Very High Level Languages (VHLL) provide higher level abstractions and more powerful primitives than high level languages (HLL). A programmer uses these abstractions to solve a problem by specifying "what" is to be done rather than "how" it is to be done.

This research work reports the design and development of Bagit, a new VHLL for application programming. Bagit provides 1) an encapsulation mechanism (bag) and, 2) an information hiding mechanism (filter) which are used in a consistent manner to support data, type, control, functional, and access abstraction.

A data abstraction (ADT) in Bagit is defined by a bag which encapsulates the representation and valid operations on a type, and a filter which defines the interface to the abstraction. A type abstraction is a data abstraction which has two or more filters, each filter defining a new type. A zero type is a type or data abstraction where all its attributes are of type function. Functional abstraction in
Bagit is provided via functions which can return bags. Furthermore, these functions may be used in conjunction with loop constructs and bag expressions, providing control abstractions. In this case they are generators of bags. Finally, associative referencing is supported by labelled bags. By hiding the irrelevant information of how an object is accessed, and only specifying what the object is, access abstraction is achieved.

The significant contributions to the field of programming languages are, 1) Realization of an information hiding mechanism as a language design principle in addition to an encapsulation mechanism, 2) Attempting to provide a bridge between the areas of programming languages and database systems, 3) Treating various forms of abstraction using a single abstraction mechanism in a consistent and uniform manner, 4) Solving the problems of zero types and more than one type, and 5) Improving functional and control abstraction.
Bagit: A Very High Level Language for Application Programming

by

Taymoor Arshi

A THESIS

submitted to

Oregon State University

in partial fulfillment of the requirement for the

degree of

Doctor of Philosophy

Completed December 17, 1982

Commencement June 1983
ACKNOWLEDGEMENT

Thank you Ted. For your contributions to this research work, for directing me through it and above all, for your friendship.
TABLE OF CONTENTS

I. Introduction
   1. Very High Level Languages (VHLL) 1
      1.1 General Features of Very High Level Languages 7
   2. Bagit: A New Very High Level Language 9

II. Basic Concepts and Terminology 14
    1. Lexical Tokens 14
    2. The Concept of Data Type 17
    3. Built-in types of Bagit 18
    4. Blocks, Locality, and Scope 22
    5. Constant, Type, and Variable Declaration 24

III. The Bag Constructor 27
     1. Introduction 27
     2. Attributed Bags and Dynamic Bags 28
     3. Labelled Bags and Access Abstraction 30
        3.1 Introduction 30
        3.2 Definition of a Labelled Bag 31
     4. Conceptual Representation of Bags 34
     5. Hidden Ordering and Unordered Processing 37
     6. Reachability, Access Rules, and Access Abstraction 38
     7. The Expressive Power of a Bag 48
     8. Filters: Information Hiding Mechanism 54
        8.1 Defining a Filter for a Bag 59
     9. Bag Calculus 61
    10. Bag Input / Output 65
        10.1 Screen Input/Output 65
        10.2 Conventional I/O 68

IV. The Operational Environment 70
    1. Assignment Statement and Expressions 70
    2. Control Structures 73
       2.1 Sequence 73
       2.2 Conditional: if-then-then-else 73
       2.3 Iterative Statements 74
          2.3.1 Forall-do 74
          2.3.2 Until-do 75
       2.4 The exit statement 76
       2.5 The in statement 77
    3. Functions 78
       3.1 Functional Abstraction 78
       3.2 Function Constructor 79
       3.3 Generic Functions 81
    4. Generators of Bagit and Control Abstraction 83
       4.1 The Iterators of Clu 84
       4.2 The Built-in Generators of Bagit 88
4.2.1 The Range Generator: .. (dot dot)  88
4.2.2 The Qualifier SUCHTHAT  89
4.3 User-defined Generators  90
5. Abstract Data Types  92
5.1 The Definition of an Abstract Data Type  96
5.2 From Simula to Clu  97
5.3 The Rationale of ADT  100
5.4 Data and Type Abstraction in Bagit  102

V. The Runtime Environment  110
1. A Compiler-Interpreter Organization  110
2. LL(1) Parsers: A Nonbacktracking Top-Down Technique  114
3. Table Management  119
  3.1 Type Definition Table  119
  3.2 Extensions of Type Definition Table  121
  3.3 Symbol Table Organization  131
  3.4 Constant Table  132
4. Storage Allocation  134
  4.1 Static Storage Allocation  134
  4.2 Dynamic Storage Allocation  135

VI. Conclusion  141

VII. Bibliography  145

VIII. Appendices  149
Appendix A: Collective Syntax Diagrams of Bagit  149
Appendix B: BNF Syntax of Bagit  156
Appendix C: The LL(1) grammar and Selection Sets  162
Appendix D: Program Listing: Lexical and Syntax Analyzers  174
Appendix E: Sample Runs  183
Appendix F: The Internal Codes  191
Appendix G: Language Reference Card  194
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Syntax of an identifier</td>
<td>15</td>
</tr>
<tr>
<td>2. Numbers</td>
<td>15</td>
</tr>
<tr>
<td>3. Character strings</td>
<td>16</td>
</tr>
<tr>
<td>4. Syntax of a block</td>
<td>23</td>
</tr>
<tr>
<td>5. The scope of variables</td>
<td>23</td>
</tr>
<tr>
<td>6. Constant declaration</td>
<td>24</td>
</tr>
<tr>
<td>7. Type definition</td>
<td>25</td>
</tr>
<tr>
<td>8. Subrange types</td>
<td>25</td>
</tr>
<tr>
<td>9. Variable declaration</td>
<td>26</td>
</tr>
<tr>
<td>10. Attributed bags</td>
<td>28</td>
</tr>
<tr>
<td>11. An example of an attributed bag</td>
<td>29</td>
</tr>
<tr>
<td>12. Dynamic bags</td>
<td>29</td>
</tr>
<tr>
<td>13. A typical bag type</td>
<td>30</td>
</tr>
<tr>
<td>14. The general form of an attributed bag</td>
<td>32</td>
</tr>
<tr>
<td>15. General form of labelled bags</td>
<td>32</td>
</tr>
<tr>
<td>16. Labelled bags</td>
<td>33</td>
</tr>
<tr>
<td>17. Generalization of bags</td>
<td>33</td>
</tr>
<tr>
<td>18. Representation of bags</td>
<td>35</td>
</tr>
<tr>
<td>19. Representation of attributed bags</td>
<td>35</td>
</tr>
<tr>
<td>20. Representation of dynamic bags</td>
<td>36</td>
</tr>
<tr>
<td>21. Representation of labelled bags</td>
<td>36</td>
</tr>
<tr>
<td>22. Tree structure of a typical bag</td>
<td>37</td>
</tr>
<tr>
<td>23. Syntax of a label</td>
<td>39</td>
</tr>
<tr>
<td>24. Syntax of an actual label</td>
<td>41</td>
</tr>
<tr>
<td>25. Accessing the bag objects</td>
<td>41</td>
</tr>
<tr>
<td>Figure</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>26. A bag type for a small database</td>
<td>43</td>
</tr>
<tr>
<td>27. Representation of the bag for small database</td>
<td>44</td>
</tr>
<tr>
<td>28. The snap-shot of the small database bag</td>
<td>45</td>
</tr>
<tr>
<td>29. Simulating associative referencing in Pascal</td>
<td>46</td>
</tr>
<tr>
<td>30. A bag type for a digraph</td>
<td>49</td>
</tr>
<tr>
<td>31. Representation of a digraph bag</td>
<td>50</td>
</tr>
<tr>
<td>32. An example of a digraph</td>
<td>50</td>
</tr>
<tr>
<td>33. The equivalent bag of a digraph</td>
<td>51</td>
</tr>
<tr>
<td>34. Arrays represented by bags</td>
<td>53</td>
</tr>
<tr>
<td>35. The concept of a filter</td>
<td>55</td>
</tr>
<tr>
<td>36. Another view of a filter</td>
<td>56</td>
</tr>
<tr>
<td>37. Scope of filtered bags</td>
<td>56</td>
</tr>
<tr>
<td>38. Use of filters to bypass nodes</td>
<td>58</td>
</tr>
<tr>
<td>39. General form of filters</td>
<td>60</td>
</tr>
<tr>
<td>40. Defining a view of a bag</td>
<td>60</td>
</tr>
<tr>
<td>41. Input/Output</td>
<td>66</td>
</tr>
<tr>
<td>42. Prompts</td>
<td>66a</td>
</tr>
<tr>
<td>43. Screen I/O</td>
<td>67</td>
</tr>
<tr>
<td>44. Syntax of a prompt</td>
<td>68</td>
</tr>
<tr>
<td>45. Expressions</td>
<td>72</td>
</tr>
<tr>
<td>46. if-then-then-else syntax</td>
<td>73</td>
</tr>
<tr>
<td>47. Forall-do statement</td>
<td>75</td>
</tr>
<tr>
<td>48. Until-do statement</td>
<td>75</td>
</tr>
<tr>
<td>49. An ADT in Bagit</td>
<td>104</td>
</tr>
<tr>
<td>50. A generic type abstraction in Bagit</td>
<td>105</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>51.</td>
<td>A zero type example</td>
</tr>
<tr>
<td>52.</td>
<td>A compiler-interpreter organization</td>
</tr>
</tbody>
</table>
In the past decade abstraction mechanisms have been under constant investigation and have become one of the major themes of research in the area of programming languages. Its growing importance can also be observed in other areas of computer science such as database systems (information hiding and data abstraction) and computer architecture (hardware level, object oriented microprocessors).

Early programming languages (e.g. Fortran) supported functional abstraction (e.g. separation of use of a SORT function from its implementation) and a primitive form of data abstraction (e.g. arrays). Since the early 1970's, data abstraction has received more attention. Today, abstract data types are amongst the principle design issues of most recent programming languages.

Abstraction is emphasized even more in Very High Level Languages by making a clear distinction between "what" is to be done and "how" it is to be done. The programmer can abstract away from the irrelevant details of a problem solving process.

This research work presents the design and development of programming language Bagit which supports data, control,
access, functional, and type abstractions in a uniform and consistent manner. We will show that four contemporary problems of abstraction can be solved by the mechanisms of Bagit. These problems are listed below and described in section 2.

1. Typing of Objects
2. Loop Iterators
3. VHLL Functional Abstraction
4. Associative Referencing

1. Very High Level Languages (VHLL)

Definitions

A Very High Level Language has been described as one which is used to specify "what" is to be done rather than "how" it is to be done [21]. In a broader context, a Very High Level program is a prescription for solving a problem without regard to details of "how" it is solved.

The youth and rapid growth of this class of languages has caused confusion in terminology. Among the terms now being used to refer to the class of Very High Level Languages are nonprocedural, less procedural, functional, goal oriented, problem oriented, specification, and declarative languages [21]. Throughout the text of this thesis the term Very High Level Language (VHLL) is used.
Language Abstractions

A careful examination of the evolution of programming languages suggests a close relationship between the concepts of information hiding and abstraction in programming languages. In the early programming languages, the only abstraction mechanism provided by assembly languages over machine languages was symbolic naming, thus, the programmer could abstract away from the machine representation of programs.

