN
END

SUBROUTINE S1274

* THIS ROUTINE GENERATES MEMBER 141 OF CLUSTER 121 OF THE
ALPHABET.

CALL S1207
CALL S1202
RETURN
END
SUBROUTINE S1203


CALL S1206
CALL S1207
CALL S1201
CALL S1203
RETURN
END

SUBROUTINE S1201


CALL VPORV (9, 7, 0)
CALL VPORV (7, 9, 1)
CALL VPORV (6, 7, 1)
CALL VPORV (6, 8, 1)
CALL VPORV (5, 9, 1)
CALL VPORV (5, 7, 1)
CALL VPORV (3, 9, 1)
CALL VPORV (2, 7, 1)
CALL VPORV (1, 9, 1)
RETURN
END
**SUBROUTINE S1121**

This routine generates member (111) of cluster (114) of the
alphabet.

```
CALL V004V ( 3, 6, 1)
CALL V004V ( 7, 6, 1)
CALL V004V ( 7, 5, 1)
CALL V004V ( 6, 5, 1)
CALL V004V ( 7, 9, 1)
CALL V004V ( 5, 9, 1)
CALL V004V ( 6, 7, 1)
CALL V004V ( 4, 7, 1)
RETURN
END
```

**SUBROUTINE S1251**

This routine generates member (21) of cluster (15) of the
alphabet.

```
CALL V004V ( 7, 16, 1)
CALL V004V ( 4, 4, 1)
CALL V004V ( 3, 7, 1)
CALL V004V ( 8, 7, 1)
CALL V004V ( 4, 5, 1)
CALL V004V ( 1, 5, 1)
CALL V004V ( 6, 3, 1)
CALL V004V ( 10, 1, 1)
RETURN
END
```
SUBROUTINE 51701

**THIS ROUTINE GENERATES MEMBER (31) OF CLUSTER (17) OF THE ALPHABET.**

```
CALL VOFPV ( 6, 15, 9)
CALL VOFPV ( 9, 7, 11)
CALL VOFPV ( 7, 6, 11)
CALL VOFPV ( 5, 7, 11)
CALL VOFPV ( 7, 10, 11)
CALL VPFPV ( 4, 10, 11)
RETURN
END
```

SUBROUTINE 51901

**THIS ROUTINE GENERATES MEMBER (11) OF CLUSTER (19) OF THE ALPHABET.**

```
CALL SIPO
CALL SIPO
RETURN
END
```
SUBROUTINE STPLC

THIS ROUTINE GENERATES MEMBER (01) OF CLUSTER (01) OF THE
ALPHABET.

CALL VPRV ( 0, 7, 1)
CALL VPRV ( 0, 6, 1)
CALL VPRV ( 0, 6, 1)
CALL VPRV ( 0, 7, 1)
CALL VPRV ( 7, 6, 1)
CALL VPRV ( 0, 7, 1)
CALL VPRV ( 0, 7, 1)
RETURN
END

SUBROUTINE S2UP1

THIS ROUTINE GENERATES MEMBER (01) OF CLUSTER (01) OF THE
ALPHABET.

CALL VPRV ( 4, 14, 1)
CALL VPRV ( 4, 5, 1)
RETURN
END
SUBROUTINE 52092

* This routine generates member (22) of cluster (22) of the alphabet.

CALL V50RV ( 9, 14, 7)
CALL V50RV ( 9, 15, 7)
CALL V50RV ( 7, 14, 7)
CALL V50RV ( 7, 15, 7)
RETURN
END

SUBROUTINE 52093

* This routine generates member (23) of cluster (22) of the alphabet.

CALL V50RV ( 4, 14, 3)
CALL V50RV ( 0, 14, 3)
RETURN
END
**SUBROUTINE S2094**

This subroutine generates member (24) of cluster (20) of the alphabet.

CALL VDFR (4, 14, 3)
CALL VDFR (4, 13, 1)
CALL VDFR (3, 13, 1)
CALL VDFR (2, 10, 1)
CALL VDFR (1, 10, 1)
CALL VDFR (0, 9, 1)
RETURN
END

**SUBROUTINE S2095**

This subroutine generates member (25) of cluster (20) of the alphabet.

CALL VDFR (6, 14, 0)
CALL VDFR (6, 13, 1)
CALL VDFR (5, 13, 1)
CALL VDFR (4, 13, 1)
CALL VDFR (3, 11, 1)
RETURN
END
**SUBROUTINE S2006**

This routine generates member (06) of cluster (2C) of the alphabet.

```
CALL VDREV (9,16,11)
CALL VDREV (7,16,11)
CALL VDREV (9,12,1)
CALL VDREV (6,12,1)
CALL VDREV (4,12,1)
RETURN
END
```

**SUBROUTINE S2001**

This routine generates member (01) of cluster (20) of the alphabet.

```
CALL S20P1
RETURN
END
```
SUBROUTINE S2002

* THIS ROUTINE GENERATES MEMBER (102) OF CLUSTER (21) OF THE
  ALPHABET.

CALL S20P1
CALL S20P2
RETURN
END

SUBROUTINE S2003

* THIS ROUTINE GENERATES MEMBER (103) OF CLUSTER (26) OF THE
  ALPHABET.

CALL S20P5
CALL S20P1
RETURN
END
**SUBROUTINE 52004**

**THIS ROUTINE GENERATES MEMBER (24) OF CLUSTER (201) OF THE ALPHABET.**

CALL S20P6
CALL S20P1
RETURN
END

**SUBROUTINE 52005**

**THIS ROUTINE GENERATES MEMBER (75) OF CLUSTER (201) OF THE ALPHABET.**

CALL S20P1
CALL S20P3
RETURN
END
**SUBROUTINE S219A**

This routine generates member (21) of cluster (21) of the alphabet.

```
CALL S2191
CALL S2194
RETURN
END
```

**SUBROUTINE S2191**

This routine generates member (21) of cluster (21) of the alphabet.

```
CALL VPORV ( 5, 4, 6)
CALL VPORV ( 5, 14, 6)
CALL VPORV ( 5, 4, 6)
RETURN
END
```
SUBROUTINE S2192

THIS ROUTINE GENERATES CLUSTER (92) OF CLUSTER (211) OF THE
ALPHABET.

CALL VPDEV ( 8, 6, 9)
CALL VPDEV ( 7, 6, 1)
CALL VPDEV ( 6, 5, 1)
CALL VPDEV ( 5, 4, 1)
CALL VPDEV ( 4, 5, 0)
CALL VPDEV ( 3, 6, 1)
CALL VPDEV ( 2, 6, 1)
RETURN
END

SUBROUTINE S2192

THIS ROUTINE GENERATES CLUSTER (92) OF CLUSTER (211) OF THE
ALPHABET.

CALL S21P1
RETURN
END
SUBROUTINE S211

THIS ROUTINE GENERATES MEMBER (11) OF CLUSTER (21) OF THE
ALPHABET.

CALL S21M1
CALL S21M3
RETURN
END

SUBROUTINE S221

THIS ROUTINE GENERATES MEMBER (11) OF CLUSTER (22) OF THE
ALPHABET.

CALL V21LY ( 2.14, 2)
CALL V21LY ( 5.14, 2)
CALL V21LY ( 8.14, 2)
RETURN
END
SUBROUTINE DASH

* THIS ROUTINE GENERATES THE SPECIAL CHARACTER (DASH).

CALL VDRV ( 9, 5, 0)
CALL VDRV (7, 5, 1)
CALL VDRV (1, 7, 1)
RETURN
END

SUBROUTINE SPACE

* THIS ROUTINE GENERATES THE SPECIAL CHARACTER (SPACE).

CALL VDRV ( 7, 7, 0)
RETURN
END
SUBROUTINE PERFACE

THIS ROUTINE GENERATES THE SPECIAL CHARACTERS (PERFACE).

CALL VDORV (4, 3, 1)
CALL VDORV (5, 4, 1)
CALL VDORV (6, 5, 1)
CALL VDORV (7, 6, 1)
CALL VDORV (8, 7, 1)
CALL VDORV (9, 8, 1)
CALL VDORV (10, 9, 1)
RETURN
END

SUBROUTINE SLASH

THIS ROUTINE GENERATES THE SPECIAL CHARACTERS (SLASH).

CALL VDORV (4, 7, 1)
CALL VDORV (1, 8, 1)
RETURN
END
SUBROUTINE OR

**THIS ROUTINE GENERATES THE SPECIAL CHARACTER: [CP 1].**

CALL VPRV (7, 11, 0)
CALL VPRV (7, 4, 1)
CALL VPRV (6, 3, 1)
CALL VPRV (4, 3, 1)
CALL VPRV (3, 4, 1)
CALL VPRV (3, 11, 1)
CALL VPRV (1, 7, 1)
RETURN
END

SUBROUTINE AND

**THIS ROUTINE GENERATES THE SPECIAL CHARACTER: (AND 1).**

CALL VPRV (7, 11, 0)
CALL VPRV (7, 10, 1)
CALL VPRV (6, 11, 1)
CALL VPRV (4, 11, 1)
CALL VPRV (3, 11, 1)
CALL VPRV (3, 2, 1)
CALL VPRV (1, 7, 1)
RETURN
END
SUBROUTINE LTXT

This routine generates the special characters (LTXT).

CALL VPDRV (6, 3, 0)
CALL VPDRV (7, 3, 1)
CALL VPDRV (4, 4, 1)
CALL VPDRV (3, 0, 1)
CALL VPDRV (7, 11, 1)
CALL VPDRV (5, 11, 1)
CALL VPDRV (3, 11, 1)
CALL VPDRV (1, 10, 1)
CALL VPDRV (2, 3, 1)
CALL VPDRV (1, 7, 1)
END

SUBROUTINE 3KSP

This routine generates the special characters (3KSP).

CALL VPDRV (3, 5, 1)
CALL VPDRV (7, 5, 1)
CALL VPDRV (6, 4, 1)
CALL VPDRV (5, 4, 1)
CALL VPDRV (1, 9, 1)
CALL VPDRV (2, 9, 1)
CALL VPDRV (6, 7, 1)
CALL VPDRV (1, 7, 1)
END
SUBROUTINE LIKRT
C
C ***
C THIS ROUTINE GENERATES THE SPECIAL CHARACTER: (L<R<
C
CALL VPORV ( 8, 3, 11)
CALL VPORV ( 6, 3, 11)
CALL VPORV ( 8, 11, 11)
CALL VPORV ( 4, 7, 11)
RETURN
END

SUBROUTINE LPAREN
C
C ***
C THIS ROUTINE GENERATES THE SPECIAL CHARACTER: (L<>
C
CALL VPORV ( 7, 3, 11)
CALL VPORV ( 5, 3, 11)
CALL VPORV ( 5, 10, 11)
CALL VPORV ( 7, 11, 11)
CALL VPORV ( 2, 7, 11)
RETURN
END
**SUBROUTINE MINUS**

This routine generates the special character (MINUS 1).

CALL VPORV (1, 7, 0)
CALL VPORV (7, 7, 1)
CALL VPORV (1, 7, 0)
RETURN
END

**SUBROUTINE NE**

This routine generates the special character (NE).

CALL VPORV (9, 6, 0)
CALL VPORV (5, 6, 1)
CALL VPORV (9, 6, 1)
CALL VPORV (7, 10, 2)
CALL VPORV (2, 5, 1)
CALL VPORV (1, 7, 0)
RETURN
END
SUBROUTINE NOT

* THIS ROUTINE GENERATES THE SPECIAL CHARACTER (NOT 1).

CALL VPOP1VEV (7, 5, 1)
CALL VPOP1VEV (7, 5, 1)
CALL VPOP1VEV (7, 5, 1)
RETURN
END

SUBROUTINE PLUS

* THIS ROUTINE GENERATES THE SPECIAL CHARACTER (PLUS 1).

CALL VPOP1VEV (7, 7, 1)
CALL VPOP1VEV (7, 7, 1)
CALL VPOP1VEV (7, 7, 1)
RETURN
END
SUBROUTINE PRINT

THIS ROUTINE GENERATES THE SPECIAL CHARACTERS (PRINT).

CALL VP0V ( 7, 11, 0)
CALL VP0V ( 3, 3, 11)
CALL VP0V ( 7, 3, 11)
CALL VP0V ( 0, 3, 11)
CALL VP0V ( 5, 4, 11)
CALL VP0V ( 6, 4, 11)
CALL VP0V ( 3, 11, 11)
CALL VP0V ( 4, 11, 11)
CALL VP0V ( 5, 11, 11)
CALL VP0V ( 4, 11, 11)
CALL VP0V ( 7, 11, 11)
RETURN
END

SUBROUTINE P3XKT

THIS ROUTINE GENERATES THE SPECIAL CHARACTERS (P3XKT).

CALL VP0V ( 6, 7, 0)
CALL VP0V ( 8, 3, 11)
CALL VP0V ( 4, 11, 11)
CALL VP0V ( 6, 11, 11)
CALL VP0V ( 4, 7, 11)
RETURN
END
SUBROUTINE RAPEN

* THIS ROUTINE GENERATES THE SPECIAL CHARACTER (RAPEN).

CALL VP3PV (5, 3, 1)
CALL VP3PV (5, 3, 1)
CALL VP1PV (5, 3, 1)
CALL V01PV (7, 4, 1)
CALL VP9R4 (7, 1, 1)
CALL VP7R4 (5, 1, 1)
CALL VP1R4 (5, 1, 1)
CALL VPOP1 (3, 7, 1)
RETURN
END

SUBROUTINE COLON

* THIS ROUTINE GENERATES THE SPECIAL CHARACTER (COLON).

CALL VP3PV (5, 3, 1)
CALL VP3PV (5, 3, 1)
CALL VP1PV (5, 3, 1)
CALL V01PV (7, 4, 1)
CALL VP9R4 (7, 1, 1)
CALL VP7R4 (5, 1, 1)
CALL VP1R4 (5, 1, 1)
CALL VP7R4 (5, 1, 1)
CALL VPOP1 (3, 7, 1)
RETURN
END
SUBROUTINE DIV

This routine generates the special character (DIV).

CALL VOPEV (7, 7, 7)
CALL VOPEV (7, 7, 7)
CALL VOPEV (6, 6, 6)
CALL VOPEV (6, 6, 6)
CALL VOPEV (5, 5, 5)
CALL VOPEV (5, 5, 5)
CALL VOPEV (4, 4, 4)
CALL VOPEV (4, 4, 4)
CALL VOPEV (4, 4, 4)
CALL VOPEV (4, 4, 4)
RETURN
END

SUBROUTINE ED

This routine generates the special character (ED).

CALL VOPEV (9, 5, 9)
CALL VOPEV (9, 5, 9)
CALL VOPEV (9, 5, 9)
CALL VOPEV (9, 5, 9)
RETURN
END
SUBROUTINE EXCL

This subroutine generates the special characters (EXCL).

CALL VPOV (5, 6, 7)
CALL VPOV (7, 11, 11)
CALL VPOV (5, 11, 11)
CALL VPOV (7, 5, 11)
CALL VPOV (6, 7, 11)
CALL VPOV (5, 7, 11)
CALL VPORV (3, 7, 3)
RETURN
END

SUBROUTINE EXPO

This subroutine generates the special characters (EXPO).

CALL VPOV (5, 7, 3)
CALL VPOV (6, 9, 1)
CALL VPOV (7, 9, 1)
CALL VPOV (5, 6, 1)
CALL VPOV (7, 5, 1)
CALL VPOV (6, 7, 1)
CALL VPOV (3, 7, 3)
RETURN
END
SUBROUTINE SEFF

THIS ROUTINE GENERATES THE SPECIAL CHARACTER: * (SEFF).

CALL VPOPV (7, 4, 1)
CALL VPOPV (5, 4, 1)
CALL VPOPV (7, 3, 1)
CALL VPOPV (3, 3, 1)
CALL VPOPV (7, 1, 1)
CALL VPOPV (1, 1, 1)
RETURN
END

SUBROUTINE FWSP

THIS ROUTINE GENERATES THE SPECIAL CHARACTER: * (FWSP).

CALL VPOPV (7, 5, 1)
CALL VPOPV (5, 5, 1)
CALL VPOPV (7, 3, 1)
CALL VPOPV (3, 3, 1)
CALL VPOPV (7, 1, 1)
CALL VPOPV (1, 1, 1)
RETURN
END
Subroutine SEF3

This routine generates the special character (SEF3).

CALL VOPARV (7, 4, 1)
CALL VOPARV (7, 4, 1)
CALL VOPARV (7, 4, 1)
CALL VOPARV (7, 4, 1)
CALL VOPARV (7, 4, 1)
CALL VOPARV (7, 4, 1)
CALL VOPARV (7, 4, 1)
RETURN

END

Subroutine WETO

This routine generates the special character (WETO).

CALL VOPARV (7, 6, 1)
CALL VOPARV (7, 6, 1)
CALL VOPARV (7, 6, 1)
CALL VOPARV (7, 6, 1)
CALL VOPARV (7, 6, 1)
CALL VOPARV (7, 6, 1)
RETURN

END
THIS ROUTINE GENERATES THE SPECIAL CHARACTER(s) 1.