The first high level languages provided a richer set of mechanisms for defining abstractions. Although primitive, these abstractions could hide more irrelevant details of how a problem is solved. The programmer could manipulate data abstractions such as a two dimensional array without regard to how the abstraction is implemented. He/She was concerned more about "what" and less about "how".

The next evolution of high level languages provided data (and in some cases control) abstraction facilities. A Simula class, a Clu cluster, or an Ada package can directly represent an information hiding module. This module can be separated from the main program, and thus the details can be hidden from the main program [22]. In this regard, information hiding and abstraction meet in their maturation as a language design principle [11].

Moreover programming languages can themselves be
viewed as abstractions away from the underlying processor.

Language Level

It seems reasonable to relate abstraction and information hiding with the 'level' of the language. Although a concrete definition of the level of a language is not yet commonly accepted, one expects the class of Very High Level Languages to be "higher level" because they offer even more abstraction. The irrelevant details of "how" a problem is solved are always hidden in the abstraction of "what" the problem is.

Among the languages considered to be Very High Level are query languages [21], set oriented languages [21] (e.g. SETL with an abstract set structure [9]), and functional languages [21] (e.g. Lisp with an abstract tree structure [9]).

Suitability of a Language

The motivation behind the work in Very High Level Languages is to ease the programming task by providing the programmer with a language containing primitives or abstractions suitable to the problem area. Very high level languages while primarily intended to increase the programmer's productivity by easing the programmer's task, can also be expected to enhance the reliability and understandability of code [24].

Programmer Productivity

The productivity of a programmer is largely effected
by the amount of effort needed to express necessary details in a program. In a very high level language the programmer is concerned about "what" is to be done rather than "how" to do it, this leads to fewer details, and thus greater productivity because he/she is then able to spend his/her effort in the right place: solving the problem.

In a very high level program, the solution is specified in terms of structures or abstractions which are relevant to the problem rather than detailed operations, data and control structures which are relevant to some machine organization [12,21]. A need clearly exists for programming languages designed with the goal of suppressing programming details [12].

Correctness

A very high level language provides facilities to build levels of abstraction corresponding to the natural levels of abstractions in a problem. This would permit better modularization and structuralism of programs thus enhancing the correctness of programs.

Structured Programming

The work on very high level languages and structured programming are closely related. Each is based on the idea of making use of those abstractions which are correct for the problem being solved. The rationale for using abstraction is the same in both approaches: to free the programmer from concern with details not relevant to the
problem being solved [24].

**Structured Data Representation**

Another aspect of very high level languages to consider is the means (building blocks) provided in the language to represent the information needed to solve a problem. A very high level language must have a data representation that lends itself to unordered processing [12]. A powerful data structure increases the expressive power of a language by avoiding a lot of unnecessary operations to interpret low level data structures. A VHLL programmer should be able to focus on "what" to represent and not "how" it is to be represented.

**Object Representation**

Most programmers think in terms of objects, tables, decisions and actions, rather than variables, arrays, control structures and subprograms. Therefore, most of the time, considerable amount of time is spent to translate the concepts in the programmer's mind into the features available in the language. This translation from a conceptual level (programmer) to a physical level (program) should be minimized by closing the gap between the two levels so that higher productivity is achieved. Very high level languages narrow the conceptual gap between man and machine by permitting a description of the problem to be solved; statement of the algorithmic solution is not explicitly required [7].
Complexity

However, providing powerful data and control structures should not result in a complex language. Over the past few years, there has been an almost unabated tendency for languages to get larger and larger [34] in an effort to provide more powerful and more varied features to satisfy more users. A language should be limited in complexity and size.

It is not possible to assert that a given programming language is a very high level language in the absolute sense, but one can state that it possesses certain VHLL features [21], as listed below:

1. Associative Referencing
2. Aggregate Operators
3. Nondeterminism

1.1 Desirable Features of Very High Level Languages

Associative Referencing

Associative referencing (content addressability of data) is perhaps one of the most important features for inclusion in a programming language, because the programmer does not have to specify access paths explicitly or program an algorithm to conduct a search for a specific data structure [21]. Therefore, unnecessary details of how to access a data item are eliminated.

Ordering a Data Structure

Searching and sorting are very closely related to
associative referencing. In many cases the programmer writes a sorting procedure merely because there will be a search at a later time. In this case, "what" we want to do is search, and the sort is an irrelevant detail. A language supporting associative referencing removes the need for searching by hiding the irrelevant order property. As a result, programs would tend to be simpler due to fewer details.

Aggregation

Another important feature of very high level languages is aggregate operators. It is possible to avoid writing loops in some programming languages that provide aggregate operators [21]. Clearly, the algebraic operators SELECT, PROJECT, and JOIN defined by Codd [5] are examples of aggregate operators. Also, the traditional set operations (union, intersection, etc.) represent another class of aggregate operators.

Sets

The traditional aggregate data structures in programming languages have been arrays where the concept of indexing and sequencing have been important. Languages which provide sets as data structures do not rely on the relative position of data items in the sets for accessing purposes, but make use of associative referencing [21]. So, there seems in general to be a close relationship between associative referencing and aggregate operators via
the supporting data structure.

**Nondeterminism**

Nondeterministic programming and parallelism is another important issue in very high level programming. This feature appears in most Artificial Intelligence languages. In a deterministic program the result of every operation is uniquely defined. A nondeterministic program is one that is allowed to contain operations whose outcome is not uniquely defined but is limited to a specific set of possibilities [16]. In most cases nondeterministic programs are executed as backtracking algorithms [13]. The importance of nondeterministic programming is in hiding of the bookkeeping details of the backtracking.

2. **Bagit: A New Very High Level Language**

**Goal of Bagit**

Programming languages have evolved to higher level notations because software development and maintenance costs dominate hardware costs for most applications [36]. The design of new programming languages is generally motivated by the following factors:

- state of technology
- new application areas
- new ideas or concepts to be developed
- improvement over existing languages by resolving the problems observed in them
The design of programming language Bagit is motivated by the last two factors.

**Typing of Objects Problem**

Most recent programming languages provide a linguistic construct to define a data abstraction. However, Rowe [35] has identified a major problem with the abstraction mechanism provided in these languages (e.g. Clu [24,25]). The problem is that every data abstraction defined in these languages corresponds to a single abstract data type. There are circumstances where we would like an abstraction to define more than one type [35]. Bagit solves this problem by a type abstraction. There are also circumstances where the data abstraction may be required to define a zero type [35]. A zero type is used to group together a collection of procedures or subroutines in a subroutine library. This problem is solved in Bagit by a data or type abstraction.

**Loop Iterator Problem**

Unlike data abstraction, control and functional abstraction has not evolved within high level languages. A few of the most recent programming languages allow control abstraction through a generalized loop structure called an iterator. An iterator produces the next object in a collection and each object is consumed by the loop, one at a time [25]. The problem with an iterator is resetting the
iterator so it can be reused. This problem is solved in Bagit by the bag generators.

**VHLL Functional Abstraction Problem**

In the area of functional abstraction effectively no significant advances have been made in programming languages since the concept of generic subprograms were proposed in Simula 67 [32].

Bagit provides greater abstraction by employing a single encapsulation mechanism called a bag which is used uniformly for different purposes. The bag is the only type constructor in the language. A type constructor is used to define new types from standard built-in types or user defined types.

**Solutions Given by Bagit**

A data abstraction consists of a bag containing data objects as well as functional objects, and a filter. The filter is the information hiding mechanism of the language. It defines the access right (or visibility right) to the objects in the bag. Only those operators (functional objects) which can access or manipulate the data objects in the bag are visible through the filter. Data abstraction is discussed in chapter III.

The two problems of zero types and more than one types which are identified in existing languages are resolved in Bagit by a type abstraction. The solution is to permit a data abstraction which can contain more than one filter.
Each filter in the encapsulation defines a new type from the same abstraction. A data abstraction which encapsulates functional objects only, defines a zero type. This issue is addressed in chapter III.

Functions in Bagit can return objects of a built-in type or type bag. Therefore, traditional functional abstraction is preserved in the language by hiding the implementation of the function. Moreover, since functions can return bags as their result, a function may represent several objects at one time. Generic functions are also supported in Bagit and are discussed in chapter IV.

Access abstraction is achieved via associative referencing capability of Bagit. Bagit allows objects of a bag to be labeled and identified by these labels. The labeled objects are then accessible through their contents and no specific access path (addressing) is required to locate these objects. Access abstraction therefore, eliminates unnecessary search for an item in a data structure. Access abstraction is explained in chapter III.

Bag generators provide control abstraction. They generate objects of a bag to be consumed by a loop construct or in a bag expression. There are two built-in bag generators in Bagit. A qualifier (suchthat) which qualifies the objects of a bag producing a sub-bag. And a range generator which generates objects between two given values of a known, enumerable type. Moreover, any function
which returns a bag, is in fact a bag generator; it produces the objects of an abstract bag and separates the process of generating the objects from their use. Generators are discussed in chapter IV.

The syntax diagrams given throughout the text of this thesis are meant to be self contained and localized. The collective syntax diagrams of the language appear in Appendix A.
II. Basic Concepts and Terminology

In this chapter the notations and vocabulary of the language are briefly discussed.

1. Lexical Tokens

The lowest level program constructs with linguistic meaning are called lexical tokens [42]. This section defines the lexical tokens of Bagit.

Characters

All tokens may be composed from alphabetic, numeric, or special characters as follows:

(a) Alphabetic characters

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

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

(b) Numeric characters

0 1 2 3 4 5 6 7 8 9

(c) Special characters

) ( ' + - * ; / = : ? . _ > < [ ] ,

and the space character (blank).

Identifiers

An identifier is formed by a sequence of alphabetic and numeric characters, the first being alphabetic. An _
(underscore) may be inserted between parts of an identifier. An identifier must fit on a single line and space character is not allowed in between. Although an identifier may be very long, the number of significant characters will be determined by a specific machine environment. The syntax diagram of an identifier is given in Figure 1.

![Figure 1. Syntax of an identifier](image)

Examples:

Legal Identifier SNOBOL A i Index5

**Numbers**

The usual decimal notation is used for numbers which are the constants of the data types integer and real (Figure 2).

![Figure 2. Numbers](image)

Examples:

2.55 -27 +2.55 0.75 3.0
Character strings

A character string is formed by a sequence of characters enclosed by quote marks. Strings may be of any length including zero (null string), (Figure 3). In strings containing a quote mark, the quote must be written twice. Each string must fit on a single line.

Figure 3. Character strings

Examples:

'This is a string'  'A'  '123'  '*'  ''
'I''m another string'

Comments

Comments start with a double minus sign (hyphen) and are terminated by the End-Of-Line marker. Comments will be ignored by the translator; their sole purpose is for program documentation and readability.

Example:

-- This is a comment,
-- and its continuation.

Reserved Keywords

Reserved keywords may not be used as identifiers. In hand written programs (and throughout this text) they are underscored. The reserved keywords of Bagit are:
2. The Concept of Data Type

A program manipulates abstract objects that represent real world objects. In reality, objects have different characteristics and behaviors. For instance, one's age has a variable nature, whereas his/her birth date is unchangable. Moreover, certain actions are conceptually valid for some objects and absolutely meaningless for others. A 15% raise on "salary" is a meaningful action, but a 15% raise on "name" is completely invalid. Some objects have a more complex structure than the others which have an primitive structure. Complex structures consist of simpler ones.