CALL VP1RV (7, 7, 1)
CALL P0PV (4, 7, 1)
CALL Y970V (4, 11)
CALL VPORV (2, 7, 1)
CALL VP09V (4, 1, 1)
CALL YPORV (5, 12, 1)
CALL Y990V (5, 10, 1)
CALL )/PRV (5, 1, 1)
CALL YDCRV (5, 10, 1)
CALL )/PRV (1, 7, 1)

RETURN
**SUBROUTINE VTA3**

This routine generates the special character (VTAR).

```
CALL VORV ( Q, 7, C)
CALL VROPV ( 7, 7, 1)
CALL VRQPV ( 7, A, 1)
CALL VPQPV ( 5, 7, 1)
CALL VORPV ( 7, 7, 1)
CALL VORPV ( 7, 7, 1)
RETURN
```

**SUBROUTINE WTXT**

This routine generates the special character (WXTX).

```
CALL VORV ( 8, 7, 1)
CALL VORV ( 8, 7, 1)
CALL VORV ( 8, 5, 1)
CALL VORV ( 8, 7, 1)
CALL VORV ( 8, 7, 1)
CALL VORV ( 8, 7, 1)
CALL VORV ( 8, 7, 1)
CALL VORV ( 8, 7, 1)
RETURN
```


SUBROUTINE EYPC

THIS ROUTINE GENERATES THE SPECIAL CHARACTER "(EYPC)."

CALL V03PV (7, 8, 7)
CALL V03PV (7, 9, 7)
CALL V03PV (8, 9, 7)
CALL V03PV (8, 8, 7)
CALL V03PV (7, 6, 7)
CALL V03PV (7, 7, 7)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 4, 1)
CALL V03PV (7, 4, 1)
CALL V03PV (7, 4, 1)
CALL V03PV (7, 4, 1)
CALL V03PV (7, 4, 1)
RETURN
END

SUBROUTINE EULT

THIS ROUTINE GENERATES THE SPECIAL CHARACTER "(EULT)."

CALL V03PV (5, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
CALL V03PV (7, 7, 1)
RETURN
END
SUBROUTINE CDMA

This routine generates the special characters (CDMA).

CALL VDPHY (7, 7, 1)
CALL VDPHY (7, 7, 1)
CALL VDPHY (7, 7, 1)
CALL VDPHY (7, 7, 1)
CALL VDPHY (7, 7, 1)
CALL VDPHY (7, 7, 1)
RETURN
END

SUBROUTINE GT

This routine generates the special characters (GT).

CALL VDPHY (4, 4, 1)
CALL VDPHY (7, 7, 1)
CALL VDPHY (4, 4, 1)
CALL VDPHY (2, 7, 1)
RETURN
END
THIS ROUTINE GENERATES THE SPECIAL CHARACTER "CHR.",

CALL V007 (5, 5, 1)
CALL V070 (6, 4, 1)
CALL V070 (6, 3, 1)
CALL V070 (5, 4, 1)
CALL V070 (5, 3, 1)
CALL W007 (6, 7, 1)
CALL W070 (6, 6, 1)
CALL W070 (5, 7, 1)
CALL W070 (5, 6, 1)
CALL W070 (4, 7, 1)
CALL W070 (4, 6, 1)
CALL W070 (3, 7, 1)
CALL W070 (3, 6, 1)
RETURN
END

---

SUBROUTINE AST

THIS ROUTINE GENERATES THE SPECIAL CHARACTER "AST",

CALL V007 (5, 7, 1)
CALL V070 (5, 7, 1)
CALL W007 (5, 7, 1)
CALL W070 (5, 7, 1)
RETURN
END
SUBROUTINE LT

THIS ROUTINE GENERATES THE SPECIAL CHARACTER (LT 1.

CALL VODAY (7, 4, 7)
CALL VODAY (7, 7, 7)
CALL VODAY (7, 10, 7)
CALL VODAY (7, 7, 7)
CALL VODAY (6, 7, 7)
CALL VODAY (7, 7, 7)
CALL VODAY (7, 7, 7)
CALL VODAY (7, 7, 7)
END

SUBROUTINE LTPACF

THIS ROUTINE GENERATES THE SPECIAL CHARACTER (LTACF).

CALL VODAY (7, 3, 7)
CALL VODAY (3, 3, 7)
CALL VODAY (5, 2, 7)
CALL VODAY (7, 7, 7)
CALL VODAY (3, 4, 7)
CALL VODAY (5, 10, 7)
CALL VODAY (7, 11, 7)
CALL VODAY (7, 7, 7)
CALL VODAY (7, 7, 7)
END
SUBROUTINE SFO

- This routine generates the special character: (0271).

CALL V056V (6, 8, 1)
CALL V056V (5, 8, 1)
CALL V056V (5, 7, 1)
CALL V056V (4, 7, 1)
CALL V056V (3, 7, 1)
CALL V056V (3, 6, 1)
CALL V056V (3, 5, 1)
CALL V056V (3, 4, 1)
CALL V056V (3, 3, 1)
CALL V056V (4, 3, 1)
CALL V056V (5, 3, 1)
CALL V056V (6, 3, 1)
RETURN

SUBROUTINE PELRR

- This routine generates the special character: (0274).

CALL V056V (6, 8, 1)
CALL V056V (5, 8, 1)
CALL V056V (5, 7, 1)
CALL V056V (4, 7, 1)
CALL V056V (3, 7, 1)
CALL V056V (3, 6, 1)
CALL V056V (3, 5, 1)
CALL V056V (3, 4, 1)
CALL V056V (3, 3, 1)
CALL V056V (4, 3, 1)
CALL V056V (5, 3, 1)
CALL V056V (6, 3, 1)
RETURN
3.2.4.3 Input/Output Driver

The input/output driver (IODRV) monitors the interaction of the necessary segments of PSSIODPP to accomplish the input and output processes. The major tasks of IODRV are:

1. to receive character codes from the terminal,
2. to generate \[\text{PEESHTAZ}\] characters by employing the appropriate character generator routine,
3. to transmit character codes,
4. to control the movement of the cursor: position it appropriately before and after generation of a character, and keep it within the boundaries of the screen, and
5. to keep the terminal in the requested mode: \[\text{PEESHTAZ}\] or ASCII.

The input/output driver follows the general algorithm illustrated in Figure 3.2.4.3.1. The source program and sample outputs of IODRV are presented in Exhibits 3.2.4.3.1 and 3.2.4.3.2, respectively.
Figure 3.2.4.3.1 The General Algorithm of IODRV.
CALL TC4AR3 (GS1)

CALL scHaRa (74IC41THIGH0A)

CALL 5C4A91

CALL SCHAR0 13L068
حریف مارسی اندازه (۳)

اعداد مارسی اندازه (۳)

حرف مخصوص بیشتر اندازه (۳)
بنابر گزارش روزنامه‌های زلین، ایران ضروری است تولید برق خود را از طریق راه‌های انرژی اتمی ۵ هزار صد ناسال ۹۹ افزایش دهد.

کمک به برق، اب تمرکز ۴ هوايی پاک، وسایل نقلیه عمومی و مواد علمی و اموری بشر، همراه دم ایران را ببخش در اورده.
3.3 the Interactive Text Editor/String Processing
BYGONE
Language of PEESHTAZ

BYGONE is an interactive text editor/string processing language which uses the alphabet and the grammar of Farsi language.

It is designed for creation, modification, and display of files under PEESHTAZ. It is to be used conversationally from a remote terminal so the user may monitor processing one command at a time. Nevertheless, it can be operated on nonconversational basis by submitting tasks.

The files to be manipulated are placed in the memory of PEESHTAZ before the start of an editing session. The manipulation of text within memory increases the throughput of the system, as a result of less input/output activities to access the files contents. Also, the files being modified remain protected against the changes made until explicitly specified (eliminating the need of creation of local/intermediate files). However, the amount of text to be edited at one time is limited to the size of the available memory. This difficulty can be resolved by dividing the file into several smaller (than the size of available memory) segments and editing each segment individually.
3.3.1 Functional Description

is composed of a set of command verbs which perform editing functions on the contents of the file residing in the memory.

Each verb carries an array of parameters, some being optional. This feature widens the scope and increases the versatility of the system. The command statements (the verbs presented with their appropriate parameters) can specify the text editing to be performed in different modes of operation.

<table>
<thead>
<tr>
<th>Mode Designator</th>
<th>Text Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>one line, several consecutive lines or several non-consecutive lines, specified by the line numbers (,), or appropriate pointers (↔)</td>
</tr>
<tr>
<td>←</td>
<td>one character or several consecutive characters of one line, one character or several consecutive characters of several consecutive lines, or one character or several consecutive characters of several nonconsecutive lines</td>
</tr>
<tr>
<td>[ ]</td>
<td>the entire text</td>
</tr>
<tr>
<td>[ ] ↓</td>
<td>one character or several consecutive characters of the entire text</td>
</tr>
</tbody>
</table>

Figure 3.3.1.1 The Modes of Operation and Maniputable Text Components.
3.3.2 Notation used in Command Statements

**Underlined.** The underlined words are required when the command statement in which they appear is used.

**[⋯]** Brackets. The brackets enclose the optional portions of a command statement. All the information within the brackets may be omitted or included at the option of user. This notation is distinguished from the symbol [ ] used to indicate "the entire text" mode of operation.

**{⋯}** Braces. The braces enclose two or more vertically stacked options in a command statement, when only one of the enclosed options can be used, at one time.
Punctuation Symbols. Punctuation symbols, comma (‘)*, connective word (؟)*, quotation mark (") and parenthesis shown within command statements are required unless they are explicitly noted as optional.

The elements of PEESHTAZ character set.
3.3.3 Special Terms

**TEXT** is the textual data to be manipulated under the control of **BYGONE** and is composed of one or several **LINES**.

**LINE** is an element of **TEXT** and is composed of one or several **CHARACTERS**.

**CHARACTER** is one of the symbols available on **PEATS** keyboard. Figure 3.2.2.3

**LINE NUMBER** is a line number (a positive decimal integer) associated with each line of text which uniquely identifies the lines comprising the text.

**END OF LINE** is denoted by a carriage return (a code which may appear at the end of lines, only).

**END OF TEXT** is denoted by two carriage returns appearing right after each other.

<character>:: = one of the symbols of **PEATS** keyboard

<characters>:: = <character> | <characters>

<character>

<nonzero digit>:: = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digit>:: = 0 | <nonzero digit>

digits :: = <empty>|<digits><digit>

<line number>:: = <nonzero digit><digits>

<end of line>:: = a carriage return

<line>:: = <line number><characters>

<end of line>
<lines> ::= <line> | <line><lines>
<text> ::= <lines><end of line>

**POINTER** is the symbolic representation of the position of a line within the text (line number) or the position of a character within a line.

**LAST-LINE POINTER** the last line of the text is designated symbolically by [ and is automatically adjusted as new lines are added at the end of the text.
FIRST LINE POINTER. The first line pointer, designated symbolically by \( \uparrow \), points to the first line of text. The editor automatically adjusts the value of \( \uparrow \) as new lines are added at the beginning of the text.

INTER-TEXT (VERTICAL) POINTERS. Besides the first and last line pointers, \( \text{BYGONE} \) provides ten other line pointers that can point to any lines of the text. They are designated by \( \leftarrow, \rightarrow, \uparrow, \downarrow, \delta, \beta, \nu, \lambda, \gamma, \text{ and } \zeta \). If additional text is added in front or at the end of the text \( \leftarrow \text{ and } \rightarrow \) do not get adjusted automatically (as opposed to pointers \( \uparrow \text{ and } \downarrow \)).

CHARACTER (HORIZONTAL) POINTERS. In addition to the vertical pointers, there are two pointers which can point to a substring within the lines of text. They are designated symbolically as \( \downarrow \text{ and } \uparrow \downarrow \), these pointers point to the beginning and the end of a string of characters within the lines of text.

FILE NAME. File name is an identifier by which a saved disk file can be identified. The characters within a file name can be any of the \( \text{PEESHTAZ} \) characters except " and blank.

LOGICAL UNIT is an integer constant satisfying

\( \lessgtr \text{logical unit} \lessgtr 20 \text{ inequality which is associated to a saved file, a scratch file, or another hardware device (line printer, card punch, and plotter).} \)
EDITING CHARACTERS belong to the set of characters that terminal can generate. Each character invokes a procedure within to satisfy a specific request. The following editing characters are in addition to the mode and pointer designators.

Transfers the control to control mode.

transfers the control to OS-3 (the operating system of Oregon State University host computer).

places terminal in ASCII mode for the transmission of ASCII characters and places the terminal back in mode after the transmission of ASCII characters is completed (when it is sent again).

moves the cursor of the terminal one place to the left, erasing what was there (leaves a blank character).

carriage returns (positions the cursor at the beginning of the next line).

scapes, positions the cursor at the beginning (right-hand side) of the next line and warns that the preceding information on that line is to be disregarded.
deletes the preceding character, it cannot delete itself. For example, is the same as transmitting: deletes the preceding portion of a line of text. For Example, is the same as transmitting: erases the screen and positions the curser at the upper right-hand side (beginning of text) of the screen. refers to the horizontal pointer; it causes the cursor to move to the next previously specified position. refers to the vertical pointer; it causes the cursor to move to the next previously specified position. forward spaces one record on the logical unit number specified.
operates as \( \downarrow \) except it backspaces instead of forward space.

searches forward on the specified logical unit for a file mark.

operates as \( \downarrow \) except it searches backward on the specified logical unit for a file mark instead of searching forward.

rewinds the specified file (positions the read/write pointer at the beginning of the file).

sets the read/write pointer at the end of the file.
If no source is specified in a command statement, the information to be input is expected to enter from the terminal keyboard.

However, if no destination is specified in a command statement, the information to be output will be displayed at the screen of the terminal or no output is generated (depending on the command).

**Direct Addressing** is the method in which line numbers and/or pointer symbols (without any + or - signs following them), are used to directly identify the lines of the text.

Examples:

<table>
<thead>
<tr>
<th>Direct Addressing</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>2</td>
<td>r, n</td>
</tr>
<tr>
<td>3</td>
<td>3, n, n</td>
</tr>
<tr>
<td>4</td>
<td>n, n</td>
</tr>
<tr>
<td>5</td>
<td>n, n, n</td>
</tr>
<tr>
<td>6</td>
<td>n, n, n, n</td>
</tr>
<tr>
<td>7</td>
<td>n, n, n, n</td>
</tr>
</tbody>
</table>
Relative Addressing is the method in which the lines of text are identified by addresses such as c±n or c*n (relative line numbers or relative pointers); where n is a line number or a pointer number and c is a positive integer constant.

Examples:

\[<\text{string}_1>,<\text{string}_0>\]
Figure 3.3.3.1  Vertical and Horizontal Pointers Within BYGONE
3.3.4 The Externally-Apparent Structure of \( \overline{\text{BYGONE}} \)

Figure 3.3.4.1 The Environment of \( \overline{\text{BYGONE}} \)

3.3.5 The Externally-Apparent Structure of Text in \( \overline{\text{BYGONE}} \)

<table>
<thead>
<tr>
<th>horizontal pointer 2 (( \uparrow \uparrow ))</th>
<th>horizontal pointer 1 (( \uparrow \uparrow ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>end of lines</td>
<td>end of lines</td>
</tr>
</tbody>
</table>

- first line pointer (\( \square \))
- vertical pointer 1 (\( \leftarrow \uparrow \))
- vertical pointer 2 (\( \leftarrow \uparrow \))
- vertical pointer 10 (\( \leftarrow \downarrow \))
- last line pointer (\( \square \))

Figure 3.3.5.1 The Externally-Apparent Structure of Text in \( \overline{\text{BYGONE}} \).
3.3.6 Commands

possesses three major categories of commands:

* data manipulation,
* file management, and
* system/man information interchange.

3.3.6.1 The Structure of Commands

3.3.6.1.1 Parameters

command parameters allow the user to adapt each command to a great variety of different editing operations. The commands may operate on the entire text, on a single line, a part of line, or a single or a set of consecutive characters on a whole succession of lines or upon particular syllables in a whole succession of lines.

3.3.6.1.2 Components of a Command Statement

The command statements are composed of four main components.

* mode designator,
* verb,
* text address, and
* substrings (not required in some commands).

3.3.6.1.2.1 Modes
Figure 3.3.6.1.2.1.1 The Modes of Operations Within BYGONE
3.3.6.1.2.2 Verbs

(1) **inputting** text into the memory of **BYGONE**,  
(2) outputting the contents of the memory,  
(3) **adding/inserting** additional text from  
    the text already in the memory,  
(4) **deleting** portion of text in the memory,  
(5) **moving** a portion of text from one place to  
    another place in memory,  
(6) **changing** portions of text in memory,  
(7) **adjusting** the pointers, and  
(8) requesting certain system information.
3.3.6.2 Data Manipulation Commands

3.3.6.2.1

This command inputs textual data to be manipulated within **BYGONE** (from the specified sources).

![Figure 3.3.6.2.1.1 The General Form of Command.][1]

**Examples:**

(1) 

Command (1) causes the text to be read from **PEESHTAZ** terminal (the default unit for the unspecified source). **BYGONE** prints the sequence number for each line and awaits to receive its contents. As the characters are entered, **BYGONE** places them in its short term memory, corrects the typing errors (if character deleter symbol \(\hat{X}\) and line deleter symbol \(\hat{X}^t\) are used), and when it detects the end of line (by receiving a carriage return \(\hat{\text{BC}}\)) moves the entire line of text from its short term memory into its long term memory....
(working storage area) and repeats the process for the next line until it detects the end of text (escape (\) or a carriage return as the first character of the line). After the end of text is detected, \textit{BYGONE} awaits for the next user's request.

\textbf{Figure 3.3.6.2.1.2 An Example of Entering Text from the Terminal. \textit{BYGONE}}
In Figure 3.19, CR (ןל) and SCAPE (טיר) notify BYGONE of the end of lines and the end of text, respectively.