Rowe [36] has distinguished two views of the role of type in a programming language. The first view is that a type is a set of values. A data type is defined by specifying the representation for the values. For example, a queue can be described as an array and two indexes FRONT and BACK, which index the elements at the front and back of
the queue respectively. Operations on values of a particular data type are coded as procedures but they are independent of the data type. This view is embedded in most programming languages such as Pascal, PL/1, Fortran, etc.

The second view, referred to as the "types are not set" view, is that a type is a language mechanism to enforce authentication and security [36]. Authentication ensures that any value supplied to an operation is consistent with the type expected by that operation. Security ensures that any operation applied to a value is meaningful for that type. Therefore, in this view of type, the behavior of objects of the type is reflected in the definition of that type. This view of type is embedded in the programming languages which support abstract data types.

3. Built-in types of Bagit

Bagit employs the second view of type, i.e., the "types are not sets" view. Any real world object is specified by a name and a data type of Bagit. Since Bagit is a strongly typed language, every data object used in the program must be declared. A declaration binds an identifier to an object.

The rest of this section describes the standard built-in types of Bagit which are: integer, real, string,
scalar, boolean, and function.

**Integers**

The constants of the data type integer are the whole numbers between some implementation-defined limits. Some examples of valid integers are given below:

-123   +20   32767   0

The traditional arithmetic operators defined for the integers are:

+      addition
-      subtraction (or unary negation)
*      multiplication
div    integer division (divide and truncate)
mod    modulus (remainder after dividing)

**Reals**

A value of type real is an element of the implementation-defined subset of real numbers. These numbers are represented with an optional sign, a whole number part, a decimal point, and a fractional part. There must be at least one digit on each side of the decimal point. The following are some examples:

3.0   -0.3   +1982.75

The traditional arithmetic operations available for real data type are listed below:
+ addition
- subtraction (or unary negation)
* multiplication
/ real division (no truncation)

Strings

The elements of the data type string consist of zero or more alphabetic, numeric, or special characters. The operators available for string data type are as follows:

concat (s1,s2,...) concatenation of two or more strings.
substr (s,i,j) a substring of s starting at position i and of length j
length (s) number of characters in string s
index (s1,s2) position of the first occurrence of s2 in string s1.
insert(s1,s2,i) inserting string s2 into string s1 starting at position i.
delete(s,i,j) deleting j characters from s starting at position i.

Strings of variable length

The number of characters in a string is called its length. Thus, 'ABC' has length 3. An integer constant may be used to specify the length of a string in a type definition or variable declaration statement. For example:
const N is integer 10;
var M is string(N);

declares M as a variable of type string which may hold up to 10 characters.

Bagit allows varying length strings. Thus, no length is specified for the string object. It may hold zero or more characters. For instance:

var M is string;

declares a variable M with varying length.

Scalars

The scalar data type represents an ordered set of manifest string constants. For example the data type:

('Red', 'Blue', 'White')

defines a scalar type where objects of the type can assume values 'Red', 'Blue', or 'White'. There are two functions available for the scalars. If S is a manifest constant in the data type T, pred(S) is the previous constant. Pred(s) is undefined if S is the first constant of T. Also, next(S) is the next constant of type T unless S is the last string constant of T. In the example given above,

pred ('Blue') = 'Red',
next ('Blue') = 'White',
but \texttt{pred('Red')} and \texttt{next('White')} are undefined.

Unlike Pascal and similar languages, input/output on the elements of this type is possible.

\textbf{Boolean}

Boolean is a subrange type ('false', 'true') and no longer needs be treated separately from Bagit scalars. For boolean type, input/output is also possible.

\textbf{Functions}

Functions are a collection of executable machine instructions as defined by a function constructor. A variable may be declared of type function as described in chapter IV.

4. Block, Locality, and Scope

A block consists of a data object declaration part, where all data objects associated with the block are declared, and a statement part where a sequence of actions is given to perform certain computations. A block is either a program or a function.

All constants, types, and variables belonging to the block are defined in the data object declaration part of the block. These objects are said to be local to the block. The syntax diagram of a block is given in Figure 4. For more details see Appendix A.
Figure 4. Syntax of a block

The scope of a data object is limited to the enclosing block. If one of the data objects in block A is redefined in an inner block B, its scope is limited to block B as long as the control remains in B. Therefore, it is legal to have a data object \( x \) of type \( T_1 \) in block A and another object with the same name of type \( T_2 \) in B (Figure 5).

\[
\begin{align*}
A & \quad \text{x of type boolean} \\
B & \quad \text{x of type integer}
\end{align*}
\]

Figure 5. The scope of variables

The statement part of a block is a sequence of statements enclosed in a begin-end construct.
5. **Constant, Type, and Variable Declaration**

A constant declaration introduces an identifier to denote a constant. The use of constant identifiers generally makes a program more readable and acts as a convenient documentation aid [18]. The identifier represents a constant value and cannot be changed by other parts of the program, anymore than the constant itself could be changed.

An identifier \( M \) representing a constant in a block may be used to represent another constant in a nested block but its practice is discouraged. The general form of constant declaration is given in Figure 6.

**Figure 6. Constant declaration**

Examples:

```plaintext
const PI is real 3.14;
max is integer 32768;
another max is max;
```

The type definition mechanism of Baqit may be used to define new types (user-defined). The new type may be a standard built-in type or bag. The definition binds an
identifier (type name) with a built-in type or a bag type as in Figure 7.

Figure 7. Type definition

The following are some examples:

```plaintext
type weight is integer;
color is ('blue', 'red', 'green');
price is real;
```

A type may be defined as a subrange of any other already defined scalar type. The definition of a subrange simply indicates the smallest and the largest constant values in the subrange. Since real data type is not enumerable, a subrange of the type real is not valid. The general form of a subrange definition is given in Figure 8.

Figure 8. Subrange types

Examples:

```plaintext
type days is ('m', 't', 'w', 'th', 'f', 'sa', 'su');
work is 'm' .. 'f';
rest is 'sa' .. 'su';
hour is 9 .. 17;
```

A variable is simply an object that can assume different values of a type during the execution of a
program. In other words, a variable can be thought of as a memory location that has a name and that can store a value. The variable name is a Bagit identifier which is not one of the reserved keywords. Since Bagit is a strongly typed language, all variables must be declared in a variable declaration part (Figure 9).

```
var x,y is integer;
Bool is boolean;
Key is real;
```

Figure 9. Variable declaration

Examples:
III. The Bag Constructor

1. Introduction

Bag is the underlying data structure of Bagit. A bag is recursively defined as a collection of objects of a standard built-in type or other bags. Conceptually, it is a combination of multi-sets [6], ordered sets [6], and record structures of Pascal [10, 38, 43]. Multi-sets are listed among the most valuable data structures supporting high level operations [6]. They are sets in which repeated elements are permitted.

Ordered sets are sets where an ordering is defined on the elements over some attributes of the elements [6]. For example, a set of people might be ordered by their names.

Records are compound structures formed by joining elements of arbitrary, possibly structured, types into a compound [43]. These elements or fields have names through which a specific element is accessed.

A bag is also the language mechanism for defining new types. Every bag definition corresponds to a user defined type and may appear in a type definition or a variable declaration statement.

This chapter starts with the syntax and semantics of effectively two kinds of bags, those which have a finite cardinality and objects in the bag are named, and those
with an infinite cardinality where objects in the bag are not named. It then continues with labeled bags and the concept of access abstraction. Next, filtered bags and the notion of information hiding and views of a bag are discussed. Finally, bag calculus and bag input/output conclude the chapter.

2. Attributed Bags and Dynamic Bags

Attributed Bags

An attributed bag is a compound structure consisting of a finite number of components (or fields) called attributes of the bag. The definition specifies for each attribute, its type and an identifier which denotes it. Figure 10 shows the corresponding syntax diagram of an attributed bag.

```
bag of begin id is field type ; end
```

Figure 10. Attributed bags

The bag type Date in Figure 11 is an example of an attributed bag with attributes Day, Month, and Year.
type Date is bag of
begin
  Day is 1 .. 31;
  Month is ('Jan', 'Feb', 'Mar', 'Apr',
            'May', 'Jun', 'Jul', 'Aug',
            'Sep', 'Oct', 'Nov', 'Dec');
  Year is 0 .. 2000;
end; -- of Date

Figure 11. An example of an attributed bag

Dynamic Bags

Dynamic bags are bags with potentially infinite cardinality. All objects placed in a dynamic bag are of the same type. The corresponding syntax diagram is given in Figure 12.

Figure 12. Dynamic bags

Therefore the declaration:

    type A is bag of T;
    var B is A;

defines a new type A and a variable B of type A. Every object in bag B is therefore of type T. The type T is either a standard built-in type or a previously defined bag type.

Combinations

A combination of attributed bags and dynamic bags may be used to define more complex bags. Figure 13 shows a typical bag type modelling the real world object, Employees.
3. Labelled Bags and Access Abstraction

3.1 Introduction

Associative processing, the accessing of data through a partial specification of its content, has long been a subject of interest in computer science [8]. It is of great value in information retrieval, computer aided design, artificial intelligence, and other areas of the field of computer science.

A number of languages prior to 1969 incorporated features to handle associative processing. LEAP [8] is essentially an extension of Algol 60 that provides set manipulation operations. The set elements may be either simple items or "associations", i.e., triples of the form (attribute,object,value). A set of associations corresponds to a binary relation in which "attribute" is
the relation name (all triples in one such set have the same attribute component) and within each triple, "object" and "value" are the two items that are associated [5]. However, augmenting Algol with associative processing required the addition of four new types in the language [8]. There were also some additional control structures.

Since LEAP does not allow sets to contain other sets, it essentially handles binary relations.

A bag encapsulates several objects in a single place. These objects are usually associated with one another directly or through a concept common to all. It also allows repetition of values. This is a desirable feature for certain applications. For instance, in a phone directory a person may have several phones or a single phone may be associated with more than one person. The language should abstract these situations in a convenient manner.

3.2 Definition of a Labelled Bag

A labelled bag is an attributed bag where a combination of its attributes (one or more) is chosen to label the bag.

In relational literature, columns of a relation are referred to as attributes [5]. Usually a unique combination of these attributes is chosen as a key to the relation. The two definitions are consistent in the sense that a
simple bag - a bag with all its attributes of a standard built-in type - is in fact equivalent to an n-ary relation. However, the label does not have to uniquely identify the bag unless otherwise designated so by the programmer.