By inputting the data from more than one source. Where there is more than one source of input, the contents of the first source (the right most source in the parameter list) up to the first end of file mark are placed in the working storage of BYGONE, and then the contents of the second source up to the first file mark are placed in the working storage and so on. Thus the contents of several files can be brought into the working storage with one command.

When the input comes from other devices than the terminal keyboard, the input lines are not echoed (displayed on the screen of the terminal), unless explicitly specified after the inputting process is completed. Figure 3.3.6.2.2.
This command outputs the entire or a portion of the textual data in the memory of the BYGONE destination, with or without the appropriate line numbers, vertical and horizontal pointers.

This command considers both of the horizontal pointers (1↑ and 1↓) whenever the mode in which it is used makes references to horizontal pointers.

Figure 3.3.6.2.2.1 The General Form of Command.
Examples:

Command (1) causes the entire memory of \texttt{BYGONE} to be displayed on the screen of the terminal (the default unit for the unspecified destination) as follows:

\begin{verbatim}
1: NURRZ BRZFRZRSLOAD.
2: SALBA LADT WO AI NAFAYAZ.
3: MYTT INSUY A ORZOLFA.
4: AMNHA RAMA MBMBEME A LAMASIGE A RUSMA SHER SYE.
5: PITA JEUJU.
6: NURRZ BRZFRZ.
\end{verbatim}

Command (2) causes the entire memory of \texttt{BYGONE} to be displayed on the screen of the terminal with the horizontal and vertical pointers (1\textsuperscript{st} = 5, 2\textsuperscript{nd} = 3, 1\textsuperscript{st} = 5, 5\textsuperscript{th} = 3, and 3\textsuperscript{rd} = 5).
Command (3) causes line number 5 of the text residing in the memory of BYGONE to be printed.

If horizontal pointer 1 and 2 are set to 8 and 16 respectively (1\# = 8, 2\# = 16) and vertical pointer 10 is set to 4 (10\# = 4) then command (4) causes the following output to be generated.

Command (5) causes every other line of text to be printed.
Command (6) causes the first three lines of text to be printed in the order of descending line numbers.

Command (7) causes line number 1, 5 and 6 to be printed.
This command allows the insertion of additional textual data after any point within the text residing in the memory of BYGONE.

Figure 3.3.6.2.3.1 The General Form of Command.

This command makes the use of one of the horizontal pointers (1↑), whenever the mode in which it is used references horizontal pointers.

Examples:

\[
\begin{bmatrix}
\{c^-\} & \{c^+\} \\
\{c^-\} & \{c^+\}
\end{bmatrix}\]

Command (1) causes the textual data to be input from
terminal (or other sources when specified) and appended to the contents of the memory of BYGONE.

\[
\text{\textbf{Command (2) is used to add additional text at the beginning of the text in residence within the memory.}}
\]

This command is used to insert additional text after line 2 but before line 3 of the text residing in the memory. The inserted line is identified by 1+2 line number so that the line numbers of the lines before the insertion are left undisturbed. New line numbers can be assigned to the lines of text by the request on behalf of the user.

\[
\text{\textbf{If the horizontal pointer 1 (the only horizontal pointer used by \textbf{\textit{Command}}} \textbf{\textit{command}}} \textbf{\textit{is set to 10 (1 \textarrow{\textup{10}} = 10) and the vertical pointer 3 is set to 2 (3 \textarrow{\textup{2}} = 2), the text to be added is inserted after the 10th character of the second line past line number 2.}}
\]
Command (5) instructs BYGONE to add the additional text at the beginning of every line within the text, if the horizontal pointer 1 is set to ")".

Command (6) instructs BYGONE to add the additional text at the end of every line within the text, if the horizontal pointer 1 is set to "(".

The command causes the additional text to be added after lines 1, 2 and 5.
This command causes a portion of the data, from any point within the existing text, to be deleted and if desired placed onto the specified destination.

Figure 3.3.6.2.4.1 The General Form of Command.

This command makes use of both of the horizontal pointers when the mode in which it is used references horizontal pointers.

Examples:

Command (1) deletes line 2 of the text.
Command (2) deletes line 2, 3, 4, and 5 of the text.

Command (3) deletes line 1, 3, and 5 of the text and places them on the logical unit 3.

Command (4) deletes lines 2 through the end of the text.

This command deletes the strings of characters formed by characters 1 through 10 of every line of text, if the horizontal pointers are set to 1 and 10 (1↓ = 1 and 2↓ = 10) before the command was issued.

This command deletes the strings of characters formed by characters 2 through 11 of the lines of text pointed by and located between the lines pointed by the vertical pointers 2 and 4, if horizontal pointers are set to 2 and 11 (1↓ = 2 and 2↓ = 11).

This command deletes the strings of characters formed by characters 2 through 11 of line 2 and line 4 of the text, if horizontal pointers are set to 2 and 11 (1↓ = 2 and 2↓ = 11).
This command moves portions of text from one place in memory to another place in memory.

Figure 3.3.6.2.5.1 The General Form of Command.

Examples:

Command (1) moves lines 2 through 3 and places them after line 4.

This command moves the lines of text pointed by vertical pointers 2 and 3, and also the lines of text located between the lines pointed by vertical pointers and places them after the line of text pointed by vertical pointer 4.
Command (3) moves the lines of text pointed by vertical pointers 3 and 5 and also the lines of text located between the lines pointed by the vertical pointers and places them in front of the text.

Command (4) moves the lines of text pointed by vertical pointers 3 and 5 and also the lines of text located between the lines pointed by vertical pointers and places them at the end of the text.

Command (5) moves line 3 of the text and places it after line 4.

Command (6) accomplishes the same task done by command (5).
This command searches the text for a specified string of characters and depending on the parameter specification the located string is replaced by another string or another string is inserted immediately before or after it.

Figure 3.3.6.2.6.1 The General Form of Command.

This command makes use of both of the horizontal pointers when the mode in which it is used references horizontal pointers.

Examples:

Command (1) conducts a search for the string "" through the contents of lines 3 and 5 and all the lines in between them. Wherever the string is found, it is replaced by the string "".
Command (2) accomplishes the same task as in (1), in addition prints the lines in which the string "(" is replaced by the string ")".

Command (3) searches through the lines of text pointed by vertical pointers 2 and 5 and the lines of text located in between the pointed lines for the string "I" whenever the string is found, the string "," is inserted immediately to the left of it.

The result of the action taken by command 4 is the same as that of command 3 (if the lines of text through which the search is conducted remain the same as those in (4)).

Command (5) searches the segment of the text indicated by the horizontal pointers of every line within the memory of BYGONE for the string " " whenever the string " " is found, the string " " is inserted immediately to its left.

Command (6) deletes all the occurrences of the string "*" within the entire text and lists the modified lines.
After the insertion of characters within the text (by using 'IPA' or 'tY4' commands) the possibility of the length of the modified line to become greater or smaller than a certain band exists. Also, it is desired in many applications to make the lengths of the lines of text as equal as possible. To circumvent the above problems command can be used.

Figure 3.3.6.2.7.1 The General Form of Command.

Examples:

if the following text resides within the memory of :
Command (1) causes the text to be rearranged as follows:

1. 
2. 
3. 
4. 
5. 

assures that each line ends at or before the 34th column.

If the following command

is executed, the text used in example 1 is modified as follows:

Then, the execution of command (2) adjusts the above line of text to that shown below:
3.3.6.3 File Management Commands

3.3.6.3.1 Files

A file is a set of records each of which is composed of a set of elementary data items (pieces of information composed of \( \star \) characters). There are two file organizations available under \( \star \): sequential and random access, with which other file organizations may be created by the user.

Both sequentially organized and random access files can be input/output and manipulated. To generate random access and sequential files, the following commands may be issued:

- Logical unit 1 is associated with a random access scratch file,
- The contents of memory are written onto logical unit 1,
- And the contents of logical unit 1 is saved under the name

The above commands create a random access file.
logical unit 1 is associated with a sequential scratch file, the contents of memory BYGONE is written onto logical unit 1, and the contents of logical unit 1 is saved under the name \\

The above commands create a sequential file.
3.3.6.3.1.1 Sequential Files

The records of sequential files are stored in successive segments of a file, representing many natural patterns of information.

![Diagram of Sequential File Structure]

Figure 3.3.6.3.1.1.1 Externally-Apparent Structure of Sequential File.

A sequential file can have one or more end-of-file marks but can possess only one end-of-data mark.

★ The internal organization of sequential files within PEESHTAZ is designed so that the degree of efficiency with which input/output activities are accomplished is increased and storage space is conserved.
A sequential file consists of a set of logical records, each of which is composed of as many of user's records as can reside within a physical record: the buffer area whose length is fixed and is dependent upon the configuration of the machine on which runs.
Figure 3.3.6.3.1.1.2 Internal Structure of Sequential Files.
3.3.6.3.1.2 Random Access Files

The advantage of sequential organization is the rapid access to successive records. However, in order to access the \( n^{th} \) record, the first \( n-1 \) records must be traversed (the file is scanned until the \( n^{th} \) record is found). Although, the internal organization of files within process sequential files nevertheless, the organization is not suitable for retrieving records out of sequence. offers random access files for those transactions which do not follow any sequential patterns.

In random access organization there are addresses (relative to the addresses of the beginning of the files) associated with each record and with each of the elementary data items within the records. To access the elementary items individually they must reside within word boundaries of the machine on which runs.
Figure 3.3.6.3.1.2.1 Externally Apparent Structure of a Random Access File.
allows the examination and modification of random access files without inputting their entire contents into memory. This can be accomplished by opening the file, moving the read/write pointer to the desired address, and requesting the appropriate action (read or write only).

Example:

logical unit 1 is associated with the random access file stored under the file name the read/write pointer is moved to the word whose address is 4, three words (9 characters in CDC3300) are read into the memory of the input is written on the screen of the terminal, text is entered from the keyboard into the memory of (replacing its previous contents), the read/write pointer is moved to the word whose address is 4, the first three characters of the contents of memory is written onto logical unit 1 (replacing the information read previously from logical unit 1 by the information entered
from the terminal.

Figure 3.3.6.3.1.2.3 Contents of the File Before the Modification.

Figure 3.3.6.3.1.2.4 Contents of the File After the Modification.
This command (opens files) associates logical unit numbers with the specified files and/or other hardware devices.

Figure 3.3.6.3.2.1 The General Form of Command.

Command (1) associates logical unit number 1 with the saved file stored under.

Command (2) associates logical unit numbers 1 and 2 with a random access and a sequential scratch file, respectively.

Command (3) associates logical unit numbers 1 and 2 with logical unit numbers 10 and 3, respectively.
This command closes the specified files whose logical units are specified. The associations between the specified files and logical units are destroyed.

Example:

Command (1) closes logical unit 1.

Command (2) closes logical units 1, 5 and 10.
This command saves the contents of the disk files whose logical units are specified and is capable of protecting them against future destruction.

![Diagram of command format]

**Figure 3.3.6.3.4.1 The General Form of Command.**

**Examples:**

Command (1) instructs to store the contents of logical unit 1 under the file name.

Command (2) accomplishes the task done by command 1 and in addition stores the contents of logical unit 3 under the name, and protects it against future destruction.
This command destroys the unprotected files whose names are specified.

\[
\text{[file name}_1\text{]} \ldots \text{[file name}_n\text{] file name}_m
\]

Figure 3.3.6.3.5.1 The General Form of Command.

Example:

\[
\text{[file name}_1\] \ldots \text{[file name}_n\] file name}_m
\]

Command (1) destroys the contents of the disk file called \text{[file name}_1\] (it removes the name from \text{[file name}_1\] directory and returns the file space to the list of available space).

Command (2) destroys the contents of both \text{[file name}_1\] and \text{[file name}_2\] files.
3.3.6.3.6 +

This command protects the specified logical units or saved files against future destruction (makes them read-only files).

\[
\text{source}_n \quad \ldots \quad \text{source}_1 \quad \text{source} \quad +
\]

Figure 3.3.6.3.6.1 The General Form of + Command.

Examples:

\[1+\mathbb{F}(1)\]

Command (1) protects logical unit number 1 against possible destruction until the termination of the current editing session.

\[-1+\mathbb{F}(2)\]

Command (2) protects the saved file against possible destructions on permanent basis, even after the termination of the current editing session.

\[-1+\mathbb{F}(3)\]

Command (3) accomplishes the same tasks as (1) and (2).
This Command removes the protection from a protected file.

Figure 3.3.6.3.7.1 The General Form of Command.

Examples:

Command (1) removes the file protection from logical unit 1, if it is not associated with a saved file. To remove the file protection from a saved file, its name must be specified as in command (2).

Command (2) removes the file protection from the saved file.

Command (3) accomplishes the tasks done by command (1) and (2).
This command requests description of the files whose logical units or names are specified.

The description contains the following information:

1. type of file
   - disk
   - plotter
   - terminal

2. mode of file
   - BCD
   - BINARY

3. number of records,
4. number of characters,
5. maximum length of the lines,
6. minimum length of the lines,
7. locations of the end-of-file, and
8. status
   - protected
   - unprotected
This command sorts a specified file according to the given keys in ascending or descending orders and places the sorted files on the specified destination:

\[
\begin{align*}
\text{key}_1 & \quad \text{destination} \\ 
\text{key}_2 & \quad \text{source} \\
\vdots & \quad \text{key}_n
\end{align*}
\]

Figure 3.3.6.3.9.1 The General Form of Command.

Where keys are specified by the beginning and the ending character positions of the fields on which sorting is to be done and the natural ordering of the characters is from right to left and the collating sequence is given in Figure

Examples:

\( (1) \)

Command (1) causes the contents of the file saved under the file name to be sorted on the first 5 columns of each record in descending order and the sorted contents to be written onto logical unit 1.

\( (2) \)

Command (2) sorts the file according to command (1) and then sorts it again on columns 10 through 14 in ascending order.
This command merges two files considering the specified keys and places the outcome on the desired destination.

Figure 3.3.6.3.10.1 The General Form of Command.

Example:

Command (1) merges files and places the outcome on logical unit 5.
This command forward spaces one or more logical records on the specified logical units.

Figure 3.3.6.3.11.1 The General Form of \( \text{\textbackslash n} \) Command.

Examples:

1. \( \text{\textbackslash } \) (1)

   Command (1) forward spaces 1 record on logical unit 1.

2. \( \text{\textbackslash } \) (2)

   Command (2) forward spaces 5 records on logical unit 1.

3. \( \text{\textbackslash } \) (3)

   Command (3) forward spaces 5 records on logical unit 1, forward spaces one record on logical unit 2 and forward spaces 3 records on logical unit 5.
This command back spaces one or more logical records on the specified logical units.

![Diagram](3.3.6.3.12.1) The General Form of Command.

Examples:

```
(1)
Command (1) backspaces 1 record on logical unit 1.

(2)
Command (2) backspaces 5 records on logical unit 1.

(3)
Command (3) backspaces 5 records on logical unit 1, backspaces one record on logical unit 2 and backspaces 3 records on logical unit 5.
```
This command searches forward looking for one or several end-of-file marks on the specified logical units.

Figure 3.3.6.3.13.1 The General Form of \( \) Command.

Examples:

\[ \] (1)

Command (1) searches forward (moves the read/write pointer forward) for the occurrence of the first end-of-file mark.

\[ \] (2)

Command (2) searches forward past the first 5 end-of-file marks.

\[ \] (3)

Command (3) searches forward past the first five end-of-file marks on logical unit 1, past the first end-of-file mark on logical unit and past the third end-of-file mark on logical unit 5.
This command searches backward looking for one or several end-of-file marks on the specified logical units.

Figure 3.3.6.3.14.1 The General Form of Command.

Examples:

1. \[ (\text{rep. fac.}) \text{ logical unit} \rightarrow (\text{repetition factor}) \text{ logical unit} \rightarrow (\text{repetition factor}) \text{ logical unit} \]

Command (1) searches backward (moves the read/write pointer backward) for the occurrence of the first end-of-file mark.

2. \[ (\text{rep. fac.}) \text{ logical unit} \rightarrow (\text{repetition factor}) \text{ logical unit} \rightarrow (\text{repetition factor}) \text{ logical unit} \]

Command (2) searches backward past the first 5 end-of-file marks.

3. \[ (\text{rep. fac.}) \text{ logical unit} \rightarrow (\text{repetition factor}) \text{ logical unit} \rightarrow (\text{repetition factor}) \text{ logical unit} \]

Command (3) searches backward past the first five end-of-file marks on logical unit 1, past the first end-of-file mark on logical unit and past the third end-of-file mark on logical unit 5.
This command rewinds (moves the read/write pointer to the beginning of the contents of) one or several specified logical units.

![Diagram showing the general form of the command.]

Figure 3.3.6.3.15.1 The General Form of Command.

Examples:

1. \( \text{REWIND} \) (1)

This command rewinds logical unit 1.

2. \( \text{REWIND} \) (2)

This command rewinds logical units 1 and 3.
This command moves the read/write pointer to the end-of-data point of one or several logical units. The file may have several end-of-file marks but has one end-of-data only.

![Figure 3.3.6.3.16.1 The General Form of Command.](image)

Examples:

1. This command moves the read/write pointer of logical unit 1 to the end-of-data point.

2. This command moves the read/write pointer of logical units 1 and 3 to their end-of-data points.
3.3.6.4 System/Man Information Interchange Commands

3.3.6.4.1

This command can be used to set the vertical pointers to lines whose numbers, relative or pointer addresses are given and/or set the horizontal points to specified positions or character positions pointed by other horizontal pointers. It can also be used to set the vertical and horizontal tabs.

Example:

\[
\begin{align*}
\text{Command (1) sets vertical pointers 1 and 3 to point to line 4 and the last line of text respectively.} \\
\text{Command (2) sets vertical pointer 1 and horizontal}
\end{align*}
\]
pointer 2 to point to the second line past the line pointed by vertical pointer 3 and to the beginning of the line, respectively.

\[ \text{(3)} \]

Command (3) sets the horizontal tab to 10, 20, and 30.

\[ \text{(4)} \]

Command (4) sets the vertical tab to 1 and 30.
This command is used to get a list of all the active pointers (the pointers set by the user during the editing session).

Examples:

1. \( \text{Command (1) requests for the line number of the line of text to which vertical pointer 2 is pointing.} \)

2. \( \text{Command (2) requests for the character position and for the line number to which horizontal pointer 1 and vertical pointer 3 are pointing respectively.} \)

3. \( \text{Command (3) requests for the character positions to which all of the horizontal pointers are pointing.} \)
Command (4) requests for the line numbers of the lines to which all of the vertical pointers are pointing.

Command (5) requests for the character positions and the line number of the lines to which all of the horizontal and vertical pointers are pointing.
This command causes to process sets of commands (tasks), on non-conversational basis.

\[
[\text{source}_1] \ldots [\text{source}_n] [\text{source}_1] \ldots [\text{source}_n]
\]

Figure 3.3.6.4.3.1 The General Form of Command.

The sources furnish with the tasks to be sequentially performed (eliminating the need for the user to monitor the process command by command when such user/interaction is not necessary).
This command allows the user to write descriptive comments during a text manipulation session. The descriptive information is displayed on the screen of the terminal without any effects on the text being manipulated.

\[
\begin{array}{l}
\text{source}_n \quad \ldots \quad \text{source}_1 \\
\end{array}
\]

Figure 3.3.6.4.4.1 The General Form of Command.

If the source is not specified with the command, assumes that the descriptive information is entered from the terminal and the end of it is specified by an end-of-file or scape. ( ).
3.3.6.4.5

This command prints the current clock time in the following form:

\[ \text{XX : XX XX} \]

\[ \text{the minutes} \quad \text{the seconds} \]

\[ \text{the hours} \]

Figure 3.3.6.4.5.1 The General Form of Command.
This command prints the current date in the following form:

\[ \text{year \, month \, day} \]

Figure 3.3.6.4.6.1 The General Form of تاریخ Command.
This command prints the total CPU time spent since entering \texttt{PEESHTAZ} and the total CPU available under the user's account in rials.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure3.3.6.4.7.1}
\caption{The General Form of \texttt{Command}.}
\end{figure}
This command causes the contents of a section or the entire computer memory to be dumped on a specified destination in either binary or decimal number systems.

Figure 3.3.6.4.8.1 The General Form of Command.
This command prints a list of the names of all the files which belong to the user with or without brief or detailed description of each.

Figure 3.3.6.4.9.1 The General Form of Command.

Examples:

1. This command lists the names of the files that are in existence under the user's job number.

2. Command (2) generates a listing which contains the names of the files saved under the user's job number with detailed description of each file. It places the generated information on logical unit 2.

3. Command (3) prints a listing which contains the names of the files saved under the user's job number with brief description (number of records and status of each file).
This command allows the user to relinquish the relationships established among certain components of BYGONE. It can be used to reset

- one, several, or all of the vertical and/or horizontal pointers, except the ones that are not subject to any changes imposed by the user ([, ], (, )],
- the horizontal or vertical tabs,
- all the active logical unit numbers,
- a combination of the above activities, by specifying more than one of the stacked items Figure 3.3.6.4.10.1 in the parameter list, and
- all of the above, by not specifying any parameter list.

Figure 3.3.6.4.10.1 The General Form of Command.
Examples:

Command (1) resets the user's environment within BYGONE, by disassociating all of the logical unit numbers with their corresponding hardware devices and undefining the horizontal and vertical tables and all of the horizontal and vertical pointers.

Command (2) closes logical units 1 and 10.

Command (3) in addition to the tasks done by command 2, undefines the vertical pointer 10 and the horizontal tabs.
3.3.7 The Internal Structure of BYGONE

The editor is stored on a system disk in form of an overlay (the absolute machine form). When the user requests to use the editor by typing under the control mode of PEESHTAZ, the message: current time current date

is printed and the control is transferred to subsystem.

The subsystem is composed of the following major components:

Memory organizer

Command Processor \[\{\text{data manipulation, file management, system information interchange}\}\]
3.3.7.1 Text Memory Organization

The storage area in which the user's text is kept is allocated dynamically and is managed by using a doubly linked circular list.

Figure 3.3.7.1.1 The Dynamic Structure of the Text Bank.

The structure makes it possible to access all of the text starting at any given point and moving in either of the possible directions.

The dynamic allocation of the storage permits the user to occupy only as much storage as is necessary. The blocks of storage are obtained from the memory pool (as more room is needed for the additional text) and the blocks are freed to the memory pool when a portion of text is deleted, thus eliminating the need for garbage collection mechanisms.

The blocks consist of 84 words and are organized as follows:
The text is a set of such blocks:

Figure 3.3.7.1.3 The Organization and Components of the Text Bank.
The allocation of storage for the textual data is done as illustrated below:

(1) **before any text is entered**

    list of available storage

    memory pool 1  memory pool 2  text bank

    Memory pool 1 pointer

(2) **after one line of text is entered**

    memory pool 1  memory pool 2  text bank
(3) after another line of text is added

memory pool 1

memory pool 2

(4) after one line of text is deleted

memory pool 1

memory pool 2

text bank

pointing to the same location as in (3)
after one more line of text is added

memory pool 1

memory pool 2

text bank

pointing to the same location as in (3)

The dynamic allocation of storage is summarized in Figure 3.3.7.1.4.
initialize memory pool 1 and memory pool 2.

... determine the total memory for the user's text; set pool 2 to be empty.

Free the block (whose contents are to be deleted) to memory pool 2.

Add or delete text?

Add

Is memory pool 2 empty?

True

Get a new block of storage from pool 1 to place the text into it.

Add the new block to the text bank.

Figure 3.3.7.1.4 The Dynamic Allocation of Storage.
3.3.8 Command Processor

The command processor of BYGONE processes all the valid commands and detects the invalid ones by first recognizing the command and then transferring the control to the appropriate section of BYGONE for the interpretation.

Figure 3.3.8.1 The Command Processor.
The General Purpose/Interactive Programming Language of is developed to create the interactive general purpose programming language which uses the alphabet and the grammar of Farsi and is machine independent (assuring transferability), relatively easy to learn and use (assuring popularity), structurally powerful in terms of problem solving (assuring effectiveness), readable in the sense that a reasonably good understanding of the program can be obtained by reading a listing of the source program (assuming the ease of documentation, and as the result, providing ease with which the programs can be extended, modified, and maintained). The interactive nature of the language allows the existence of a close man/machine interaction and dynamic capabilities.

3.4.1 Data and Its Control

Data is the information transformed by an algorithm (like the input and the output strings of Turing machines). The means by which the data is represented by the language (primitive data structures), the set of operations for data manipulation, and the mechanisms by which it is controlled, form some of the major differences among languages in existence.
Figure 3.4.1.1 illustrates the representation of data within DARRIE.

Figure 3.4.1.1 Data Representation Within DARRIE.
The data within DARRIE belongs to two categories: programmer-defined and system-defined.

3.4.1.1 Programmer-Defined Data

The category of data which is explicitly defined by the programmer and is directly manipulated by the programmer's program (simple variables, homogenous/heterogeneous data structures, stacks and other specially organized data) is programmer-defined data.

3.4.1.2 System-Defined Data

System-defined data consists of the housekeeping data used by processor to make the execution process possible (input/output buffer, stacks of sub-routines return points, data structure descriptors, dynamic/static storage manager, and etc.)

Both categories (programmer and system-defined) are represented by means of primitive data structures and non-primitive structures.

3.4.1.3 Primitive-Data Structures

The primitive-data structures are represented by either simple variables or organized group of simple variables.
3.4.1.4 Non-Primitive Structures

Non-primitive structures are organized groups of primitive data structures which are explicitly defined by the programmer (special forms of trees, lists, specially organized files, etc).

3.4.1.5 Simple Variables

Simple variables are basic units of information which can be viewed as locations of memory with corresponding values (contents). All data structures within DARRIE are composed of simple variables, arranged differently. Simple variables are referred to by identifiers (the symbols formed from the alphabet of PEESHTAZ as defined in Figure 3.4.1.5.1).

```
<alphabetic character> ::= "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
<numeric character> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<dash> ::= "-"
<identifier> ::= <alphabetic character> | <identifer> <numeric character> | <alphabetic character> <identifier> | <identifier> <dash> <identifier>
```

Figure 3.4.1.5.1 The Definition of Identifiers Within DARRIE.
Examples of valid identifiers:

362

Examples of invalid identifiers:
3.4.1.5.1 Physical Positioning (Environment): INTERNAL and EXTERNAL Attributes

Simple variables in can be either INTERNAL or EXTERNAL. The INTERNAL variables are those which reside within the main memory of computer during the execution (like variables of other programming languages), FORTRAN, ALGOL, PL/1, and etc.) The EXTERNAL variables however, reside in the main memory of computer only when they are referenced. They are kept outside of the main memory (on an external storage device such as a disk file) while in existence but not active. This capability enables the programmer to solve problems with larger-than-available memory requirements by creating a virtual memory environment. Any reference to an EXTERNAL variable (which in most cases would involve reference to a large organized group of simple variables) would cause the system to swap (roll in) its contents from an external storage device into the memory of the machine. If the content of an EXTERNAL variable is modified the modified version is swapped (rolled out) from the memory onto an external storage device when the variable becomes inactive. Otherwise, if the contents of the swapped in EXTERNAL variable do not require any updating, no swapping out is performed. Swapping out is also eliminated when there is no shortage of storage. Swapping activities can,
however, be monitored by the programmer when explicitly specified. This technique allows for efficient storage handling for successive references to the same EXTERNAL variable and at the same time enables the programmer to disregard it when appropriate (in a time sharing environment, to reduce the length of time in which main memory is occupied and/or when the computing environment provides fast input/output activities).

3.4.1.5.2 Life Expectancy (Persistence): PERMANENT, SEMI-PERM ANENT and TEMPORARY Attributes

The INTERNAL and EXTERNAL variables fall into two groups of persistence attributes: implicit and explicit. PERMANENT (\texttt{PER}) and TEMPORARY (\texttt{TEM}) attributes result in implicit storage allocation.

PERMANENT variables are created at the time of activation of the main program and reside in memory until their destruction time: the time at which the main program is terminated - similar to the STATIC storage class in PL/1 and the variables in FORTRAN.

TEMPORARY variables are created at the time of activation of containing block or a procedure and reside in memory until the exit from the containing block or procedure is made - similar to LOCAL variables in ALGOL and AUTOMATIC variables in PL/1.
Controlled variables must be explicitly created by the execution of ALLOCATE and destroyed by RELEASE statements.

3.4.1.5.3 Popularity (Scope): PUBLIC, PRIVATE and SEMI-PUBLIC Attributes

PUBLIC, PRIVATE, and SEMI-PUBLIC attributes specify the scope of a variable: whether it is known throughout the entire program, within the declaring block or within certain blocks.

PUBLIC variables are known to all the blocks and procedures. PRIVATE variables are known only to the declaring blocks and a variable that is declared to be SEMI-PUBLIC is known to all the blocks which declare the same variable as SEMI-PUBLIC.

3.4.1.5.4 READ ONLY

The contents of a variable that is declared READ ONLY can be modified only within the defining block or procedure. This attribute is quite useful in situations where there is a need for declaring a variable PUBLIC enabling the other blocks or procedures to have access to its contents but not allowing them to disturb its contents.
3.4.1.5.5 INTEGER

INTEGER ( C"-P") attribute declares a variable to represent integer data.

The precision attribute can be attached to INTEGER ( C"-P") attribute if other than default precision (dependent upon the machine on which runs) is desired.

The precision is expressed as an integer constant enclosed in parenthesis following INTEGER ( C"-P") attribute. The integer indicates the total number of decimal digits with which the contents of the variable are represented.

3.4.1.5.6 REAL

REAL ( C"-P") attribute declares a variable to represent real (FLOATING POINT ( C"-P") and FIXED POINT ( \text{سریا} )) data.

The precision attribute that can be attached to a REAL FIXED-POINT variable is expressed as a pair of integers (I.R.) "I" defines the field width in terms of digits and "R" indicates the scale (the position of the decimal point). For example if the precision is (8, 3) then the following representation is assumed: 

![Diagram of real number representation with 8 digits and 3 decimal places]
3.4.1.5.7 COMPLEX

COMPLEX ( ) declares a variable to represent complex data.

3.4.1.5.8 CHARACTER

CHARACTER ( ) declares a variable representing character string data.

<characters> :: = "any sequence of characters not containing"

<character string> :: = "characters"

There is a length (the number of characters within a variable) parameter associated with each CHARACTER variable, the length can be fixed or variable. If it is fixed, the length may be specified by an integer* (enclosed in parenthesis following CHARACTER attribute) and if it is of variable type, attribute VARYING ( ) must follow the CHARACTER attribute.

3.4.1.5.9 LOGICAL

LOGICAL ( ) attribute specifies a variable which represents logical values.

logical value :: = TRUE ( ) | FALSE ( )

*If a character variable has a fixed length but its length is not specified, then the default length, 1, is assumed.
3.4.1.5.10 LABEL

LABEL (\label{p}) attribute specifies a variable which represents statement labels.

\<label\> ::= \<identifier\> | \<unsigned integer\>

3.4.1.5.11 PROCEDURE

\textit{PROCEDURE (\procedure{p})} specifies that the declared variable represents a procedure name.

\<procedure name\> ::= \<identifier\>

3.4.1.5.12 POINTER

\textit{POINTER (\pointer{r})} attribute declares that the declared variable represents the machine address of another variable.
3.4.1.5.13 Simple Variable Declaration

In order to resolve the references made to variables, declaration statements are used both to describe the variables and to impose constraints on them as to what kind of information can be held in them. Declaration statements cause the execution of mapping algorithms which result in the appropriate assignment of descriptors to variables.

The variables within a program are brought into existence by declaration statements and all needs to know about the variables to resolve references to them in the course of execution is obtained from the descriptors.

The declaration statements for simple variables show all the different paths (from the root to the leaves) of the tree in Figure 3.4.1.5.13.2. The general form of the declaration statement for simple variables is given in Figure 3.4.1.5.13.1.

A set of attributes for a variable (or several variables with the same attributes) can be obtained by traversing downward on one and only one of the paths of the declaration tree and obtaining the attributes placed at the nodes. For any missing attributes the attributes marked by "*"s in Figure 3.4.1.5.13.1 are assumed. The entire declaration statement may be excluded which results in the assumption that all the undeclared variables are of SIMPLE (ساده) category and have the default attributes.
Figure 3.4.1.5.13.1 The General Form of the Declaration Statement for Simple Variables.
Figure 3.4.1.5.13.2 The Declaration Tree of Simple Variables.

† lines indicate default routes. Traversal of default routes would result in the assumption of default attributes.
Figure 3.4.1.5.13.3  The Definition of Simple Variable Declaration Statement.
3.4.1.5.14 Simple Variable Descriptor

The descriptors for all simple variables are constructed uniquely as in Figure 3.4.11 and they all contain the following information:

(1) the category:
   SIMPLE

(2) the environment:
   INTERNAL
   EXTERNAL

(3) the persistence:
   PERMANENT
   CONTROLLED
   TEMPORARY

(4) the scope:
   REGULAR
   PRIVATE
   PUBLIC
   SEMI-PUBLIC

(5) the changeability:
   READ-ONLY
   READ/WRITE

(6) the type:
   INTEGER
   REAL
   COMPLEX
There are several moments in which binding (association of attributes to variables) occur in DARRIE. In general the descriptors and storage for all the PERMANENT (٥٢٤) variables are allocated at the beginning of the program execution. The TEMPORARY (٥٥٢) variables are created automatically when a block is entered and are destroyed when the block is terminated. CONTROLLED (٥٥٤) variables are only created when ALLOCATE (٥٥٥٤٩) statements, referring explicitly to the variables, are executed and are destroyed upon the execution of RELEASE (٥٥٥٤٩) statements.
3.4.1.5.15 Simple Variable Accessing

Simple variables are accessed whenever their identifiers appear in executable (non-declarative) program statements. The symbol table and the list of descriptors are used to resolve accessing for the following possible purposes:

1. Getting the contents of a variable,
2. Storing a value into the contents of a variable (if the variable is not restricted to READ-ONLY), and
3. Creating/destroying a variable by using ALLOCATE (RELEASE) statements (if the variable belongs to CONTROLLED class).