To be more specific, consider the attributed bag given in Figure 14.

\begin{verbatim}
type A is bag of begin  
x1 is t1 ;  
x2 is t2 ;  
...  
xm is tm ;  
end;
\end{verbatim}

Figure 14. The general form of an attributed bag

A label for A may be any combination of its attributes \( x_1, x_2, ..., x_m \), say \( x_{i_1}, x_{i_2}, ..., x_{i_k}, k \leq m \). Figure 15 represents a labelled bag which corresponds to Figure 14.

\begin{verbatim}
type A is bag (x_{i_1}, x_{i_2}, ..., x_{i_k}) of begin  
x1 is t1 ;  
x2 is t2 ;  
...  
xm is tm ;  
end;
\end{verbatim}

Figure 15. General form of labelled bags

The general syntax diagram of a labelled bag is given in Figure 16.
We conclude this section by an example (Figure 17) which is a generalization of the example of Figure 13.

```plaintext
type
  Date is bag of
    begin
      Day is 1 .. 31;
      Month is 1 .. 12;
      Year is integer;
    end;
  Year info is bag (Year) of
    begin
      Year is 50 .. 83;
      Status is bag of
        begin
          Married is boolean;
          Children is integer;
        end;
      Salary is real;
      Pos is string;
    end;
  Employees is bag (lastname, firstname, ssn) of
    begin
      lastname, firstname is string (20);
      ssn is integer;
      history is bag of Year info;
      sex is ('male', 'Female');
      birth is Date;
    end;
var
  Employee is bag of Employees;
```

Figure 17. Generalization of bags
In the example of Figure 17, Year_info is a labelled bag type with the label 'Year'. Also, Employees is a labelled bag where the label (lastname, firstname, ssn) uniquely identifies objects of the data type Employees.

4. Conceptual Representation of Bags

The structure of a bag is defined by a tree $T=(V,E)$ where $V$ is the set of nodes representing objects in the bag, and $E$ is the set of edges representing the inclusion relationship between two objects.

The root stands for the entire bag and is identified by the bag name. All leaves are non-bag objects which are identified by an attribute name or by their parent node. In the latter case, the parent node represents a bag of values of a built-in type where no attribute name is given to its objects (e.g. bag of integer). All the intermediate nodes show a bag nested inside the bag represented by the parent node. The intermediate nodes are identified by an attribute name, a label, or a combination of both written as: attribute name:(label). For convenience, a subtree is sometimes shown by a triangle and its type name is written inside the triangle. Figure 18 represents an empty bag (a bag with no objects), a simple bag (a bag which does not contain any object of type bag), and a general bag.
Therefore the definition of an attributed bag (e.g. Figure 14) has the following structure (Figure 19).

where the subtrees with roots at $x_i$, $i=1,\ldots,m$ are the corresponding trees of types $t_i$, $i=1,\ldots,m$. If $t_i$ is a built-in type, its corresponding tree will be empty.

Similarly, the definition of the dynamic bag $A$:

```plaintext
type T is --some type definition
var A is bag of T;
```

is represented by the following tree structure (Figure 20):
Figure 20. Representation of dynamic bags

All the subtrees represent the same type, T. The root of each subtree is identified through the corresponding type T. Variable A may have zero or more of these subtrees.

Finally, the definition of the labelled bag given in Figure 15 is represented by the following tree (Figure 21):

Figure 21. Representation of labelled bags

This section is concluded by the representation of the example given in Figure 17 which is as follows (Figure 22):
Figure 22. Tree structure of a typical bag

5. Hidden Ordering and Unordered Processing

Attributed bags have a static behavior like Pascal record structures. Dynamic bags on the other hand have the characteristics of multi-sets and ordered sets. They are multi, ordered sets.

Although dynamic bags have the appearance of unordered structures, they carry a hidden ordering inside. In other words, one can refer to the first, next, or last objects of a dynamic bag. Hiding the ordering in the bag makes the structure more abstract and relieves the programmer from the irrelevant details of ordering the objects. He/She is
only concerned about "what" the first, next, or last objects are and not "how" they are ordered.

If the type of the objects of a dynamic bag is a labelled bag, at any given time all these objects are ordered based on the labels of the objects. Adding, removing, and updating preserve the relative ordering. Otherwise, if the objects are not labelled bags, the ordering is automatically maintained as the objects are added to the bag (first come, first order). Again, when new objects are added to or old objects are removed from the bag, the ordering will be modified dynamically, preserving the same relative ordering.

6. Reachability, Access Rules, and Access Abstraction

An object \( x \) in a bag \( A \) is said to be reachable if it is identified by an attribute name or a label. Therefore, no direct reference may be made to objects of a dynamic bag (e.g. \( \text{var } A \text{ is bag of string} \)). That is, the entire bag of objects is referenced by its name \( A \), but specific objects in \( A \) are not directly reachable. However, intrinsic functions such as first, last, etc. may be applied to \( A \) which are described in a later section.

One should make the distinction between bags \( A \) and \( B \) in the following variable declaration:
```plaintext
var A is bag of string (10);
B is bag of
begin
  name is string (10);
end;
```

in which A has potentially an infinite cardinality (i.e., A is a dynamic bag) and B is an attributed bag, namely we have the following situation:

```
      A
     /  \
    .    ...
    /    /
   /    /
  .    .
    /
    /
  name
```

If x is a reachable object in bag A, it may be referenced by an attribute name, a label, or both. An attribute name is an identifier of Bagit. A label as defined before has the following general form (Figure 23):

```
      (  )
     /    id
    /      |
   /       j
```

**Figure 23. Syntax of a label**

A dot operator is used uniformly to reference reachable objects of a bag. If an object x in a bag A is identified by an attribute name y,

```
A.y
```

is a reference to the object x. This implies that all attribute names of a bag must be distinct identifiers,
otherwise ambiguity would result.

In case the object x is identified by a label, say (a₁,a₂,...,aₘ), to reference x we write:

\[ A.(b₁,b₂,...,bₘ) \]

where (b₁,b₂,...,bₘ) is called the actual label and every component bₖ, k=1,...,m may stand for one of the following:

1. ? (don't care)
   In this case, ? is a "don't care" operator. That is, the jth attribute of x is not important (for the programmer) to narrow down the search for x.

2. 'v' (literal)
   The quotes represent direct associative referencing and, v is the actual value which is expected to be found in attribute aⱼ of x.

3. v (variable)
   This case indicates that indirect associative referencing is used and v contains the actual value expected to be found in attribute aⱼ of x.

4. n (numeric value)
   This case is similar to (2) except that n is a numeric value expected to be found in aⱼ.

Note that semantically A.(?,?,...,?) is equivalent to A, i.e., the label does not specify any particular object of A and therefore, all objects are referenced.

The general form of an actual label is given below (Figure 24).
Dot Notation

Every application of a dot operator is equivalent to going down the tree structure representing the bag by one level. In other words, the path from A (the root) to the object x must be traversed. Since there is always a unique path from the root to a descendant node in the tree, there is no ambiguity when accessing objects in a bag.

If x itself is a bag nested inside A, apply more dot operators to reference the objects in x. The same accessing rules are recursively applied. The general form for accessing an object in a bag is as follows (Figure 25).

Figure 24. Syntax of an actual label

Figure 25. Accessing the bag objects
Access Abstraction

Access abstraction is achieved in Bagit via associative referencing. The programmer can specify "what" objects are to be accessed rather than code an algorithm to determine "how" the objects are accessed. Therefore, access abstraction hides irrelevant details of searching a data structure for an object.

To sum up the ideas discussed so far, consider the following example. Assume we want to have a small database for a university with various schools, each containing several departments. A typical structure for such a database is given in Figure 26. The representation of each type is given in Figure 27. A snapshot of the 'univ' database then follows in Figure 28.
type --define a faculty type

faculty is bag (last, first, id) of
begin
  last, first is string (20);
  id is bag (ssn) of
  begin
    ssn is integer;
    sex is ('male', 'female');
    age is 0 .. 99;
  end;
  interests is bag of string;
end; --faculty

department is bag (deptname) of
begin
  deptname is string (30);
  chairman is bag of
  begin
    first, last is string (20);
    phone is integer;
  end;
  faculties is bag of faculty;
end;

schools is bag (schoolname) of
begin
  schoolname is string (30);
  budget is real;
  depts is bag of departments;
end;

var univ is bag of schools;

Figure 26. A bag type for a small database
Figure 27. Representation of the bag for small database
Figure 28. The snap-shot of the small database bag

The terminal nodes show the actual values stored in the corresponding attribute (e.g. age of John Doe is 35). All intermediate nodes stand for bags nested inside the parent bags and are identified by an attribute name (e.g. faculties), a label (e.g. (comp sci)), or a combination of both (e.g. id:(123456789)). The root represents the entire bag of university (indicated by univ in Figure 28).

In order to access the age attribute of John Doe, the path from "univ" to the corresponding node for John Doe's age must be traversed. This path is shown by heavy lines in
Figure 28, and is specified as follows:

\[
\text{univ.('college of sci').depts.('comp sci').faculties .('Doe','John',('123456789')).id.age}
\]

Without these labels, the programmer has to search for the right school, department, and faculty member. Only the last associative reference, i.e.,

\[
\text{faculties.('Doe','John',('123456789')).id.age}
\]

has the following equivalent code in a conventional language like Pascal (Figure 29).

\[
\begin{align*}
\text{type} & \quad (* \text{ for faculty only } *) \\
\text{faculty : record} & \\
\text{last,first : string;} & \\
\text{id : record} & \\
\text{ssn : integer;} & \\
\text{sex : (male,female);} & \\
\text{age : 0 .. 99;} & \\
\text{end;} & \\
\text{interest : (* its type *)} & \\
\text{end;} & \\
\text{var faculties : array [1..n] of faculty;} & \\
\end{align*}
\]

\[
\begin{align*}
\text{for } i:=1 \text{ to } n \text{ do} & \\
\text{with faculties [i] do} & \\
\text{if (last='Doe') and} & \\
\text{(first='John') and} & \\
\text{(id.ssn='123456789')} & \\
\text{then } a:=\text{id.age;}
\end{align*}
\]

Figure 29. Simulating associative referencing in Pascal.

Note that this segment of code could be written using a tree structure or some other structure. But the search must be performed one way or another. Also, similar kinds of
loops could be set up for the other two associative references made (i.e., for 'comp sci' and 'college of sci'). The following are a few more references to various objects in univ bag:

a) univ
   -- the entire bag of schools of university
b) univ.('college of sci')
   -- the bag of college of science
c) univ.('college of sci').depts
   -- all departments in college of science
d) univ.('college of sci').depts.('comp sci')
   -- the bag for comp sci department
e) univ.('college of sci').depts.('comp sci').faculties
   -- all faculties in computer science department

**Indirect Associative Referencing**

All the above examples use direct associative referencing. Indirect associative referencing may be used as in the following example:

```
univ.(college_name).depts
```

where college_name is a variable of type string (30) and its content is the string 'college of sci'. This way the programmer can read in different values into the variable and reference different schools.

If a don't care operator ? is used in an actual label, it may result in referencing several objects. For instance,
the reference:

..... faculties.('Doe','John',?)

may refer to more than one John Doe bag in the department. This only happens if there are more than one faculty member in the department with the same name. As a result, any action specified for John Doe, will be applied to all instances of John Doe's bag.

7. The Expressive Power of a Bag

In this section we will show that the expressive power of a bag is equivalent to that of a directed graph (or digraph).

A graph G consists of two sets called vertices V and the edges E. V is finite non-empty set of vertices (sometimes called nodes) usually numbered 1,2,...,n and E is a finite set of pairs of vertices [16].

If the pairs are ordered i.e., the pair (i,j) is different than the pair (j,i), then we call the graph directed (or a digraph), otherwise we call it undirected [16]. Throughout this section we assume that the pair (i,j) is always directed from i to j, i.e.:

\[ i \rightarrow j \]

and is different from (j,i) which is as follows:

\[ j \rightarrow i \]
Equivalence of Bags and Diagraphs

We will show that a bag can simulate a digraph and conversely a digraph can simulate a bag.