When a variable is accessed may have to perform several tasks before which the accessing request is satisfied. When an EXTERNAL variable is referenced, swapping in/swapping out activities may be required. Or when a variable with variable size is referenced, descriptor/link list manipulation may be required. The performance of several tasks before satisfying the programmer's request occurs in many other situations, yet it is not apparent to the programmer and does not affect the program's results. However, the proper selection of data attributes and
storage conservation can affect the amount of storage management overhead and thus can increase machine throughput. Figure 3.4.1.5.15.1 illustrates some of the major activities of accessing algorithm for simple variables.
requests for accessing variables

DATA ACCESSOR SUBSYSTEM

The symbol table created by mapping algorithms which process declaration statements is searched for the identifier which represents the variable to be accessed

not found

found

since the identifier is in the symbol table, the address of its descriptor is obtained

From the descriptor the address of the variable is obtained

the control is returned to the appropriate segment of MAIN to SPART

get the contents of the variable to replace its contents by a value, or complete the RELEASE request

the table of CONTROLLED variables created by mapping algorithms which process declaration statements is searched for the identifier which represents the variable to be accessed

not found

found

the descriptor for the variable is created by the mapping algorithms and the identifier which represents the variables placed in the symbol table

the control returns to the user's program

since at this point the only possibility is that the request must be of an undeclared variable, the control is passed on to the mapping algorithms to generate the descriptor with default attributes and place its identifier in symbol table

Figure 3.4.1.5.15.1 Some of the Major Activities of Accessing Algorithms for Simple Variables.

solid arrows represent the direction of the logic flow
dotted arrows represent communication lines
3.4.1.5.16 Simple Variable Initialization

Figure 3.4.1.5.16.1 shows the definition and the general form of the initialization statement for simple variables.

The initialization statement:

\[ \text{initializes variable which must be a CHARACTER (57) variable to a string of characters:} \]

\[ \text{initializes simple variables and to zeroes.} \]

The initialization statements within \text{DARRIE} are of non-executable type (the same as declaration statements). They must appear before any executable statements within block or procedure and are processed only once during the translation phase for PERMANENT (1) variables and during the execution upon the entry into each block or procedure. Thus they differ from the assignment statements which are executed every time they appear in the sequence in which the execution proceeds.

\text{DARRIE} sets all the uninitialized variables to zeroes at the time in which they are brought into existence.
Figure 3.4.1.5.1

\[ \begin{align*}
\text{initializing value: identifier} & \quad \text{initializing value: identifier}_n \ldots \text{initializing value: identifier}_1 \\
\end{align*} \]

Figure 3.4.1.5.16.1  The General Form and the Definition of the Initialization Statement for Simple Variables.
3.4.1.6 Organized Group of Simple Variables

There are two techniques for grouping data elements within DARRIE:

(1) homogeneous n-dimensional arrays, and
(2) heterogeneous n-dimensional arrays.

3.4.1.7 Homogeneous n-Dimensional Array

Homogeneous n-dimensional arrays (defined in Figure 3.4.1.7.1) can have either fixed-size length or variable-size length (allowing for the number of elements of which the array is composed to increase or decrease dynamically during program execution). The size of each element, however, remains fixed for both the variable-size length and the fixed-size length arrays.

\[
\begin{align*}
&<\text{subscript expression}> ::= <\text{arithmetic expression}> \\
&<\text{subscript list}> ::= <\text{subscript expression}> | <\text{subscript list}> \\
&<\text{subscript expression}> \\
&<\text{array identifier}> ::= <\text{identifier}> \\
&<\text{array}> ::= (<\text{subscript list}> ) <\text{array identifier}>
\end{align*}
\]

Figure 3.4.1.7.1 The Definition of Arrays Within DARRIE
3.4.1.7.1 Homogeneous Array Declaration

The declaration statements for homogeneous arrays allow the traversal of all the different paths (one at a time) from the root to the leaves of the array declaration tree. There are paths allowing certain nodes to be by passed; the traversal on those paths cause the default attributes (those attributes marked by *'s in Figure 3.4.1.7.1.1).

The general form of the declaration statement depicts the capabilities of DARRIE in allowing the homogeneous arrays to be organized in stack and/or queue disciplines.

Figure 3.4.1.7.1.3 defines the array declaration statement.
Figure 3.4.1.7.1.1 The General Form of the Declaration Statement for Homogeneous Arrays
Figure 3.4.1.7.1.2 The Declaration Tree for Homogeneous Arrays.

If the array is declared as having variable size length then at least one of the upper bounds in the list of bound pairs must be stated as &. so can recognize the array as an array with variable length when it is not explicitly declared so by variable length attribute.

When an identifier is declared as an array and its bound pair list is not specified, it is assumed to be a one dimensional array whose lower bound on its only subscript is 1.
Figure 3.4.1.7.1.3 The Definition of the Declaration Statement for Homogeneous Arrays.
3.4.1.7.2 Homogeneous Array Descriptor

The homogeneity and fixed-size elements imply that one data descriptor is sufficient for describing all elements of an array with a fixed-size length and also one data descriptor (with the same structure and containing similar information of that of fixed-size length) is sufficient for describing all elements of a variable-size length array.

<table>
<thead>
<tr>
<th>homogeneous array</th>
<th>fixed size</th>
</tr>
</thead>
<tbody>
<tr>
<td>length of each element</td>
<td>length of array</td>
</tr>
<tr>
<td>environment</td>
<td>persistence</td>
</tr>
<tr>
<td>scope</td>
<td>changeability</td>
</tr>
<tr>
<td>type</td>
<td></td>
</tr>
<tr>
<td>machine address of the first element, $x$</td>
<td>lower bound on the first subscript, $l_1$</td>
</tr>
<tr>
<td>upper bound on the first subscript, $u_1$</td>
<td>:</td>
</tr>
<tr>
<td>lower bound on the $n^{th}$ subscript, $l_n$</td>
<td>upper bound on the $n^{th}$ subscript, $u_n$</td>
</tr>
</tbody>
</table>

Figure 3.4.1.7.2.1 The Descriptor of a Homogeneous $n$-Dimensional Array with Fixed-Size Length.
The information contained in the descriptor of a homogeneous n-dimensional array with variable size length is similar to that of a fixed-size array with the following differences:

1 (1) Since the length of array changes dynamically, the contents of the location that contains the length of array is subject to modification during the program execution.

(2) Since the upper bounds on the subscripts change dynamically as the array length increases or decreases, the locations which hold the upper bounds on the subscripts are updated during the program execution.

(3) The machine address of the first element is the pointer to a linked list which holds the array elements.

The homogeneity and the fixed-size length of arrays also imply that the array can be composed of a set of sequentially distributed simple variables. Each element can be accessed by means of one (for singly-dimensioned arrays) or more than one (for multidimensional arrays) integer subscript(s) which simply indicate(s) the position in sequence at which the element is stored. All multidimensional arrays are stored in row-major form.
However, the variable-size length arrays impose the necessity of less straightforward data representation than that of the fixed-size length arrays. They require the utilization of run-time descriptor manipulation.

Figure 3.4.1.7.2.2 Fixed-Size Length and Variable-Size Length Arrays Storage Layout.
3.4.1.7.3 Homogeneous Array Accessing

Arrays can be referenced in the following ways:

(1) the entire array can be referenced by the appearance of its identifier (alone) in those program statements which allow such referencings,

(2) in case of a multidimensional array with fixed size length a particular segment of an array (composed of several elements) can be accessed.

For example if \( \begin{pmatrix} a & b \\ c & d \end{pmatrix} \) is a matrix, the \( i \)th row is denoted by: \( (\cdot, i) \) and the \( j \)th column is denoted by: \( (i, \cdot) \).

If \( \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \) is declared as \( (1, \cdot, 2, \cdot) \), then \( (\cdot, i, \cdot, j) \) refers to all the elements marked by '+'s, \( (\cdot, i, \cdot, j) \) refers to all the elements marked by '/'s, \( (\cdot, i, \cdot, j) \) refers to all the elements marked by '%'s, \( (\cdot, i, \cdot, j) \) refers to all the elements marked by '.'s, etc.
accessing the elements of variable-size arrays tends to be relative due to their storage representations. In the variable-size arrays can be organized into two primitive disciplines:

- queue (queue), and
- stack (stack).

Although any element of the variable-size arrays can be referenced using the appropriate subscript list (representing the relative positions of the element), a set of program statements are included within the repertoire of for more natural manipulation of such disciplines.

For queues which represent FIFO (first in first out) system of queuing, the following statements can be used to place a cell at the rear of a queue and get the information part of the cell and/or delete the cell at the front of a queue:

For stacks which represent LIFO (last in first out) system of queuing, the following statements can be used to place (push) a cell onto a stack and get (pop) the information part of the cell and/or delete the cell on the top of a stack.
array identifier (identifier, identifier)

identifier (identifier, array identifier)

array identifier (identifier, identifier)
3.4.1.7.4 Homogeneous Array Initialization

One or several elements of a homogeneous array can be initialized using the same method discussed in 3.4.1.1.5.15 (the initialization of simple variables) except that the element must be identified by its identifier and its subscripts.

To initialize all of, or a segment of a multi-dimensional array, the initialization statement in Figure 3.4.1.7.4.1 can be used.

```
{ initializing,...initializing:array
  value
  identifier
}
```

Figure 3.4.1.7.4.1 The General Form of Homogeneous Array Initialization Statement.

The initialization statement:

\[
\text{initializes the first six elements of the array to 30, 45, 65, 30, 45 and 65 respectively. If is a matrix with (4 by 2) dimension, then it would appear as follows after the above initialization statement is processed:}
\]

\[
\begin{array}{cc}
45 & 30 \\
30 & 65 \\
65 & 45 \\
0 & 0 \\
\end{array}
\]

The elements of the last row are left unchanged: all the
uninitialized variables are set to zeroes by DARRIE.

The initialization statement:

\[
\begin{array}{c}
45 \\
45 \\
45 \\
45
\end{array}
\]

sets the above matrix to:

\[
\begin{array}{ccc}
45 & 45 \\
45 & 45 \\
45 & 45 \\
45 & 45
\end{array}
\]

The initialization statement:

\[
\begin{array}{c}
0 \\
80 \\
0 \\
0
\end{array}
\]

sets the above matrix to:

\[
\begin{array}{ccc}
0 & 0 \\
80 & 65 \\
0 & 0 \\
0 & 0
\end{array}
\]

The position at which the initialization statements must appear and the method by which they are processed are in accordance to those of simple variables initialization statements.
3.4.1.8 Heterogeneous n-Dimensional Array (\(\text{\textbar{\textcheckmark}}\))

The heterogeneity implies that not all of the array elements have the same attribute and so a heterogeneous array can be used in situations where data elements, although not unique, are in close relationship to each other. \(\text{\textbar{\textcheckmark}}\) offers structures (\(\text{\textbar{\textcheckmark}}\)) for the representation of hierarchical information.

![Figure 3.4.1.8.1 An Example of a Structure.](image)

Figure 3.4.1.8.1 is an example of a multidimensional structure within \(\text{\textbar{\textcheckmark}}\). The structure shown is composed of a single linear array named \(\text{\textbar{\textcheckmark}}\) with three lower level structures \(\text{\textbar{\textcheckmark}}\), and \(\text{\textbar{\textcheckmark}}\) (each of which is itself another heterogeneous array) and one elementary data
item (the element of structure which is not itself another heterogeneous array): Figure 3.4.1.8.2 is the tree representation of the same example.

Figure 3.4.1.8.2  The Tree Representation of the Structure in Figure 3.4.1.7.2.2.

provides facilities to organize a set of heterogeneous n-dimensional array with the same constructs into an array of structures. Figure 3.4.1.8.3 shows an array of heterogeneous n-dimensional arrays, composed of n structures.

Figure 3.4.1.8.3  An Array of Structures.
The arrays of structures can have either fixed size-length or variable-size length (allowing for the number of structures of which the array is composed to increase or decrease dynamically during program execution). The size of the elementary data items also can be either fixed or variable. Thus, there are two categories of heterogeneous n-dimensional arrays: fixed-size which implies all the elementary data items are fixed in size and variable-size which implies one or more of the elementary data items is/are variable in size.

The arrays of structures also fall into two categories: fixed-size length and variable-size length.
The storage representations of heterogeneous n-dimensional arrays depend on whether the elementary data items are of fixed or varying size.

For structures with fixed-size elements a set of sequentially distributed simple variables are used, such as the storage representation of the example of Figure 3.4.1.8.3.

3.4.1.8.1 Heterogeneous Array Declaration

The flexibility in declaring different kinds of structures is shown by the general form of heterogeneous n-dimensional array declaration statement given in Figure 3.4.1.8.1.1.
Figure 3.4.1.8.1.1 The General Form of a Heterogeneous Array (⇒) Declaration.
Figure 3.4.1.8.1.2 The Storage Representation of a Structure Whose Elementary Data Items Are Fixed in Size.

3.4.1.8.2 Heterogeneous N-Dimensional Array Descriptor

Figure 3.4.1.8.2.1 illustrates the general form of the descriptors for heterogeneous n-dimensional arrays with fixed-size elements.

Where lower and upper bounds on the subscript (1 and u) indicate the size.
Figure 3.4.1.8.2.1 The General Form of the Descriptors for Heterogeneous n-Dimensional Arrays.

Where the descriptors for the lower level substructures are other heterogeneous n-dimensional arrays with fixed size elementary data items descriptors, homogeneous n-dimensional arrays with fixed length and fixed size descriptors, or simple variable descriptors.

Figure 3.4.1.8.2.2 shows the complete set of descriptors for the example in Figure 3.4.1.8.1.
Figure 3.4.1.8.2.2  The Complete Set of Descriptors for the Example in Figure 3.4.1.8.1.
For the representation of structures with variable size elements or arrays of structures with variable-size length a set of sequentially distributed simple variables cannot be used. Instead, linked list storage and utilization of run-time descriptors manipulations are required.

If the element whose size is variable is a SIMPLE, CHARACTER variable, then the address field in its description points to a linked list which holds its contents. The element whose size is variable and is a homogeneous n-dimensional array with variable-size length is treated the same as a homogeneous n-dimensional array with variable-size length.

To illustrate the above tactics the example in Figure 3.4.1.8.1 is modified as shown in Figure 3.4.1.8.2.3.

* The element can be of variable size if and only if it belongs to one of the two following categories:
  (1) simple variable and CHARACTER ( ), or
  (2) homogeneous n-dimensional array with variable-size length.
Figure 3.4.1.8.2.3  The Modified Version of the Structure in Figure 3.4.1.8.1.

The complete set of descriptors for the above structure is given in Figure 3.4.1.8.2.4.
Figure 3.4.1.8.2.4  The Complete Set of Descriptors for the Structure in Figure 3.4.1.8.2.3.
3.4.1.8.3 Heterogeneous Array Accessing

All segments of a structure can be referenced when the identifiers representing all segments of a structure are included within a program statement. For example, \[ \text{6.304:} \] represents the entire structure given in Figure 3.4.1.8.1. \[ \text{6.304:} \] represents both \[ \text{6.304:} \] and \[ \text{6.304:} \] elementary data items. \[ \text{6.304:} \] represents a single elementary data item, which belongs to substructure \[ \text{6.304:} \] which itself belongs to structure \[ \text{6.304:} \]. Figure 3.4.1.8.1 represents each identifier and the elementary data items or substructures of which it is composed.

The identifiers which represent substructures or elementary items may be qualified as in the following examples.

\[ \text{6.304:} \]
\[ \text{6.304:} \]
\[ \text{6.304:} \]

The above qualifications are not essential and serve as clarification for documental purposes. However, when there are more than one elementary data items or substructure with exactly the same identifier, the references to those variables must be qualified.
In the following structure:

two elementary data items are called \textit{two elementary data items are called}. To reference the data items without ambiguity, the references must be qualified.

\begin{itemize}
\item \textit{refers to} (the elementary item that is marked by a \textit{+}) and
\item \textit{refers to} (the elementary item that is marked by an \textit{*}).
\end{itemize}

In order to reference a structure of a homogeneous array of structures, the same structures and referencing methods discussed for referencing arrays hold.
The General Form of the Initialization of Structures.

Figure 3.4.1.8.4.1

Illustrates the General Form of Heterogeneous Array Initialization.
The list of initializing values for the elementary items of the array of (structure) ...

{ initializing values for the elementary items
  : list of subscript
  : array of structure identifier

 initializing values for the elementary items
  : list of subscripts which indicate a segment of a multi-dimensional array of structures
  : array of structure identifier

Figure 3.4.1.8.4.2 The General Form of the Initialization of Arrays of Structures.
3.4.2 Program Structure

Within (\text{\textipa{DARRIE}}), a program (an algorithm expressed in \text{\textipa{DARRIE}}) is the basic unit of computation which is not linguistically dependent upon previous executions, nor it can influence succeeding program executions (unless its output is stored on an external storage device and is used as its input on the succeeding run). It is composed of a set of recursively nested blocks with or without internal/external procedures and overlays (modules) each of which is composed of program statements.

The main program and external procedures are independent compilation units. Thus they can be compiled independently allowing the elimination of recompilation of the entire program where there is a need for modification of a subset of the modules. With the compile-time module integration facilities provided in \text{\textipa{PEESHTAZ}}, the groups of separately compiled modules are integrated into one module before the run.

Conditional compilation (compiling or ignoring certain text within the body of \text{\textipa{DARRIE}} programs upon the satisfaction or failure of certain conditions) is to be the subject of future investigation within \text{\textipa{DARRIE}}.
Figure 3.4.2.1 The Program Structure.
3.4.2.1 Program Execution

The program execution may occur when the following job control statement is submitted to PEESHTAZ:

```
[destination] [source]
```

The above statement causes the following activities to take place.

1. The command processor of PEESHTAZ transfers the control to processor;

2. processor translates the source program into its object form and if there are not any syntactical errors in the source program, it writes the object program onto a disk file. The listing of the source program is written on the specified destination if the parameter is specified;

3. the control is transferred to the loader/linkage editor which initiates the program execution; always the execution begins with the first executable statement of the main program and ends with an abnormal condition or with the execution of a statement,

4. when the program execution terminates, the control is transferred to the control mode of PEESHTAZ to process the next user’s request.
3.4.2.2 Main Program/Modules Communication

In the occasions in which a unique set of program statements may be required at several points within the program (Figure 3.4.2.2.1 illustrates such an occasion) procedures are used.

Figure 3.4.2.2.1 A Program Which Requires a Unique Set of Statements at Several Points.
3.4.2.3 Insertion of a Set of Instructions Within a Program

Figure 3.4.2.3.1 shows two methods by which the program can have access to the required set of instructions.

Figure 3.4.2.3.1 The Methods by Which a Set of Instructions Can be Inserted at Different Points Within a Program.

Figure 3.4.2.3.2 illustrates the usage of method 1.
Figure 3.4.2.3.2 One Method of Having Access to a Unique Set of Statements at Several Points in a Program - Method 1.

The program that uses method 1 (the insertion of the unique set of instructions at the points where the unique set is needed) does not take advantage of the uniqueness of the set and requires much more storage than the program which uses method 2. Method 2 requires less storage but a mechanism to transfer the control to the first statement of the unique set of statements and to return the control
to the next statement in sequence following the statement in the calling program at which the control was transferred is required. The mechanism requires at least two extra jump instructions for each call to the unique set of statements: one for transferring the control to the unique set of statements and another one to return the control to the statement following the statement in the main program at which the control was transferred and of course a memory location to hold the address of the return point.

DARRIE offers facilities for using both methods.
The order in which statements appear in a program is unimportant. The request for insertion of the set of statements can appear at any point within the bodies of the main program and the procedures. The flexibility in placing the set of instructions not always at the beginning of the source program may require an extra pass over the same program during compilation.

To use method 2, offers the internal/external procedures and overlays.

3.4.2.4 Procedure Structure

Procedures allow the user to use method 2 of accessing a set of instructions and/or when a large task is divided into several subtasks for either taking advantage from their simplicity in designing and debugging small subtasks or assigning different programmers to work on different subtasks. also allows recursive calls: calls to procedures that call themselves (although this effect can be obtained in other ways and no algorithm has to be expressed recursively in because of the existence of effectively infinite-sized arrays through the use of ALLOCATE and RELEASE statements). The external procedures can be translated by processor separately (they do not have to accompany main programs for translation). Although, they
are units of modularity and translation but are not units of executions, they cannot run without a main program.

3.4.2.4.1 Procedure/Main Program Information Exchange

The variables within the main program and external procedures are unknown to each other (to an internal procedure, the variables declared in the containing block are known but, the variables declared in the internal procedure are not known to the containing or any other block unless explicitly specified) unless communicated to each other by one of the following methods:

1. the use of PUBLIC (attribute,
2. the use of SEMI-PUBLIC (attribute), and/or
3. the use of procedure arguments list.

When a variable is declared as PUBLIC (attribute) in the main program or any of the procedures within a program, it is known to all the compromising modules (except the ones which declare the variable with the same identifier).

A variable declared as SEMI-PUBLIC (attribute) in one module is known to another if and only if the same variable is declared as SEMI-PUBLIC (attribute) in it.
Figure 3.4.2.4.1.1 An Example of Programs with Their External Procedures.

In program 1 above, variable \( v \) in the main program and its two external procedures represent the same object. In program 2 variable \( v \) in the main program and procedure 2 refer to the same object but variable \( v \) in procedure 1 refers to a different object from the one in the main program and procedure 2. In program 3, variable \( v \) in the main program, the block contained in the main program, and procedure 2 refer to the same object.
In program 4 variable \( m \) is known to the main program, procedure 1 and 2 but it is different from variable \( m \) in the block contained in the main program, since that block possesses its own declaration for variable \( m \).
3.4.2.4.1.1 Arguments/Formal Parameters Dynamic Relationships

Where a procedure is invoked arguments are associated with the formal parameters in the formal parameter part of a procedure. There are three traditional methods by which the associations between the arguments and formal parameters are established: Call by name, call by value, and call by reference.

3.4.2.4.1.1.1 Call by Name

If a formal parameter is declared as name, at the invocation moment, all of its occurrences within the procedure are replaced by the corresponding argument. The procedure with two formal parameters and which are assumed to be declared as name is given below.

\[
\text{\texttt{t}} + \text{\texttt{r}} - \text{\texttt{1}}.
\]

The action of is to increment (global variable defined in the containing block) by 1 and then set the variable to the value of plus 100.

If the procedure is invoked by call, the effect is as if the following text is inserted at the point of invocation and executed.
The usual method of implementing such calls is to distinguish between the arguments which are single identifiers and those which are expressions. In the case of single-identifier arguments, the addresses of the identifiers are passed to the called procedure. In the cases of arguments which are expressions, the address of the routine to compute them is passed.

3.4.2.4.1.1.2 Call by Value

In call by value, all the formal parameters declared as value are assigned the values of the corresponding arguments before entering the procedure body.

If the formal parameters of the procedure are assumed to be declared as value, then invokes the procedure with the following results achieved.

\[ \text{contents of global variable is incremented} \]

A formal parameter declared as value cannot appear on the left-hand side of an assignment statement, since the parameter represents a constant but not a variable. In programming languages, such as in FORTRAN, it is possible to change the values of constant due to the lack
of imposing the above restriction: a formal parameter declared as value cannot appear on the left-hand side of an assignment statement.

3.4.2.4.1.1.3 Call by Reference

This method, also called by address, passes a reference (address) for each argument. When the arguments are simple variables, the references passed to the called procedure are the machine addresses of the variables. So the called procedure has access to the contents of the variable and since the procedure knows the address of the variable it can modify the variable contents.

When the arguments are arrays, then the address of the subscripted arguments are evaluated at the point of call and passed to the procedure.

When the arguments are expressions there are at least two possible interpretations (with different effects):

(1) computing the expression at the procedure call, storing it in a memory location, and passing the address of the location to the called procedure, and

(2) sending a reference to a routine which evaluate the expression.

Method (1) produces the effect of call by value with the exception that the argument can appear on the
left-hand side of an assignment operator. Method (2) reproduces the effect of call by name.

3.4.2.4.1.1.4 Call by Name-and-Value

\[ \text{DARRIE} \] combines the different methods of establishing arguments/formal parameters dynamic relationship to allow powerful means of information exchange among main programs and procedures.

When the argument is a simple variable, the address of the argument is passed to the called procedure. Since, a simple variable in \[ \text{DARRIE} \] can be of procedure type, an argument can be a reference to a procedure and thus can produce the effect of call by name.

When the argument is a constant, the corresponding formal parameter is set to that value and special care is taken to not allow that formal parameter to appear on the left-hand side of any assignment operators (the undesirable feature of FORTRAN which can lead to changing the values of constants is not available in \[ \text{DARRIE} \]).

When the argument is an expression or subscripted variable, two options may be desired by the user:

(1) evaluating the expression, placing the result in a memory location, and then passing the address of the location in which the result
is held; determining the address of the subscripted variable and passing the address, or

(2) reproducing the effect of call by name.

Option 1 is assumed by \( \langle S \rangle \) if the corresponding formal parameters are of simple type.

Option 2 is assumed by \( \langle S \rangle \) if the corresponding formal parameters are of procedure type and/or the arguments are enclosed in less than and greater than operators (\( \langle, \rangle \)).
Figure 3.4.2.4.1.1.4.1 illustrates the general form of a procedure and its declaration definition.
3.4.3 Arithmetics

DARRIE provides capabilities to process expressions composed of the basic arithmetic operations and operands.

3.4.3.1 Operators

Figure 3.4.3.1.1 illustrates the repertoire of arithmetic operations and their corresponding operators (the symbolic representations of arithmetic operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
<th>Infix</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>subtraction</td>
<td>-</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>multiplication</td>
<td>×</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>division</td>
<td>÷</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>exponentiation</td>
<td>‡</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>assignment</td>
<td>+</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>plus</td>
<td>+</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>negation</td>
<td>−</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.4.3.1.1 The Repertoire of Arithmetic Operations.
The infix operators require two operands (quantities manipulated in an expression), such as in the following examples:

\[ \begin{align*}
+ & \quad - \\
* & \quad / \\
\Rightarrow & \quad \Leftrightarrow \\
\Rightarrow & \quad \Leftrightarrow \\
\Rightarrow & \quad \Leftrightarrow \\
\Rightarrow & \quad \Leftrightarrow
\end{align*} \]

The assignment symbol, \( \Rightarrow \), states that the variable on its left-hand side is defined by the expression (may be a single variable) on its right-hand side, i.e., the value of the expression or the single variable on the right-hand side is placed in the variable on the left-hand side. The assignment statement differs from an equation which defines an equality relationship. For example, the assignment statement

\[ \text{in } \text{DARRIE} \]

means that the value of \( \Rightarrow \) is incremented by 1 but it is an unsatisfied equation in mathematics.

The prefix operator negation (\( \Rightarrow \)) is used to specify the negative value of the operand which follows it. The operator plus (\( + \)) represents the true value of an operand and its omission also implies, as in algebra, the same thing: the true value of the operand.
3.4.3.2 Operands

Operands are the quantities that are manipulated in arithmetic expressions. All categories of variables (SIMPLE/ متوازنة، HOMOGENEOUS ARRAY/ متوازنة، and HETROGENOUS ARRAY/ متوازنة) , constants, and functions may represent operands of arithmetic expressions. The types of the variables and constants which represent operands are restricted to INTEGER/ صحيح، REAL/ صحيح، and COMPLEX/ صحيح, in any desired base. In case of a simple assignment statement, when there is only one variable or constant, the restriction is relaxed to include all of the other types of data (LOGICAL/ صحيح، CHARACTER/ صحيح، LABEL/ صحيح، POINTER/ صحيح، and PROCEDURE/ صحيح).

3.4.3.3 Constants

Constants (literals), within دارج، refer to the values which cannot be changed in the course of program execution (unlike the values of variables). Constants do not reside in variables (storage locations), thus they can never appear on the left-hand side of an assignment operator. They can represent the values of all data types: INTEGER/ صحيح، REAL/ صحيح، COMPLEX/ صحيح، LOGICAL/ صحيح، CHARACTER/ صحيح، POINTER/ صحيح، LABEL/ صحيح، and PROCEDURE/ صحيح. The constants
like the variables can be either numeric (INTEGER, REAL, and COMPLEX) or non-numeric (LOGICAL, CHARACTER, POINTER, LABEL, and PROCEDURE).

The numeric constants may represent values in different bases. The base of a numeric constant is specified by enclosing the base in brackets and placing it immediately to its right. For example, \( 1[2] \) represents decimal number 2 in base 2 and \( 187[7] \) represents decimal number 237 in base 7. The omission of the base specification results in the assumption that the constant belongs to the decimal number system.
3.4.3.3.1 INTEGER/ Constants

INTEGER constants belong to a finite set of natural numbers, the largest of which depends upon N (the maximum number of digits allowable by the implementation). INTEGER constants may represent negative numbers if they are preceded by the negation (−) sign. Figure 3.4.3.3.1.1 defines an INTEGER constant.

```
<digit> ::= - | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<non-zero digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<base> ::= <non-zero digit><base><digit>
;base designator> ::= <empty><base>
<unsigned integer> ::= <digit><unsigned integer><digit>
<unsigned integer with base designation> ::= <unsigned integer><base designator>
<INTEGER constant> ::= <unsigned integer with base designation>:+<unsigned integer with base designation>−<unsigned integer with base designation>
```

Figure 3.4.3.3.1.1 The Definition of INTEGER/ Constants.
3.4.3.3.2 REAL Constants

REAL constants represent the finite rational approximations for irrational quantities. They have wider range of assumable values than INTEGER constants do. The REAL constants, like REAL variables, can be either FIXED-POINT or FLOATING-POINT.

3.4.3.3.2.1 REAL FIXED POINT

requires an extra character, decimal point* to distinguish REAL FIXED-POINT constants. For example, trr represents 1023.57 and represents 0.5002.

3.4.3.3.2.2 REAL FLOATING POINT

To represent REAL FLOATING POINT constants a method analogous to the "scientific notation" which is often used in dealing with very large and very small numbers is used. For example, 3.456 * 10^20 is written instead of a 21-digit number. The method allows the representation of a wide range of constants with a reasonable amount of accuracy and eliminates problems of shifting, as required in the operations with operands represented in fixed floating point notation.

*Symbol, /, is the decimal point equivalent in Farsi arithmetics.
There are different schemes for the internal representation of the floating point constants; the following is a fairly typical method used in CDC 3300.

A non-zero constant $x$ is represented in floating point by finding a fraction $f$ and an integer $P$ such that $X = f \times 2^P$, and $1/2 < |f| < 1$. If the following storage structure is designed for a floating point variable:

```
| sign (1 bit) | biased exponent (11 bits) | fraction (36 bits) |
```

Then only those constants with $-1023 < p < +1023$ can be represented and $p < -1023$ or $p > +1023$ cause UNDERFLOW or OVERFLOW conditions, respectively. This means that $x$ must be in the (approximate) range:

$$10^{-308} < |x| < 10^{308}.$$  

The problem of having the appropriate signs for both $f$ and $p$ (which do require sign bits) is circumvented by representing $f$ in usual one's complement notation, and by "biasing" $p$. If $p$ is negative, it is biased by adding $1777_8$ to it, if it is non-negative, the bias is $2000_8$. 
432
True Exponent
(Octal)

True Exponent
(Decimal)

Biased Exponent
(Octal)

+1023

+1777

3777

+1022

+1776

3776

+2

+0002

2002

+1

+0001

2001

+0

+0000

2000

-0

-0000

1777

-1

-0001

1776

-1022

-1776

0001

-1023

-1777

0000

The biasing also eliminates the need for special
checks in processing the expressions that involve relational operators.

For example, the unbiased representa-

tion of 0.0528.
0 11

111

111

101

equivalent to:

101 010 000 000 000 000 000 000 000 000 000 000
37755200000000008

appears to be greater than the unbiased representation of
52

8:

0 00 000 000 010 101 010 000 000 000 000 000 000 000 000 000 000
equivalent to:

00025200000000008, but if the exponents are

biased, the simple machine determination agrees with the
true order relation.


Thus biasing the exponent eliminates the need for both a separate sign bit for the exponent and special checks when relational operators whose operands are of real floating point type are executed. The sign bit in this representation is set to 0 for positive constants and set to 1 for negative constants since in setting the sign of a floating point number, all the bits (48 of them in this case) are complemented.

The following are some examples of floating point constants using the preceding scheme.

\[+1 = .1_2 \times 2^1 = .100_2 \times 2^1 = .4_8 \times 2^1 = 200140000000000_8\]
\[+2.375 = 10.011_2 = .10011_2 \times 2^2 = .100110_2 \times 2^2 = .46_8 \times 2^2 = 200246000000000_8\]
\[+.1875 = .0011_2 = .110_2 \times 2^{-2} = .6_8 \times 2^{-2} = 177560000000000_8\]
\[-1 = 57763777777777778_8\]
\[-2.375 = 57753177777777778_8\]
\[-.1875 = 60021777777777778_8\]
\[.5 = 200040000000000_8\]
\[.25 = 177640000000000_8\]
\[0 = 000000000000000_8\]
REAL FLOATING POINT constants have the following general form:

\[
\left\{ \begin{array}{c}
[-] \\
[+] \\
\end{array} \right. \text{fraction exponent}
\]

For example \( -1.5 \times 10^{3} \) represents \(-1.5 \ldots\) or \( 1.5 \times 10^{-3} \) represents \( \ldots1.5 \ldots \).

Figure 3.4.3.3.2.2.1 defines REAL constants within DARRIE.

\[<\text{part a}> ::= \langle\text{empty}\rangle | <\text{unsigned integer}>\]
\[<\text{part b}> ::= <\text{part a}> | <\text{part a}> + <\text{part a}> | -<\text{part a}>\]
\[<\text{REAL FIXED POINT constant}> ::= <\text{part b}> | <\text{unsigned integer with base designation}>\]
\[<\text{exponent}> ::= <\text{unsigned integer}> | +<\text{unsigned integer}> | -<\text{unsigned integer}>\]
\[<\text{REAL FLOATING POINT constant}> ::= <\text{REAL FIXED POINT constant}> | <\text{exponent}>\]

Figure 3.4.3.3.2.2.1 The Definition of REAL/\(5,\ldots\) Constants.
3.4.3.3.3 COMPLEX/ Constants

The COMPLEX constants have the following general form:

\[
\begin{align*}
\{ & \{ \text{INTEGER} \} \} : \{ \{ \text{REAL} \} \} \\
\{ & \{ - \} \} \{ \text{REAL} \} : \{ \{ \text{REAL} \} \}
\end{align*}
\]

For example, \( 1.0 \cdot 0 + \pi \) and \( 1.0 \cdot 1 + 1.0 \) represent two COMPLEX constants in decimal and binary number systems. The base designator is specified with the real part of COMPLEX constants and the imaginary part must be represented in the specified base.
3.4.3.3.4 LOGICAL Constants

The LOGICAL constants within are: TRUE and FALSE.

LOGICAL Constants ::= TRUE | FALSE

3.4.3.3.5 CHARACTER Constants

The CHARACTER Constants within are composed of one or any number of characters set enclosed in double quotes.