1) The structure of a bag as discussed before is a tree where the nodes represent the objects of a bag and edges represent the inclusion relationship between two objects. If object $x$ is in object $y$, the edge $(y,x)$ is directed from $y$ to $x$. This is clearly a digraph.

2) To show that bags can simulate digraphs, we write a general bag definition for a general digraph $G=(V,E)$ where

$$V=\text{set of } 1,2,\ldots,n$$
$$E=\text{set of edges in } G$$

Every node will be labelled with its number, i.e., $1,2,\ldots,n$ and will be accessed via associative referencing. The following is the bag definition for a digraph (Figure 30).

```pascal
type T is 1..n;
node is bag(v) of
  begin
    v is T; end;
arcs is bag(source, dest) of
  begin
    source is node;
    dest is node;
  end;
digraph is bag of
  begin
    nodes is bag of node;
arcs is bag of arc;
  end;
var G is digraph;
```

Figure 30. A bag type for a digraph
The representation of G is given in Figure 31. Therefore digraphs and bags are equivalent structures.

![Diagram of a digraph bag]

Figure 31. Representation of a digraph bag

For example the digraph G of Figure 32 will have the bag representation as in Figure 33.

![Diagram of a digraph example]

Figure 32. An example of a digraph
In the previous few paragraphs we showed that bags and digraphs are equivalent structures. The motivation was to avoid a cumbersome approach to show that for other structures such as arrays, records, pointers, etc. there is a bag definition and consequently show that bags are powerful structures.

All these other structures can be interpreted in terms of digraphs. Therefore, each can be simulated by a bag. We only mention a few.

**Arrays**

An array $A$ is digraph $G=(V,E)$ with an empty set of edges, i.e., $E=\emptyset$. Every node of this digraph represents certain information but there are no edges among the nodes.

**Records**

A record structure is merely a tree and therefore is a
special case of digraph.

**Pointers**

Pointer structures are also special cases of digraphs. Every node contains certain data and may point to zero or more nodes. They have a dynamic nature which is equivalent to dynamic bags of labelled objects.

**Lists**

Finally lists are rooted trees with no isolated nodes. To satisfy our curiosity and to mention two rather important points we will show how bags can directly simulate arrays.

A one dimensional array (or vector) with n elements may be defined as follows:

```pascal
type vector is bag of
  bag (i) of
  begin
    i is 1..n;
    contents is -- e.g. info
  end;
var A is vector;
```

Here, i is the index to the elements of the array and attribute represents the contents of the indexed element. In Pascal the same array is declared as follows:

```pascal
type vector : array[1..n] of info;
var A : vector;
```

The variable A as defined in Bagit has the following structure:
and to access its jth object we write $A.(j).\text{contents}$.

There are two important issues to notice. First, if $A$ is a sparse array, in Bagit we have the option to show only the nonzero elements. Figure 34 shows a snapshot of $A$ in Bagit and the same $A$ in Pascal:

![Diagram of arrays represented by bags]

**Figure 34. Arrays represented by bags**

All elements other than 2, 5, and 100 are zero or not defined, and there is no need to have them in $A$.

The second point is that in many applications, $A$ may contain unique objects. In that case we can redefine $A$ as:

```pascal
type vector is bag of
bag (contents) of
begin
\ i \ is \ 1 \ .. \ n ;
\ contents \ is \ -- \ info
```

```
We can now apply associative referencing rather than writing a piece of code to locate certain data in A.

To complete our discussion we define an n-dimensional array in Bagit and conclude this section:

```pascal
var A is bag of
  bag (I1,I2,...,In) of
  begin
    I1 is L1 .. H1;
    I2 is L2 .. H2;
    ...
    In is Ln .. Hn;
    contents is -- info
  end;
```

The elements of array A may be referenced as A.(J1,J2,...,Jn). This way the programmer can specify a slice of the array using the don't care operator "?". For example, A.(I,?) refers to the Ith row of a matrix A, A.(1,?,?) refers to the front face of cube, and A.(I,?,?,...,?) refers to the Ith slice of the n-dimensional array A.

8. Filters: Information Hiding Mechanism

A filter is the language mechanism for information hiding. It is defined on a bag and consequently, a bag filter becomes a mechanism for defining new access rights (or visibility rights) for referencing the objects encapsulated in a bag.

A filter in general specifies an interface to a data
or type abstraction by naming those attributes of a bag which are visible and may be accessed from outside the bag. Figure 35 shows the general idea of a filter.

![Diagram of data or type abstraction](image)

**Figure 35. The concept of a filter**

As Figure 35 shows, the visibility of objects of an abstraction A is defined by filters $F_1, F_2, ..., F_m$ which are defined in the abstraction. Any access to the objects of A must be made through one of these filters, otherwise a compile time error will result. BagIt guarantees that no object of A is accessed except those specified by the filters. We will discuss this issue in detail in chapter IV.

In this section we are concerned with a special case of a filter which defines a restricted view of a bag (Figure 36). Here, $A'$ is a view of A. It contains all objects of A except those which are filtered out by the filter F. The filtered out objects of A are hidden and not accessible.
Defining Filters in Baqit

The filter $F$ is defined in the variable declaration part of a block. If the filtered view of a bag is declared in a block other than where the bag itself is defined, the language guarantees that no access is made to the bag. Only objects of the filtered view are visible. For instance consider the block structures given in Figure 37.

```
B
  A is bag ...
  A1 is filtered A ...
  B1
    A2 is filtered A ...
  B2
    A3 is filtered A ...
  B3
```

Figure 37. Scope of filtered bags
The different views of A, namely, A1, A2, and A3 are defined in blocks B, B1, and B2 respectively. Although A is a global variable for blocks B1 and B2, the language guarantees that there is no access to objects of A in the scopes of B1 and B2. Only those objects of A made visible by A2 and A3 are accessible in B1 and B2 respectively. However, in the scopes of blocks B3 and B, all objects of A as well as its view A1 are accessible.

Advantages of Filters

There are important advantages in using a filter. First, as we will discuss in the next chapter, a bag and its filters can define data abstractions which can define more than one type. Lack of this feature has been one of the drawbacks in the data abstraction mechanisms of almost all recent programming languages.

Second, filters enhance sharing the same collection of data among several modules. A bag B may be defined in a module and every sub-module has a restricted view of B. Certain information are hidden from a sub-module which may be visible by other modules.

Third, filters can be used to bypass some nodes of the tree structure representing the bag. Assume B1 is a bag nested inside bag A and that Bi is a bag in Bi-1 for i=2,3,...,m. Also assume x is an object in bag Bm and is identified by y (an attribute name or a label).
To access $x$ we must use $m+1$ dot operations, e.g.,

$$\text{old} := A.B_1.B_2. \ldots .B_m.y;$$

This is a rather cumbersome way to access $x$. It is not only tedious but does contain a lot of details. "What" we like to access is object $x$ in the outermost bag $A$. Therefore, $A.y$ is what we need to specify and $A.B_1.B_2. \ldots .B_m.y$ is in fact "how" object $x$ is accessed. Using filters we can resolve this problem (Figure 38). In this case the filter may be used to maintain a pseudo link from the root $A$ to the desired node $y$ (dotted line in Figure 38).

![Figure 38. Use of filters to bypass nodes](image)

The programmer defines a filter, say $A'$, on $A$ to maintain the pseudo link, e.g.,

$A'$ is filtered $A$ of $y$;

As a result, the reference $A.B_1.B_2. \ldots .B_m.y$ becomes
equivalent to $A'.y$. Note that if $A'$ is declared in the scope of a different block where $A$ itself is defined, the names $A$ and $A'$ may be chosen the same, i.e.,

$$A \text{ is filtered } A \text{ of } y;$$

and consequently $A.y$ will be allowed.

Finally, in some cases it is good to be able to change the label of a bag (Figure 40). That is, specifying another combination of the bag attributes to be used as a new label without changing the bag itself. One can then get the objects of a bag in a different order. In this case defining a filter is equivalent to writing a sort routine to sort the objects based on a different key.

8.1 Defining a Filter for a Bag

Filters are defined as attributes of a bag or in the variable declaration part of a block (for defining views of a bag). The bag for which a filter is being defined must have been defined before its filter.

Defining a filter $F$ for a bag $A$ is exactly the same as defining $A$ except that only visible objects of $A$ are specified. The types of these objects need not be given because they are already known. However, if types are specified in the filter, they must be exactly the same as those in the original bag $A$. The general form of a filter is given below (Figure 39).
For example, a filter for the example given in Figure 17 may be declared as follows (Figure 40).