<Character>::= one of the characters of character set.

<string of characters>::= <character> | <string of characters> <character>

<CHARACTER constant>::= "<string of characters>"
3.4.3.3.6 LABEL/ AND PROCEDURE/ Constants

The LABEL and PROCEDURE constants are the same as CHARACTER constants. DARRIE can determine from the context of the program statement (in which a LABEL constant is used) or from the attribute of the target variable in assignment statements whether the string of characters enclosed in double-quotes is a LABEL or CHARACTER constant.

3.4.3.3.7 POINTER/ Constant

The POINTER constant is the same as an unsigned integer. Figure 3.4.3.3.1.1.
### Library Functions

Besides the user defined functions, there are a number of library functions (system defined) that the users can include in the arithmetic statements without explicitly defining them.

<table>
<thead>
<tr>
<th>Function Designator</th>
<th>Arguments</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIN/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COS/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TAN/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARCSIN/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARCCOS/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARCTAN/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINH/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COSH/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TANH/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARCSINH/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARCCOSH/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ARCTANH/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LN/</td>
<td></td>
<td>$\log_e$</td>
</tr>
<tr>
<td>EXP/</td>
<td></td>
<td>$e^x$</td>
</tr>
<tr>
<td>MAX/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MIN/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABS/</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SIGN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RECIPROCAL/</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
FACTORIAL

if \( x \leq 2 \) then \( \text{other-wise FACTORIAL} \)

MODULUS

integer R, \( \alpha \leq R < \beta \)
integer q \( \alpha \leq q < \beta + R \)

FLOOR

largest integer \( \alpha \leq m < \beta \)

CEILING

smallest integer \( \alpha \leq m < \beta \)

TRUNC

concatenates the arguments \( \alpha \) and \( \beta \).

REAL FIXED


REAL FLOATING


LENGTH


CONCATENATE


INSERT


EXTRACT


DELETE


REPLACE


TIME

current clock time as:
XX : XX : XX
hour minute second

DATE

current date as:
XX : XX : XXXX
month day year
MFBLKS/

total number of memory file blocks

TRAFFIC/

total number of current users
3.4.3.4 General Form and Definition of Arithmetic Statement

Figure 3.4.3.4.1 The General Form and Definition of Arithmetic Expressions.

```
[ROUNDED][in case of SIZE ERROR imperative statements]

{identifier
{identifier,...,identifier}
- [operand [signed operand] [operator] [operand [signed operand]]

<round up clause> ::= <empty>
<add or subtract> ::= +-
<multiply or divide> ::= *%
<primary> ::= <constant>|<variable>|<function designator> | ( arithmetic expression)
3.4.3.4.1 ROUNDED/Clause

ROUNDED/Clause is used to ensure that the final result assigned to the target is rounded; otherwise, the result may be truncated.

3.4.3.4.2 SIZE ERROR/Clause

SIZE ERROR/Clause is used to execute a set of imperative statements when the result of an arithmetic statement cannot be placed in the target variable without loss of significant digits due to the precision attribute of the target variable. When SIZE ERROR occurs and SIZE ERROR clause is not present, unpredictable results may be obtained.

3.4.3.4.3 OVERFLOW/UNDERFLOW/Clause

SIZE ERROR/Clause differs from OVERFLOW/UNDERFLOW clauses. Overflow and underflow conditions can occur when the storage requirement for representing the result of an arithmetic operation exceeds the amount of storage used to represent the real or integer variables, either because the result is too big (overflow condition) or it is too small (underflow condition). The OVERFLOW/UNDERFLOW clauses can be used to direct the control to a set of imperative statement when overflow or underflow occurs; otherwise prints an error
message and terminates the execution, abnormally.

The general form of the statement is:

\[
\ldots \begin{cases} \text{OVERFLOW/UNDERFLOW statement} \\ \text{other non-executable statements} \end{cases}
\]

OVERFLOW/UNDERFLOW statements like other non-executable statements must appear before any executable statements within a block. The option may be turned off at any point after specified by the following statement.

\[
\ldots \begin{cases} \text{specified statement} \\ \text{other non-executable statements} \end{cases}
\]

The option may be turned back on at any point (after it had been turned off or before it has ever been turned on) by the general form of the statement.
<table>
<thead>
<tr>
<th>Operation</th>
<th>First Operand Precision</th>
<th>Second Operand Precision</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition/subtraction</td>
<td>((P_1, Q_1))</td>
<td>((P_2, Q_2))</td>
<td>(P_r = \text{MIN}(N, \text{MAX}(P_1 - Q_1, P_2 - Q_2) + \text{MAX}(Q_1, Q_2) + 1))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Q_r = \text{MAX}(Q_1, Q_2))</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>((P_1, Q_1))</td>
<td>((P_2, Q_2))</td>
<td>(P_r = \text{MIN}(N, P_1 + P_2 + 1))</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(Q_r = Q_1 + Q_2)</td>
</tr>
</tbody>
</table>

Figure 3.4.3.4.1 The Precision of the Fixed-Point/Results of Arithmetic Operations.
In the above, P's are the number of digits provided for the result, Q's are the scale factors (the number of digits to the right of the decimal point), and r, 1, and 2 subscripts refer to the result, operand 1 and operand 2 respectively. N is the maximum number of digits allowed by the implementation. MIN (m,n) and MAX (m,n) refer to the smaller and the larger of the two integers m and n respectively.

The precision of a REAL FIXED-POINT result is the greater of the precisions of the two operands.

3.4.3.4.5 Generic Operations

(3/) allows generic operations (the operations in DARRIE which the operands of an operator can be of different types).

<table>
<thead>
<tr>
<th>Operator b</th>
<th>INTEGER</th>
<th>REAL FIXED</th>
<th>REAL FLOATING</th>
<th>COMPLEX</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>INTEGER</td>
<td>REAL FIXED</td>
<td>REAL FLOATING</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>REAL FIXED</td>
<td>REAL FIXED</td>
<td>REAL FIXED</td>
<td>REAL FLOATING</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>REAL FLOATING</td>
<td>REAL FLOATING</td>
<td>REAL FLOATING</td>
<td>REAL FLOATING</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
<td>COMPLEX</td>
</tr>
</tbody>
</table>

Figure 3.4.3.4.5.1 Type Dominancy Within Generic Arithmetic Operations.
Figure 3.4.3.4.5.1 illustrates that the following type attribute dominancy exists within the generic operations.

```
<table>
<thead>
<tr>
<th>level of dominancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPLEX</td>
</tr>
<tr>
<td>REAL FLOATING</td>
</tr>
<tr>
<td>REAL FIXED</td>
</tr>
<tr>
<td>INTEGER</td>
</tr>
</tbody>
</table>
```

Obviously the capabilities that allow generic operations impose testings for the types of operands and invokations of conversion routines during the execution. DARRIE also provides the following functions for converting the value of a variable from one type to another.

### 3.4.3.4.6 Type Attribute Conversion Functions

- **FLOOR/**
- **CEILING/**

The above two functions return integer values of their arguments. **FLOOR/** returns the largest integer I such that I ≤ arg, while **CEILING/** returns the smallest integer I such that arg ≤ I.

- **REAL-FIXED**
- **REAL-FLOATING**

The above two functions return REAL FIXED and REAL FLOATING values of their arguments.
COMPLEX/

The above function returns COMPLEX value of its argument (it attaches an imaginary part, whose value is zero, to the argument, if the argument is not already a complex variable or constant).

The argument type of the type-attribute-conversion functions can be INTEGER, REAL, and COMPLEX. When functions FLOOR, CEILING, REAL-FIXED, and REAL-FLOATING are invoked and the argument is either a COMPLEX variable or constant, a second argument (REAL or IMAGINARY) is included to specify whether the action must take place on the real or on the imaginary part of the complex variable or constant.

The same effects as calling the conversion type-attribute-conversion functions can be obtained by using assignment statements whose target variables' attributes are different from the attributes of the variables on the right-hand side of the assignment operator. For example if the type attribute of the target variable is INTEGER and the value of the real variable or the constant on the right-hand side is 13.67, then the integer value 13 (FLOOR (13.67)) is assigned to the target variable. Or if the attribute of the target variable is COMPLEX and the value of the integer variable (or the constant) on the right-hand side is 24, then the complex value (24 + 0i) is assigned to the target variable. The following tables resolve ambiguities among the assignment statements whose target and right-hand side variable attributes are of POINTER, LABEL and PROCEDURE types.
<table>
<thead>
<tr>
<th>Target Variable Attribute</th>
<th>Right-hand side Variable Attribute</th>
<th>Value of the Target Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>POINTER</td>
<td>POINTER</td>
<td>the same as the address represented by the right-hand side variable</td>
</tr>
<tr>
<td>POINTER</td>
<td>NON-POINTER</td>
<td>the address of the right hand side variable</td>
</tr>
<tr>
<td>NON-POINTER</td>
<td>POINTER</td>
<td>the contents of the variable pointed by the right-hand side variable</td>
</tr>
<tr>
<td>LABEL/PROCEDURE</td>
<td>LABEL/PROCEDURE</td>
<td>the same LABEL/PROCEDURE represented by the right-hand side variable</td>
</tr>
<tr>
<td>LABEL/PROCEDURE</td>
<td>NON-LABEL/NON-PROCEDURE</td>
<td>the contents of the variable on the right-hand side variable</td>
</tr>
<tr>
<td>NON-LABEL/ NON-PROCEDURE</td>
<td>LABEL/PROCEDURE</td>
<td>the same LABEL/PROCEDURE represented by the right-hand side variable</td>
</tr>
</tbody>
</table>
3.4.3.4.7 Operator Precedence

The definition of the arithmetic expressions, also describes the specification of operator precedence within \((\text{DARRIE})\). Figure 3.4.3.4.7.1 shows the operator precedence (applicable with minor modifications to most high-level languages).

![Operator Precedence Diagram]

Figure 3.4.3.4.7.1 The Operator Precedence.

To remove possible ambiguity among the expressions involving sequences of equal precedence operators, left-to-right associativity direction is assumed.
3.4.4 Input/Output

The assignment and initialization statements can be used to associate values to the variables of an algorithm expressed in $\text{(5-\text{DARRIE})}$. However, if the values of the variables change from problem to problem and/or the algorithm processes a large set of values, the following difficulties arise:

1. The program must be altered (the assignment and initialization statements within the program have to be physically changed) for each problem, and/or
2. The processing of a large set of values becomes extremely tedious, because of the inclusion of a large set of assignment or initialization statements, in addition to the above difficulty (1).

The above difficulties are circumvented by distinguishing program data from $\text{(5\text{-DARRIE})}$ statements. The data to be processed may be, physically not contained within the program but, kept on an external storage device and means of value acquisition be provided.

Also, certain problems impose the necessity of retaining data between program executions which introduce the linguistic need for deposition of data on an external device.
The capabilities of data acquisition and disposition statements within C5, exceeds the mechanism to transfer data in and out of computer. They are capable of reading and/or writing program statements within the program in the course of execution. This allows the possibility of dynamic program modification, the feature that may be of significant value in an artificial intelligence environment in which the components do not operate merely upon explicit learning.

3.4.4.1 Acquisition

The means of assignment of values to variables where the values come from an external (input) source or a segment of memory is provided by DARRIE statement, illustrated in Figure 3.4.4.1.1.
Figure 3.4.4.1.1 The General Form of Statement.
The general form of \( \text{L.y}_9 \) statement in \( \text{DARRIE} \) illustrates the following capabilities:

* list-directed value acquisition
  - the transfer of information from terminal,
  - the transfer of information from files (sequential or random access), and

* data-directed value acquisition
  - the transfer of information from a segment of memory (data or any other program statement).

Through the set of available options, mechanisms to perform the following activities (in addition to the information transfer) exist:

* the transfer of control to a specific statement, a procedure, or a block when an end-of-file mark or an I/O error has occurred.
* the manipulation of the input file prior to or after the completion of an inputting process,
* the transfer of formatted or unformatted data, and
* the transfer of self-identifying data, in which the data stream contains not only the value represented but also the names of the target variables. \( \text{DARRIE} \) allows the flexibility of the omission of the entire list of variables in the statement. This provides means of dynamic transfer of values into any of the program variables with a single statement. Example 2 in 3.4.4.1.4.
3.4.4.1.1 Clause

This clause is used for formatted transfer of information from the stream of data into variables.

The clause includes a format string immediately following it or references a statement whose contents form a format string. The reference is made either by a statement label or an array identifier in which the format string is held and thus can be changed dynamically. The format string immediately following the clause or stored in an array is enclosed in parenthesis. The format string which appears as a program statement and is referenced by its label must start with and any other descriptive phase following it before the right parenthesis of the format string.

Example:

(...format string...) 

3.4.4.1.2 Format String for Data Acquisition

A format string contains information about the detail of data transfer: the locations and the lengths of the fields from which the data must be obtained and the types of conversion processes required to convert the raw data to the appropriate internal form.

The following are the elements of which format strings
for data acquisition are constructed.

\[ (1) \quad \text{(base)} \quad \frac{1}{l+f} \]

The element (1) indicates that an integer constant whose length (total number of digits of which the constant is composed and a position for its sign) is 1 is to be read. If the constant belongs to a number system other than decimal, its base must be indicated immediately following the base indicator: \( \frac{1}{l+f} \). The target variable must be an integer.

\[ \left\{ \frac{1}{l+f} \right\} \quad \text{(base)} \quad \frac{1}{l+f} \]

The element (2) specifies a real fixed-point constant whose length (total number of digits of which the constant is composed, a position for its sign, and a position for the decimal point,/) is 1 and its precision is \( f \) is to be read. If the decimal point (or similar to that in other bases) is present in the constant it must be denoted by / otherwise the position at which the decimal point is to be assumed must be indicated by . If the constant belongs to a number system other than decimal, its base must be indicated immediately following the base indicator. The target variable must be a real fixed-point variable.

\[ \left\{ \frac{1}{l+f} \right\} \quad \text{(base)} \quad \frac{1}{l+f} \]

The element (3) indicates a real floating-point constant whose length (total number of digits of which the constant is composed, a position for the sign, a position
for the decimal point if it is present, and five positions for the exponent, \( \pm \xxx \) is 1 and its precision is \( q \) is to be read. If the decimal point is not present within the constant, the position at which it must be assumed is shown by \( \dagger \). The target variable must be a real floating point variable.

The element (4) indicates a string of \( \dagger \) characters whose length is 1 is to be read into a CHARACTER (\( \dagger \)) variable, a LABEL (\( \dagger \)) variable, a PROCEDURE (\( \dagger \)), or into a LOGICAL (\( \dagger \)) variable when the string is .. or ..

\[
(\{1:i\}) \quad [(\text{base}) \quad \dagger \] \quad (5)
\]

The element (5) indicates a complex constant whose length (total number of digits of which the constant is composed, a position for the sign, and a position for, if \( : \) is present within the constant) is 1. If \( : \) is not present within the constant, the position at which it must be assumed is shown by \( \dagger \). The target variable must be a complex variable.

\[
[(1)] \quad \dagger \quad (6)
\]

The element (6) indicates the field that must be skipped. The length of the field to be skipped is denoted by 1. If the length is not specified, the field with length one is assumed.
The element (7) indicates the source from which the information is transferred must be forward spaced \( n \) logical records. If the number of records to be forward paced is not specified, one record to be forward spaced is assumed.

The element (8) indicates the source must be back spaced \( n \) records. If \( n \) is not specified, one record is back spaced.

The element (9) rewinds the source.

The element (10) sets the read/write pointer at the end of the file.

The element (11) searches the source backward and places the read/write pointer past the \( n \)th end-of-file mark. If \( n \) is not specified, it is assumed to be one.

The element (12) searches the source forward and places the read/write pointer past the \( n \)th end-of-file mark. If \( n \) is not specified, it is assumed to be one.
3.4.4.1.3 Clause

This clause causes the control to be transferred (to the program statement whose label is specified, to the block immediately following it, or to a procedure whose name is specified) when an end-of-file mark is detected.

3.4.4.1.4 Clause

This clause sets the I/O Error (variable, global to all the programs, appropriately when one of the following faults is detected during the inputting process.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Fault</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>Input unit is not defined</td>
</tr>
<tr>
<td>2</td>
<td>Reading from a write-only file</td>
</tr>
<tr>
<td>3</td>
<td>Illegal character encountered</td>
</tr>
<tr>
<td>4</td>
<td>File at the end of data</td>
</tr>
<tr>
<td>5</td>
<td>Reading from an illegal device</td>
</tr>
</tbody>
</table>

Examples of statements:

(1)

Statement (1) reads three values entered from terminal and places them in simple variables and , respectively. The values are read from the terminal because the source is unspecified so the default source (terminal) is assumed. The values
are entered free of format because of the absence of option. The information entered from the keyboard may not be self-identifying as follows:

or may be self-identifying:

When the data is self-identifying, the order in which it appears in the data stream is unimportant.

If an end-of-file mark or an error occurs while statement (1) is being processed, the appropriate error message is printed and the program execution is terminated, because and/or clauses are not specified.

In statement (2), no clauses or even the list of variables is specified. In this case, the assumption that self-identifying data is entered from the terminal, is made. The omission of the entire list of variables provides flexibility in changing the contents of any program variables during the program execution with a single statement.

The following input places the values and in the variables and respectively.

The following input places the values and in the variables and.
Statement (1) rewinds logical unit identifier 11. forward spaces one record and then reads four values into the simple variables and and free of format. If an end-of-file mark is detected in the process of reading, the control is transferred to statement.

Statement (4) reads from logical unit identifier , into the variables and according to the format statement and transfers the control to the statement if any input error is detected. The given format state-
ment informs that each record of the input file is organized as:

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAxxxxxxNNxxxNN
```

where N's denote digits, x's denote spaces, and A's denote any character.

Statement (5) reads from logical unit identifier into the variables and according to the format which illustrates that the record layout is as follows:

```
NNNNNNNNxxxxNNNxxNNNNLNN
```

Statement (6) places the first ten constants entered from the terminal keyboard into the elements of array identified by subscripts 1 through 10 and the next ten symbols entered from the terminal keyboard into the elements of array identified by subscripts 10 through 20.

Statement in (7) places the first 17 constants entered from the terminal keyboard into all the elements of column 11 of table.

All the accessing features on arrays and structures
are available with statements.

Statement (8) places the first five strings in the data statement into those elements of array which are identified by subscript 1 through 5.

The next read from the above data statement would cause the sixth symbol to be transferred; to start with the first one symbol must be used.

3.4.4.2 File

File as defined in is the basic repository for storing data.

3.4.4.2.1 Opening

The required preliminary operations to make a file ready to be read from and/or written onto is accomplished by the following statement.
Figure 3.4.4.2.1.1 The General Form of Statement.

Examples:

Statement (1) opens the file stored under the name file 10 and associates it with logical unit 10. Because of the attribute the file is write-only (its contents cannot be read unless explicitly respecified within the block or procedure in which it was opened).

Statement (2) opens the file stored under and associates it with identifier 7. The file is read-only (its contents cannot be disturbed) because of the default, attribute.

Statement (3) opens a scratch random access file and associates it with logical unit 5.
3.4.4.2.2 Closing

Certain operations are necessary upon completion of data transmission from/onto a file. The necessary operations are accomplished by \( S \rightarrow \) when the program execution terminates or when it is explicitly specified by the following statement.

\[
\begin{align*}
\text{logical unit number}_n & \quad \text{logical unit number}_i & \quad \text{logical unit number} \\
\text{logical unit identifier}_n & \quad \text{logical unit identifier}_i & \quad \text{logical unit identifier}
\end{align*}
\]

Figure 3.4.4.2.2.1 The General Form of Statement.

3.4.4.2.3 Manipulation

All of the file manipulation commands in can be used as program statements within programs. To increase program readability, identifiers can be associated with files in addition to logical unit numbers.

\[
\text{Statement (1) rewinds the files associated with logical unit identifier } \text{id} \text{ and logical unit number } 2.
\]
3.4.4.3 Disposition

The means of transmission of messages and contents of program variables is provided by \textit{statement}, illustrated in Figure 3.4.4.3.1.
Figure 3.4.4.3.1 The General Form of Statement.
In formatted statements, when the destination is terminal the file manipulation symbols (PEESHTAZ, -, , , +) are ignored and the first string in the format string is considered as the indicator of the vertical position of the line.

<table>
<thead>
<tr>
<th>first character of the line</th>
<th>action taken before printing the line</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>clear the screen and start at the right top corner ( )</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>skip 1 line</td>
<td>skip 2 lines</td>
</tr>
<tr>
<td>skip 3 lines</td>
<td>skip 4 lines</td>
</tr>
<tr>
<td>skip 5 lines</td>
<td>skip 23 lines</td>
</tr>
</tbody>
</table>

Figure 3.4.4.3.2 The Carriage Controls.

The appearance of within the format string causes the following information to be printed on another line (its vertical position is indicated by the contents of the first following string).

Example 3

The clause can be used with the statements to display data on the screen of terminal according to the size of characters specified. The general form of this clause is:
where \( n \) indicates the factor by which the characters must be magnified. The normal size of characters (when \( n=1 \)) is

- 10 by 16 - dot matrix
- 0.2048 by 0.3277 centimeter
- 0.0806 by 0.1290 inch

Examples of statements:

Statement (1) prints the value of the variable on the screen of the terminal.

If the value stored in the variable is 15.9, statement (2) prints the following line of information after skipping 11 lines.

If the values stored in the variables are 15.9 and 20 respectively, then statement (3) prints the following lines of information after skipping one line.
3.4.5 Execution Control

3.4.5.1 Conditional Statements

The conditional statements are used to transfer the program control from one program statement to another one, conditionally.

Figure 3.4.5.1.3 illustrates the general form of the basic conditional statement within DARRIE. The statement is similar to the conventional statement IF-THEN-ELSE statement. It is composed of three parts: the decision making part where some "conditions" are checked, the program statements which are executed when the "conditions" are satisfied, and the program statements which are executed when the "conditions" are not satisfied.

In the above, "condition" and "Boolean expression" are defined as follows:

- <condition> ::= <arithmetic expression> <relational operator> <arithmetic expression>
- <Boolean primary> ::= <logical variable> | <logical function designator>
- <Boolean secondary> ::= <Boolean primary> | <Boolean secondary>
- <Boolean factor> ::= <Boolean secondary> | <Boolean factor> n <Boolean secondary>
<Boolean expression>::= <Boolean factor>|<Boolean term>u<Boolean factor>|<Boolean term@<Boolean factor>

The above definition imply the existence of the following relational operators and logical operators precedence within DARRIE.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Relational Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>equality</td>
<td>=, ( \neq ) or ( \in )</td>
</tr>
<tr>
<td>non-equality</td>
<td>( \neq )</td>
</tr>
<tr>
<td>greatness</td>
<td>&gt;, ( \geq )</td>
</tr>
<tr>
<td>smallerness</td>
<td>&lt;, ( \leq )</td>
</tr>
<tr>
<td>greatness and equality</td>
<td>( \geq )</td>
</tr>
<tr>
<td>smallerness and equality</td>
<td>( \leq )</td>
</tr>
</tbody>
</table>

Figure 3.4.5.1.1 The Relational Operators.

<table>
<thead>
<tr>
<th>Order of Precedence</th>
<th>Logical Operation</th>
<th>Logical Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest</td>
<td>NOT</td>
<td>( \neg )</td>
</tr>
<tr>
<td></td>
<td>AND</td>
<td>( \land ) and ( \land )</td>
</tr>
<tr>
<td></td>
<td>OR and</td>
<td>( \lor ) and ( \lor )</td>
</tr>
<tr>
<td></td>
<td>EXCLUSIVE-OR</td>
<td>( \oplus )</td>
</tr>
</tbody>
</table>

Figure 3.4.5.1.2 The Logical Operators and Their Precedence.
Figure 3.4.5.1.3 The General Form of Conditional Statement.
3.4.5.1.1 Scope Terminator of Conditional Statements

Unlike ALGOL 60 and PL/1, but like ALGOL 68, DARRIE requires a special terminator "." as an indication of the scope of the operations to be performed when the condition of the Boolean expression in a \( \text{if} \) statement without \( \text{else} \) clause is satisfied or as an indication of the scope of \( \text{else} \) clause. This method, as opposed to a compound statement as in ALGOL 60 and PL/1, causes the conditional clauses to be terminated uniformly, where they consist of one or more than one statement. It also reduces the possibility of the occurrences of BEGIN/END unsynchronization. Therefore, the programmer is forced to explicitly define the scope of statements and thus erroneous flows of control are less likely to occur since the programmer's reliance on implicit syntactic conventions is reduced.

Examples:

\[
\text{\( \text{if} \) clause (1)}
\]

\[
\text{\( \text{else} \) clause}
\]

The above conditional statement is the expression of the following algorithm within DARRIE.
3.4.5.1.2 Nested Conditional Statements

Unlike FORTRAN and COBOL permits nested conditional statements: a natural generalization of conditional statements capabilities in order to achieve uniformity and generality.

Example:

```
(1) 
```

```
アーシーズ: ピー - ミー ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノード ノー
Example (1) is an example of nested \( \text{if} \) statements and the expression of the following algorithm within C.

```c
print
if (condition) {
    call subroutine
}
```
The above conditional statement is the expression of the following algorithm within DARRIE.
3.4.5.2 Unconditional Transfer Statements

3.4.5.2.1 Unconditional GOTO

Figure 3.4.5.2.1.1 illustrates the unconditional GOTO statement in C5, which can be used to transfer control to a statement, unconditionally.

Although, the unconditional transfers of control are more efficient than more restricted forms of control transfer, they could be misused to develop complex control flows which could be expressed more understandably with other control structures.

3.4.5.2.2 Indexed GOTO

Because an array can be of LABEL/ G5 kind within C5, then the individual array elements can be used as the destination in an unconditional transfer of control statement.

Example:

declaration:
initialization:

usage:

The above transfer of control statement causes the control to be transferred according to the following table:

<table>
<thead>
<tr>
<th>Value of $d$</th>
<th>The statement to which the control is transferred</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

3.4.5.2.2.1 Out-of-Range Index of an Indexed GOTO

When the index of an indexed GOTO statement does not fall within the range of values for which a label is defined, the clause of an indexed goto statement. This feature provides implicit test of range value only when clause is specified, i.e., when the programmer is not certain that all index values will be within the appropriate range and enables the programmer to explicitly transfer the control to the appropriate statement. 's interpretation is more
reliable and explicit than ALGOL 60's and some FORTRAN implementations interpretation which specify an undefined effect when the index value is outside the range of values for which a label is defined. DARRIE's interpretation also provides more understandability and portability than FORTRAN which permits each implementation to specify the effect of an out-of-range index.

![Diagram](image)

Figure 3.4.5.2.2.1.1 The General Form of Indexed GOTO Statement.

3.4.5.2.3 Multiple-Route GOTO

DARRIE offers a combination of conditional and indexed GOTO capabilities in the multiple-route GOTO statement whose general form is given in Figure 3.4.5.2.3.1.
Figure 3.4.5.2.3.1 The General Form of Multiple-Route GOTO.
3.4.5.2.3.1 Implicit Indexing of Constituents

Example (1) demonstrates the implicit indexing of the constituents capability in a multiple-route GOTO statement within (3'). It is equivalent to:

```
DARRIE
GOTO (bab-253, bab-55, bab-65, bab-95)
```

in FORTRAN (with exception that the statement labels cannot be mnemonic in FORTRAN) and to:

```
SWITCH S [ bab-253, bab-55, bab-65, bab-95 ]
  .
  GOTO S [ alf ]
```

in ALGOL.

3.4.5.2.3.2 Explicit Indexing of Constituents

Example (2) demonstrates the explicit indexing of constituents. It is equivalent to:

```
DARRIE
GOTO (bab-253, bab-55, bab-65, bab-95)
```

in FORTRAN (with exception that the statement labels cannot be mnemonic in FORTRAN) and to:

```
SWITCH S [ bab-253, bab-55, bab-65, bab-95 ]
  .
  GOTO S [ alf ]
```

in ALGOL.
Example (2) demonstrates the explicit indexing of the constituents capability in a multiple-route GOTO statement within (5'). It is not equivalent to any single FORTRAN or ALGOL statement, but if clause is not taken into consideration, it is equivalent to the following JOVIAL/J73 case statement:

```
CASE  of
   2 :  
   3 :  
   5 :  
   7 :  
```

3.4.5.2.3.3 Constituents with Several Values

Example (3) demonstrates that the constituents of multiple-route GOTO statements can have several values, as permitted in PASCAL.
3.4.5.2.3.4 Constituents with Index Ranges

Example (4) illustrates the capability of specifying index ranges with the constituents of a multiple-route GOTO statement within \[ \text{GOTO} \], as suggested by Barth.²

3.4.5.2.3.5 Explicit Predicates

Example (5) demonstrates a powerful feature which incorporates a kind of decision table capability into \[ \text{GOTO} \]. The following is a pictorial representation of the above multiple-route GOTO statement.
3.4.5.3 Return of Control from Procedures

statement is the means of returning control from a subroutine to its caller.

3.4.5.4 Termination of Execution

statement is the means of terminating program execution and returning control from \( \text{DARRIE} \) to \( \text{PEESHTEZ} \).

3.4.5.5 Iteration Statements

\( \text{DARRIE} \) provides a variety of methods for repetitively executing a set of program statements (to exploit the
inherent capabilities of computers). The methods can be used for programming iterative solutions, processing sequential files and arrays, or remaining in a nonproductive loop until a special event occurs by means of interrupt facilities, etc.

3.4.5.5.1 Various Loop Structures

All the three forms of iterations reported by Knuth\textsuperscript{7,1974} and suggested by Dahl:\textsuperscript{3}

\begin{verbatim}
LOOP WHILE ...; LOOP; LOOP;
... ... ...
REPEAT; REPEAT WHILE...; WHILE ...
... REPEAT;
\end{verbatim}

are included within the general forms of the iteration statements of DARRIE. In addition the general form includes the indefinite iteration with the loop control mechanism.
Figure 3.4.5.5.1.1 The General Form of Test-at-Beginning Iteration Statement.
Figure 3.4.5.5.1.2 The General Form of Test-at-End Iteration Statement.
Figure 3.4.5.5.1.3 The General Form of Test-in-the-Middle Iteration Statement.
3.4.5.5.1.1 Test at the Beginning (WHILE Statement)

This form of iteration statement ensures that the loop body is not executed unless a condition or a Boolean expression is satisfied or a control variable has not exceeded the specified bound.

Examples:

The above set of statements are the expression of the following algorithm:
The above algorithm can be expressed by means of an indefinite iteration statement in which the loop control mechanism is used, as follows:
3.4.5.5.1.2 Test at End (REPEAT-UNTIL)

This form of iteration allows the body of the loop to be executed once and then the subsequent executions of the body of the loop are under the control of a condition, a Boolean expression, or an infinite iteration with a control variable.

Example:

\[ p^0 \rightarrow 1 \]

(1)

The above set of statements are the expression of the following algorithm:
3.4.5.5.1.3 Test in the Middle

This form of iteration statement allows the loop condition to be tested after the execution of a designated first part of the loop, but before the execution of the second part.
3.4.5.5.2 Control Variable in Indefinite Iterations

The indefinite iteration statements included within the general forms of the three available iteration statements, contain a control variable, an expression defining the initial value of the control variable, an incremental/decremental value, and a terminating value.

In (\( \sigma \)), such iteration statements have fixed number of iterations. The fixed number of iterations can be determined once the initial, terminating, and incremental/decremental values are known.

3.4.5.5.2.1 Control Variable Values

The control variable values are limited to discrete values. The restriction would allow the generation of sequences of values that can be used for accessing array elements and error-prone approximations, which could occur with real control variables can be prevented.

3.4.5.5.2.2 Subscripted Control Variable

Like ALGOL and PL/1 allows the loop control variables to be array elements (consistent with the ability to use subscripted and unsubscripted variables interchangeably). In such cases, the subscripts of the control variables are evaluated once, upon the initiation of the iteration statement. Thus the address of the array element
used is computed once and thus contributes to error prevention and explicitness of the program.

3.4.5.5.2.3 Control Variable Changeability

While an iteration statement is active, its control variable may not be explicitly assigned. As the result the READ ONLY (\texttt{READ ONLY}) attribute is attached to it until the termination of the iteration. More efficient code, and more reliable programs are the result of such restriction.

3.4.5.5.2.4 Control Variable Accessability

In the control variable can be accessed outside of the body of the loop for the convenience of the programmer, although this would enforce the compiler to update the value of the control variable before the loop is exited.

3.4.5.5.2.5 Initial Value of Control Variable

The initial value must always be assigned to the control variable and can be an expression which is evaluated once at the initiation of the iteration statement.

3.4.5.5.2.6 Increment/Decrement

The increment/decrement may be omitted which would cause the default value 1 to be used. For the convenience of the programmer the increment/decrement can be an
expression which is evaluated once at the initiation of the iteration statement.

3.4.5.5.2.7 Termination Value

The termination value is evaluated at the initiation of the iteration statement and it is subsequently used to determine when the iteration is to be terminated.

3.4.5.5.3 Scope Terminator of Iteration Statements

In $C_\text{DARRIE}$ loop bodies are delimited by a special terminator "." as an indication of the scope of the operations to be performed when the condition or the Boolean expression is satisfied or the control variable has not exceeded the termination bound. This method creates uniformity within iteration statements, whether or not they consist of a single or more than one statement. Also it reduces the possibility of BEGIN/END unsynchronization.


29 TACPOL Language for Command and Control Systems.

30 USA Standard FORTRAN, USA Standards Institute, USAS X3.9-1966, March 1966.


3.5 Conclusion

In this thesis the design specification of an interactive computer programming system (in which all the major tools to solve problems by preparing computer programs and communicating with computers using a subset of Farsi language are included) was given. In addition to the design specification of a simulated input/output device for transmission of Farsi characters, the implementation process of such a subsystem was also described. The implementation of the other system components, such as the text editor/string processing language and the general purpose programming language, are necessary to achieve the desired practical goals and are considered to be part of the future investigation of this thesis.

Before the implementation to begin, a thorough study of the feasibility of the inclusion of all the features included in the design specification, must be carried out. The addition of the new features meet the computational need of the society at that time, and the modification of certain features and/or structures must be carried out.

Also, the importance of the creation of this system must be publicized and those in the area of education must be educated about its importance in the advancement of education in computer science and in other areas in which computer application can have impact.
BIBLIOGRAPHY


Short, Robert A., unpublished lecture notes for switching theory and automata courses, Department of Electrical and Computer Engineering, Oregon State University, 1974.


TACPOL Language for Command and Control Systems.


USA Standard FORTRAN, USA Standards Institute, USAS X3.9-1966, March 1966.