```plaintext
var view_of_Employee is filtered Employee (sex, lastname) of begin
    lastname, firstname, sex, birth, ssn;
    history is bag of begin
        Year, Status;
    end;
end;
```

Figure 40. Defining a view of a bag

In this view of the bag Employee which is called view_of_Employee, the label is changed from (lastname, firstname, ssn) to the new label (sex, lastname).
and consequently, a new ordering is defined on the bag. Note that this time the label is not chosen to be unique. Besides, the Salary and Pos attributes are hidden in this view of the bag.

9. Bag Calculus

In this section, operations defined on bag objects are briefly described. We will assume that all objects used in the examples of this section are of appropriate type. Moreover, we will use square brackets ([ and ]) to denote bags. Therefore, \([a,a,b]\) is a dynamic bag containing two instances of a and one instance of b.

**Union:** 
A+B is the bag containing all objects in A or in B or both. If the bag objects are labelled, the ordering on A+B is defined by the ordering of the labels of A and B. Otherwise, the objects of B would follow those of A. Hence, if \(A=[a,a,c,d]\) and \(B=[b,c,e,e]\), then \(A+B=[a,a,b,c,c,d,e,e]\) if objects of A and B are labelled, and \(A+B=[a,a,c,d,b,c,e,e]\) otherwise.

**Intersection:** 
A*B is the bag containing all the objects in A and in B. Therefore, if \(A=[a,a,b,b,c]\) and \(B=[a,b,b,c,c]\) then \(A*B=[a,b,b,c]\). In this case A*B contains two instances of object b because both A and B contain two instances of b.
Difference: -

A-B is the bag containing all objects in A which are not in B. Therefore, if A=[a,b,b,b,c] then A-[b]=[a,c]. Note that all instances of object b in A are affected by this operation.

Inclusion: in

The value of " x in B " is true if object x is in bag B; false otherwise.

Relational Operators

1. sub-bag: <

A<B is true iff for every instance of every object in A there is a corresponding object in B; false otherwise. Therefore,

[a,b,b] < [a,a,b] is false, and

[a,b] < [a,a,b] is true.

2. super-bag: >

A>B is the same as B<A.

3. equality: =

A=B iff A<B and B<A.

4. inequality: <>

A<>B is true (false) iff A=B is false (true).

Universal Qualifier: all

The expression " all x in B " refers to all objects in bag B and is therefore the same as B itself. It is normally used in conjunction with the suchthat
generator to generate a sub-bag of $B$. The such that generator is explained in chapter IV.

Besides the above operators, there are a few intrinsic functions defined on bags which are as follows:

**First($B$)**
The first object of bag $B$. The ordering of objects is based on the labels of the objects or the first come, first order as described before.

**Last($B$)**
The last object of bag $B$. The ordering is determined as described before.

**Next($B$)**
The successor of object $x$ in bag $B$, where $x$ is the previous object accessed.

**Pred($B$)**
The predecessor of object $x$ in bag $B$, where $x$ is the previous object accessed.

**Current($B$)**
The current object of bag $B$ (i.e., the previous object accessed).

**Delete**
The general form of the delete function is delete($f(B)$), where $f$ is one of the intrinsic functions first, last, next, pred, and current. The delete function is used to remove the object from $B$ which corresponds to one of the above functions.
Therefore, if \( B = [a, a, b, c] \), after the execution of \( \text{delete(first(B))} \), \( B = [a, b, c] \). Note that if the bag difference operation were used, all instances of the first object would be removed from \( B \), i.e., \( B := B - [\text{first}(B)] = [b, c] \). Using the delete function one can remove any object from a bag \( B \) by first locating the object and then applying \( \text{delete(current}(B)) \).

**Size(B)**

It returns the number of objects (not necessarily distinct) in bag \( B \). Note that \( \text{size}(B) \) is the number of immediate sons of \( B \) and not all the descendents of \( B \).

**Empty(B)**

It returns true if \( B \) is empty, false otherwise.

**EOB(B)**

\( \text{EOB}(B) \) returns true if the end of bag \( B \) has been reached (i.e. all objects of \( B \) have been visited), false otherwise.

Once in a while the programmer may need to save the bag containing the results of his/her program. Moreover, he/she may have some input data for the program. In other programming languages there are file structures to handle the external data. In Bagit, a bag and two intrinsic functions are used which are described bellow.

**Open**

The statement
Open (bagname, disk directory name);
associates a disk directory name with a bag name. A value of false is returned if the Open statement fails (e.g., the disk directory name is not found).

Close

The statement

Close (bagname, status);
associates a status with a bag name. The status may be one of the following:

'KEEP' -- to retain the bag after termination of the program.
'DELETE'-- to discard the bag after termination of the program.

Close will return a false value if the function failed.

10. Bag Input / Output

10.1 Screen Input / Output

Bag input/output is screen oriented via bags that are constructed apart from the compiler. For any type bag, we may define a prompt which is kept in a separate bag. The input is then guided by an interpretive scan of this bag to capture the values from the fields defined after the program is compiled. This way it is possible to change the prompt without recompiling the entire program.
Two intrinsic functions "keyin" and "display" will be used to capture input and to display contents of a bag respectively. Their general form is given in Figure 41.

For instance, in Keyin(form.prompt_name, bagname) the identifier bagname identifies a bag to be read in. The prompt associated with the bagname is called prompt_name. The identifier "form" is a bag which contains the prompt. It may contain other prompts too, and programmer should identify which prompt he intends to use (Figure 42).
type person is bag (name, id) of
begin
  name is string (20);
  phone is integer;
  id is bag (ssn) of
  begin
    ssn is integer;
    sex is ('male', 'female');
  end;
end;

var company is bag of person;

keyin (forms.personprompt, company);
display (forms.another_prompt, company);

........................

forms is bag of
begin
  personprompt is prompt (person) of
  begin
    'Name?' is name(20); 'Soc.Sec.\#: ' is id.ssn(9);
    'Phone\#' is phone(10); 'Sex?' is id.sex(5);
  end;
  another_prompt is prompt (person) of
  begin
    ...
  end;
end;

Figure 42. Prompts

It is important to state that the spacing and placement of these prompts on the screen will be identical to the placement of this information on the screen when it is entered into forms.personprompt. Moreover, the numbers enclosed in parenthesis, e.g. in name(20), specify the field length of the associated attribute. As soon as keyin
statement is executed, a person prompt will be displayed on the screen and the programmer can fill out the spaces which are provided by the prompt (Figure 43).

Name?----------------- Soc.Sec.#:----------
Phone#------------ Sex?-----

Screen: before
Name? John Doe Soc.Sec.#:123456789
Phone# 757-9878 Sex? male

Screen: after

Figure 43. Screen I/O
As long as it is not the end of input bag (e.g., EOB), another prompt is displayed on the screen and should be filled out. The general form of a prompt is given in Figure 44 below.

Figure 44. Syntax of a prompt

10.2 Conventional I/O

Besides the screen I/O, Bagit provides conventional I/O facilities similar to other modern programming languages. Two other intrinsic functions "input" and "output" are introduced to read in and print out objects of input a bag is to read each of its attributes as follows:
input ( bagname.attributename);
output( bagname.attributename);

The general form of input and output statements are as follows:

In case that a bag is associated with a permanent bag by an Open statement, e.g. Open(bagname,diskbag), then we can write the statement input(bagname) which reads in objects of the bag identified by bagname as a whole from the permanent bag named diskbag. Otherwise, if a bag is not associated with a permanent bag, the standard input (which is a pseudo bag) is assumed and the objects must be of a built-in type.
IV. The Operational Environment

1. Assignment Statement and Expressions

An assignment statement has the form $V := E$; where $V$ is a variable identifying some object, $E$ is an expression, and $:=$ is the assignment operator. The expression $E$ is evaluated and the result is stored in the variable $V$. Both $V$ and the value of $E$ must be of the same type.

Any Bagit expression is an abstraction of a real world action. Expressions are the basic syntactic building blocks from which statements are built. They consist of operands and operators. In previous chapters we discussed bag operators as well as arithmetic operators of Bagit. The relational operators of the language result in a boolean value and are as follows:

\[
\begin{align*}
&= \quad \text{-- equality} \\
&<> \quad \text{-- inequality} \\
&< \quad \text{-- less than} \\
&> \quad \text{-- greater than} \\
&\leq \quad \text{-- less than or equal} \\
&\geq \quad \text{-- greater than or equal}
\end{align*}
\]

The logical operators are the conventional operators AND, OR, and NOT.

In Bagit, any expression which will be evaluated to a
value of type T, is called a T-expression.

**Qualified Reference**

In a bag-expression, a suchthat generator may be used to qualify specific objects. For instance, if Employee is a bag of objects of type employees (Figure 17), the bag-expression:

```
all x in Employee suchthat x.sex='male';
```

would result in a bag of all male employees (if any).

**Range Generator**

Another bag generator is the range (..) generator which generates a sub-bag of another bag. For instance the bag expression:

```
Employee.('Doe','John',?) .. Employee.('Foe','Joe',?)
```

generates all employees whose names lay between the two given bounds, i.e. John Doe and Joe Foe.

These two generators (which are built-in generators of the language) will be discussed in a later section.

**Operands**

The basic operands in an expression are constants and variables. More complex operands may be formed using operators and parenthesis as in other languages. It suffices to look at the following syntax diagram for expressions (Figure 45). The operator precedence is also determined by this diagram.
Figure 45. Expressions
2. Control Structure

In this section we briefly explain Bagit control structures as well as bag generators.

2.1 Sequence

Sequence is merely a group of statements (or basic actions) of the form:

statement; statement; .......

A compound statement is formed by enclosing a group of statements in a begin-end block.

2.2 Conditional: if-then-then-else

The decision making capability of Bagit is supported by a single construct called if-then-then-else. The general form of it is as given in Figure 46.

```
if \[bool\] expr then statement else statement
```

Figure 46. if-then-then-else syntax

Therefore all of the following statements are valid:

```
if condition then statement;
if condition then statement
  else statement;
if condition1 then statement1;
  condition2 then statement2;
  ....
  conditionk then statementk
  else statement;
```
As soon as one of the conditions is satisfied, i.e., the value of the boolean expression is true, the corresponding statement will be executed and control leaves the block. Otherwise, if none of the conditions are satisfied, the statement corresponding to else will be carried on and then control leaves the block.

The if-then-then-else construct combines the case and if-then-else constructs of other modern programming languages.

2.3 Iterative Statements

2.3.1 Forall-do

The most natural looping construct is one which enables the programmer to visit every object of a bag. Bagit provides a forall construct which has the following form:

forall x in B do S;

where:

- *x* is a declared variable whose type is the same as the type of objects of the bag *B*,
- *B* is either a predefined bag or a bag generator, and
- *S* is a statement or block of statements called the body of the loop.

If *B* is a predefined bag, it may be viewed as the identity bag generator which returns (or generates) all the
objects in the bag. The generator will produce objects of B and x will assume the value of these objects one at a time. The body of the loop in turn will consume the generated objects. The loop is terminated whenever all objects of B are visited, i.e., $EOB(B)$ is true. The syntax diagram of the construct is given in Figure 47.

![Figure 47. Forall-do statement](image)

2.3.2 Until-do

The second looping construct provided in Bagit enables the programmer to set up loops controlled by a condition. The statement:

```plaintext
until condition do S;
```

tests the condition first and executes the statement $S$ as long as the condition is false. Therefore, as control leaves the loop the condition has a value of true. This statement has the following syntax (Figure 48).

![Figure 48. Until-do statement](image)

Note that the until-do construct can easily simulate the forall-do statement. That is, the statement:
forall x in B do S;

may be written as:

\[
x := \text{first}(B); \\
\text{until } \text{EOB}(B) \text{ do} \\
\text{begin} \\
\quad S; \\
\quad x := \text{next}(B); \\
\text{end};
\]

2.4 The exit statement

The exit statement simply specifies transfer of control from one point in the program to another. In Bagit, the exit statement is used to leave one or several enclosing blocks. A block in this case can be a begin-end, an if-then-then-else, a forall-do, or an until-do statement.

In order to use it to leave several nested blocks, one should name (or label) the block from which the control is to be transferred. For instance in the following block structure:
the **exit** A statement transfers control out of the block named A to the first statement following the block. The block name is an identifier of Bagit.

If the block name is not used in the exit statement, the control transfers out of the innermost enclosing block, namely:

![Diagram showing control flow](image)

### 2.5 The in Statement

In order to access and manipulate objects of a bag, the programmer should use a dot operation. When there are several bags nested inside one another, this notation becomes tedious. The in statement conceptually corresponds to the *with* statement of Pascal. Semantically, every use of an in operator opens a bag so that all objects in the bag may be directly accessed without preceding the attribute name by a dot and the bag name. Therefore, instead of:

```
A.x := somexpression;
```

we can write:

```plaintext
in A do x := somexpression;
```
This notation is more readable when several such objects in A are to be manipulated. The syntax diagram is as follows:

```
  in  variable  do  statement
```

3. Functions

3.1 Functional Abstraction

A major goal of abstraction is the separation of use from implementation. What we desire from an abstraction is a mechanism which permits the expression of relevant details and the suppression of irrelevant details [24]. One of the major contributions of conventional programming languages is providing functional abstraction. The complexity of computation may be decomposed around procedures and functions used in a program. A programmer using the SQRT function to evaluate the square root of a number is only concerned about what the function does. He/She is not concerned about how it has been implemented. In other words, the use of a function or procedure is a relevant detail but its implementation is irrelevant to the actual problem.

Essentially, the major difference between procedures and functions is that a function evaluates a single value and may be called in an expression, where a procedure may return zero or more values and should be invoked by a
special call statement. The value returned by a function is of a built-in type. In Bagit, there is no need for a procedure because functions are allowed to return objects of any type, specifically of type bag. They may be called within an expression or as a separate statement.

What the programmer should be concerned about is the process of logic abstraction by separating a computational task in a function which may result in zero or more values, from the usage of these values. How these values are returned is irrelevant to the computation.

If the computation involves more than one value, the function would return a bag of values and then each value may be accessed in the normal manner using a dot operator.

The methods of parameter passing in Bagit are pass-by-reference (or address) and pass-by-value, as defined in programming language Pascal. If the parameter is preceded by a var, it is pass-by-reference, otherwise by value.

3.2 Function Constructor

Functions of Bagit have the following general form:

```
function_name is function (formal parameter) of T;
local declarations
begin
    statements....
end;
```
Here, if no type (i.e. \( T \)) is specified for the function, the function does not return any value. \( T \) is the type of the object returned by the function, identified by the function name as its name.

An invocation of this function has the following form:

\[
\text{function name (actual arguments)}
\]

which may be used in any expression.

A function is defined in the variable declaration part of a block. It will be treated as a variable of type function. That is why a function constructor is sometimes called a pseudo-type in Bagit.

The following is a simple example of a function which computes \( a+b, a-b, a*b, \) and \((a \div b)\) for its two arguments \( a \) and \( b \):

```plaintext
type basic_ops is bag of 
  begin
    add, sub, mul, dvd is integer;
  end;
var F is function (a,b is integer) of basic_ops;
begin
  F.add := a+b;    F.sub := a-b;
  F.mul := a*b;    F.dvd := a \div b;
end;
```

The function may be invoked as:

\[
F(c,d); \quad -- \text{just like a procedure call}
\]

which returns a bag of four objects identified by \( F(c,d) \).

It may be used in an expression too:

\[
h := F(c,d).sub +1;
\]
where an attribute of $F$ is selected via a dot operation.

Note that the function $F$ is a compact operator in the sense that standing alone it represents four different operators $+,-,\times,$ and $\text{div}$.

Furthermore, the same function constructor is used for overloading operators, i.e., extending the meaning of an operator to operands of a new type. For instance the following function defines both addition and multiplication of complex numbers in a single function:

\begin{verbatim}
type complex is bag of
  begin
    re is integer;
    im is integer;
  end;
ops is bag of
  begin
    add, mul is complex;
  end;
var complex_ops is
function (var x,y is complex) of ops;
beg
  in complex_ops do
  begin
    add.re := x.re + y.re;
    add.im := x.im + y.im;
    mul.re := x.re*y.re - x.im*y.im;
    mul.im := x.re*y.im + x.im*y.re;
  end;
end;
\end{verbatim}

3.3 Generic Functions

In some recent programming languages a facility for making generic definitions is provided in order to allow the programmer to write a single textual definition that serves as an abbreviation for many closely related specific
definitions. The main computation in these definitions are the same but the types of operands change from one to another. For example consider a procedure to swap the contents of two variables of type T1 and another for variables of type T2. In most modern programming languages (e.g. Pascal), one has to write two different procedures which are exactly the same except for the types of the arguments.

The primary purpose of a generic routine is therefore to factorize the computation (e.g. swapping) and pass the types of the operands when needed. This would enhance readability and efficiency as well as reduce the size of the program.

In Bagit such a facility is provided through two constructs: oneof and type. The oneof construct defines alternative types for an object. It may only be used in generic functions or generic abstract data types (which are explained shortly). Therefore, the clause:

\[
x \text{ is oneof (T1,T2,T3);}
\]

specifies that object x may be of type T1, T2, or T3.

The type function returns the type of its argument. It may be used in variable declaration part to define the type of a variable. Therefore, if object x is of type T, the declarations:
\[
\text{var } y \text{ is type}(x);
\]

and \[
\text{var } y \text{ is } T;
\]

are equivalent.

The generic functions of Bagit are syntactically defined as ordinary functions. No new keyword distinguishes them from ordinary functions.

To conclude this section, a generic function for swapping two variables (which may be of type integer, real, or some predefined type T) is given:

\[
\text{swap is function(var } x, y \text{ is oneof(integer, real, T)) of ;}
\]

\[
\begin{align*}
\text{var } & \text{ temp is type}(x); \\
\text{begin } & \text{temp :=x;}
\quad x :=y; \\
& y :=\text{temp; }
\end{align*}
\]

The type of the actual arguments \(z\) and \(w\) in the invocation \(\text{swap}(z,w)\) must be integer, real, or T. Any other type would result in a compile time error.

4. Generators of Bagit and Control Abstraction

We mentioned two generators (\texttt{suchthat} and \texttt{range}) in section 1 of this chapter. In this section we will explain the generators in more depth and compare them with the iterators of the programming language Clu [25].

A generator, built-in or user-defined, will produce objects of an abstract bag. The purpose of having generators is to provide control abstraction by hiding irrelevant details and separating the use of objects of a
bag from the selection of the objects.

Bagit generators may be used in expressions as described before, or in conjunction with the looping constructs.

4.1 The Iterators of Clu

The iterators of Clu [25] are used in conjunction with the for statement. The Clu for statement can iterate over collection of any type of object. They produce the objects in the collection one at a time; each object is consumed by the for statement in turn [25]. The following is an example given by Liskov in [25]:

```plaintext
count_numeric = proc (s:string) returns ( int );
    count : int :=0;
    for c:char in string_char(s) do
        if char_is_numeric (c)
            then count:=count+1;
            end;
        end;
    return (count);
end;
```

```plaintext
string_char = iter (s:string) yields (char);
    index:int :=1;
    limit:int :=string$size(s);
    while index <= limit do
        yield (string$fetch (s,index));
        index:=index+1;
        end;
end;
```

The `string_char` is an iterator defined by the programmer. It produces the characters in a string in the order in which they appear. The for loop (in the procedure
count_numeric) initially invokes the iterator, passing it some string. Each time a yield statement is executed in the iterator, the character yielded is assigned to the variable \( c \) declared in the for statement and the body of the for statement is executed. Then the iterator is resumed at the statement following the yield statement in the same environment as when the character was yielded. The procedure counts the number of numeric characters in the string \( s \), using a function called char_is_numeric.

**Problems of Clu's Iterator**

There are some problems with Clu's iterators which are briefly discussed using the example given above.

First, the iterators are invoked by for statements. If two for statements are nested and both invoke the same iterator, it is not clear how the iterator would interact with both of the for statements. In other words, the way an iterator is defined in Clu dictates that in the above situation the iterator should remember which for statement has invoked the iterator, or otherwise ambiguity would result. To be more specific, consider the following situation which uses the same iterator string_char of the previous example:

```plaintext
for c:char in string_char (s) do
  ....
  
for d:char in string_char (s) do
  ....
```
When the first for loop invokes the iterator for the first time, the iterator yields the first character of string s which will be assigned to variable c. Next, the second for loop invokes the same iterator. It is not clear whether variable d is assigned the second character of string s or the first. So, the question is how an iterator is reset and then reused.

Second, the same problem arises if an iterator is written recursively either by invoking itself or by invoking another iterator which in turn invokes the first one. In this case we have the following situation:

```plaintext
for ... in I1 ... do
  for ... in I2 ... do
    end;
  end;
end;
```

where:

```plaintext
I1 = iter ... :

for ... in I2 ... do
  end I1;
I2 = iter ... :

for ... in I1 ... do
  end I2;
```
The first for loop invokes iterator I1 which in turn invokes I2, and I2 invokes I1. Therefore, the relationship between the two iterators is:

\[ \text{---} \rightarrow \text{---} \rightarrow \text{---} \]

\[ \text{for} \quad I1 \quad I2 \quad I1 \]

\[ \text{---} \leftarrow \text{---} \leftarrow \text{---} \]

where \( \rightarrow \) shows the invoking relation and \( \leftarrow \) indicates the yielding relation. When the second for loop invokes I2 we have:

\[ \text{---} \rightarrow \text{---} \rightarrow \text{---} \]

\[ \text{for} \quad I2 \quad I1 \quad I2 \]

\[ \text{---} \leftarrow \text{---} \leftarrow \text{---} \]

The second time I1 or I2 are invoked, the same problem of interaction between the iterators and their associated for loops exists.

Third, if there are two yield statements in the iterator, it is not clear where the iterator resumes the process. This makes the proof of correctness hard and sometimes doubtful.

Finally, this new feature as well as the keywords \text{iter}, \text{yield}, and \text{yields} are added to the language. The translator should handle them and the execution will be slowed down by transferring control between a for statement and the iterator back and forth. It adds to the size and complexity of the language.

In the next section an alternative solution is given
in Bagit.

4.2 The Built-in Generators of Bagit

4.2.1 The Range Generator: .. (dot dot)

.. is a built-in generator of Bagit which generates the values in the range of two given variables. The form X..Y specifies two values X and Y of the same type. They may be constants of an enumerable built-in type or actual labels of some objects in a bag. If X<Y, it produces:

X, next(X), next(next(X)),..., and Y.

Otherwise, if X|Y, it produces:

X, pred(X), pred(pred(X)),..., and Y.

For example:

1..n generates 1,2,3,...,n
n..1 generates n,n-1,n-2,...,2,1 and

Employee.("Doe","John",?)..Employee.("Smith","Bob",?)

which generates all objects in the bag Employee whose labels are in the range of John Doe and Bob Smith (on the last name). The result is an implied bag containing all the objects in the specified range and the ordering is the same as that of Employee.

Bag Former

A bag former is used to construct dynamic bags from objects of a standard built-in type. All objects should be of the same type. If p is an object of type t, then [p] is
A bag containing object p.

A bag former may be used to form more complex bag generators, for example:

```
['0'..'9', 'A'..'Z']
```

is a dynamic bag containing '0','1','...','9','A','...','Z' in that order. Therefore, ['0'..'9', 'A'..'Z'] and ['A'..'Z', '0'..'9'] are two dynamic bags containing the same objects but with different orderings. However, '0'..'9' and ['0'..'9'] are the same bags.

The bag former may also be used in bag expressions where all operands should be of type bag. For instance if S is declared as bag of integer and j as integer, then:

```
S := S + j;
```

is invalid (types are not compatible) and should be written as:

```
S := S + [j];
```

4.2.2 The Qualifier SUCHTHAT

suchthat is another built-in generator of Bagit which generates qualified objects of a given bag. It is usually used to define a sub-bag by selecting those objects which satisfy a qualification. For example, the expression:

```
all x in 1..n suchthat x mod 2=0
```

generates all even integers in the range of 1 to n.

It may also be used in conjunction with a forall statement as in the following example:
forall x in Employee suchthat x.Pos='manager' do ...
which would perform an action on all the managers of a
company represented by the bag Employee.

It is possible to apply a combination of both built-in
generators as well as bag operators in a single statement.
For example, assume cs511 and cs415 are two bags of objects
of type student, then the statement:

forall students in cs511*cs415 suchthat
    student.age in 20..30 do action;

performs the action for all students in cs511 and cs415
whose age (an integer number) is in the range of 20 and 30.

4.3 User-defined Generators

The main purpose of a generator is to provide control
abstraction, by hiding and sometimes separating the process
of producing the objects to be consumed in a computation.
This would give the programmer the ability to iterate over
a bag of abstract objects. The two built-in generators
described in the previous section both hide the next object
selection so that the programmer would only specify "what"
objects to be manipulated, and not "how" these are
generated.

In some cases we need more flexibility, the objects
may not belong to certain range of values, or may not be
qualified objects of another bag. Still, what we need is a
way to produce these objects prior to the consumption process.

The functions of Bagit are allowed to return values of any type, particularly values of type bag. The programmer can write a function which when called would generate the required objects and return them back to where the function was invoked.

Unlike the Clu's iterators, the user-defined generators of Bagit (which are nothing but ordinary functions) can be used anywhere in the program and are not restricted to for loops. The general form of a function used in a forall loop is:

```plaintext
forall x in F(arguments) do ...
```

where F is a function invoked in the loop and its implementation is independent of the actions to be performed on its objects.

Moreover, unlike Clu, Bagit does not introduce a new feature and new keywords into the language. All it does is generalize functions by permitting them to return values of type bag.

Finally, the generator defined by a function, can be written recursively without any ambiguity in syntax or semantics.

To conclude this section, we give an alternative solution for the problem of counting numeric characters in a string s, discussed in section 4.1 of this chapter. The
solution in Clu used an iterator as given before. The corresponding solution in Bagit is as follows:

```plaintext
type char is string(1);
var count is integer;
c is char;
chars is function(s is string) of bag of char;
begin
  if length(s) = 1 then chars := [s]
  else chars := [substr(s, 1, 1)] +
              chars[substr(s, 2, length(s) - 1)];
end;

begin
  count := 0;
  forall c in chars(s) suchthat c in '0'..'9' do
    count := count + 1;
end;
```

Here, `chars` is a function which returns a bag of characters of its argument `s`. The ordering is the same as ordering in the string itself. The `forall` statement uses `chars` as a generator and combines the two built-in generators of Bagit (i.e., `suchthat` and `..`) to count number of numeric characters of string `s`.

5. Abstract Data Types

One of the most important contributions to the field of programming languages has been the concept of abstract data types (ADT). Over the past several years the concepts of "data types" and "data structures" have gradually merged into a single concept known today as abstract data types (or equivalently abstract data structures) [9].
Virtually all languages support abstract structures to some extent. For example an array structure abstracts a block of memory locations and index registers for selecting a specific location.

The realization of data abstraction in programming languages is a fairly complex process. The view that abstract structures should be manipulated by only operations defined specifically for them is fairly recent [9]. In a sense, abstract data types encapsulate an abstract structure and operations on it in a single place. From this point of view, an array with selection and assignment of individual elements may be considered as a very primitive form of abstract data types.

The development of the concepts embodied by data abstraction started to achieve practical importance in the early 1970's, largely as a result of the attention that ideas on program structuring had received a few years earlier [9,31].

The subject was originally referred to as "protected data access" and "protected data objects" in the 60's, and later as Information hiding. There have been a number of attempts to incorporate these ideas on abstract types into programming languages, either by extending existing languages or designing new languages [9]. In most of the recent languages, the information hiding and limited access to data objects is guaranteed by the language via a
linguistic construct.

These languages can be divided into languages that are based on abstract structures and those supporting programmer defined abstractions. The first group includes languages like Lisp [28] with abstract tree structures, and SETL [39] which is based on abstract set structure. The remaining languages, Simula [4,32], Clu [24,25,45], Alphard [40], ELL [41], Euclid [1,3], Modula [1,44], Mesa [10], and Ada [42] all support some notion of programmer defined abstraction. The languages of this type that are currently receiving the most attention are Simula, Clu, and Ada.

In an attempt to free users and programmers from the details of conventional computer systems, work has proceeded in two directions [23]:

(1) in the language area, there have been some investigations to allow more natural operations over types of data that more closely resemble real-world objects, and

(2) in the database area, there have been some investigations to promote more natural methods for manipulating large databases.

In addition to the two areas mentioned above, research has been conducted in the computer architecture area. Intel's "iAPX432" architecture [17] is the first commercial system to support an object-oriented programming
methodology. Abstract data types are supported by hardware recognized, hardware protected, and hardware manipulated structures [17].

A fundamental focus of much recent research in the database systems area has been concerned with database abstraction [29]. In the context of database systems, data abstraction refers to physical data independence, i.e., separating the physical representation of data from its meaning. For instance, a "date" object is represented by a number, but certainly dates are not numbers. Taking the cosine of a "date" is a meaningless operation which should not be allowed.

The issue is that neither the language area, nor the database area can usefully operate without the other. Moreover, both fields need the appropriate environment supported by the computer architecture. Programming languages must eventually be extended to deal with complex data abstractions that exist in the real world. Database systems must begin to characterize transactions in terms of application oriented operations in their programs, e.g., a database operation to "hire" an employee rather than "inserting" an employee record into a database [23].

It will be out of the scope of this thesis to go beyond what has already been said about data abstraction research in databases. For a detailed discussion see [2,20,30].
In the next few sections we will discuss abstract data types from a programming language point of view, their rationale, history, etc.

5.1 The Definition of an Abstract Data Type

Two views of the role of type in a programming language have been discussed in the literature [33,36].

In the first view, a type is defined as a set of values. This view is embedded in Pascal, PL/1, etc. A data type is defined by specifying the representation for the values. Operations on the values of a type are coded as procedures but they are independent of the data type. Languages employing this view of type do not provide adequate protection to insure that only meaningful abstract operations are performed on a value which represents an abstract entity [36]. For example, if an array is used to implement a stack object, any element of the stack can be changed because any element of an array can be reassigned. In other words, operations on the representation of data type (e.g. array) can be performed on the abstract data type (e.g. stack) even though those operations are not semantically valid on the abstract type [36]. The language lacks a suitable provision for limiting access to data.

In the second view, a type is a language mechanism to enforce authentication and security [36]. The language guarantees that all operations applied to a value are
meaningful for that type. This definition of type is embedded in several new programming languages. Most researchers in the field of programming languages now agree that an abstract data type consists of a set of objects and a set of operations characterizing the behavior of the objects [14,25,36,40]. Languages allowing user defined data abstraction provide a language construct to encapsulate objects and operations defined on them in a single place. Although these linguistic constructs have different names in different languages, they practically serve the same purpose. The following are some of these constructs and the corresponding languages:

- **cluster** Clu
- **class** Simula
- **form** Alphard
- **package** Ada
- **module** Modula
- **module** Euclid
- **module** Mesa

5.2 From Simula To Clu

The Simula programming language [32] is the first language credited with the first implementation of a primitive form of an abstract data type almost two decades ago.
The original specification of a class in Simula did not meet the stringent information hiding requirement that is now considered to be a crucial feature in any data abstraction mechanism [31]. Namely, the entities declared in a class can be accessed directly without using the class operations. Euclid's modules also suffer from the same problem [3].

Among the newly designed languages which truly implement a data abstraction are Clu, Ada, and Alphard. Because of the general attention paid to the Clu programming language, we will focus on this language in our discussions.

The data abstraction mechanism in Clu is called a cluster. It permits a data abstraction to be implemented as a single unit by encapsulating a representation for data object and defining an algorithm for each operation in terms of that representation. Access to the representation is allowed only through the operations in the cluster and is controlled by type checking at compile time.

The following is an example in Clu, defining an abstract data type for a generic stack object:

```
stack = cluster [t:type] is
  push,pop,top,empty,create;
  rep = record [ tp : integer,
                 stk: array [1..n] of t ];
  push= proc (s:cvt; v:t);
      if s.tp = n then & raise overflow
      else s.tp := s.tp +1;
```
s.stk[s.tp]:=v;
end;
end;

pop = proc code which removes the top element
  end
end

top = proc code which fetches the top element
  end
end

empty = proc code which checks if stack is empty
  end
end

create = proc code to allocate storage necessary
     to manage a new instance
    end
end stack;

Operations visible from outside the cluster are listed after the keyword is. The reserved type identifier rep indicates that the type specification to the right of the equal sign is the representing type for the stack cluster. All operations valid on the representation are then defined in the cluster. We have only defined the push operation and the rest of the operations are similar. No access may be made to rep unless through one of the operations pop, push, top, empty, and create. The keyword cvt is used only inside the cluster and indicates a conversion of viewpoint between the external abstract type and the internal representation type. In other words, inside the cluster it is necessary to view a stack object as being of the representation type, simply because the implementation of the operations are defined in terms of the representation.

In other languages such as Alphard, Modula, etc. the data abstraction has the same general form as discussed above. In Bagit, the data abstraction is rather different. In section 5.4 the same stack abstract data type in Bagit
is given.

5.3 The Rationale of ADT

Horning [15] has made an excellent comparison between procedural abstraction and data abstraction. Although there is a clear distinction between these two kinds of abstraction, he argues that data abstraction has at least the same advantages achieved by procedural abstraction. Unfortunately, the nature of the abstractions that may be conveniently achieved through the use of procedures is limited [14].

We can relate the necessity of data abstraction in a programming language to several issues: reducing overall complexity of software systems, security, verifiability and reliability, independence, and modularity.

In today's application programming, the programmer is faced with very large and complicated data. In any problem there is a natural hierarchy of abstractions. Through the process of data abstraction the complexity of data can be reduced by breaking it into levels of abstractions. The mapping from the conceptual levels of abstraction to some physical levels of abstraction can be much simplified in a language supporting data abstraction. All information about a type definition (representation, operations and their implementation) are gathered in a single place. The overall complexity may be then decomposed around these data
abstractions. Moreover, the repetition of several definitions for similar units can be avoided.

Security

Security may be one of the most important aspects of data abstraction. The data is not only complicated, but is sensitive. They range from bank accounts and credit cards to space programs. Security and protected data in these systems is a "must". It is based on information hiding and limited access to data objects. Even in a single program, some modules may have access to a data area which must be hidden from others. Certainly a data abstraction process enhances data protection. The language should guarantee limited access to the objects in the abstract object.

Correctness

In languages supporting abstract data types, verifiability and reliability will be improved because they permit proofs of correctness to be decomposed around the type definition [26]. An abstract data type separates implementation from specification. After one proves that the implementation truly models the specification, throughout the rest of the program the proof of correctness only depends on the abstract behavior of the type.

Reliability

Also, if for some reason (e.g. efficiency check) the representation of abstract object or the implementation of operations defined on the object are changed, the modules
using the abstract object will not change. The use of an abstract data type is independent of its implementation. For example, the fact that a stack abstract data type is represented by an array or a linked list should not concern the programmer at the time of using the abstraction.

Structure

Finally, data abstraction enhances modularity and supports structuralism. A large software program would be difficult for one person to complete. In large projects, different modules are written by different programmers. It should be possible for different modules to interact easily. Moreover, a change in one module should not affect other modules drastically. Data abstraction enhances this idea by encapsulating the implementation details in a single place and allowing the modules to interact via abstract specification of objects.

5.4 Data and Type Abstraction in Bagit

Most articles on abstract data types ignore the more general issues of information hiding and visibility rules. Semantically, a data abstraction is a special case of a general rule for defining scope.

Irrelevance is a relative concept. Representation of an abstract object is an irrelevant detail outside the scope of the abstract type, but is relevant to the operations defined inside the abstraction. Moreover, an
abstract data type itself may be irrelevant to one module and relevant to another.

The programming language Bagit generalizes the accessability and visibility of objects by providing filters. Bags provide the necessary encapsulation in the language. Filters define the visibility of objects in the bag. In a sense, we have already stated the data abstraction capabilities of Bagit without calling it an abstract data type.

A bag can contain objects of type function as well as any other type. The filters are defined as attributes of the bag and specify the visibility of these objects. Figure 35 (section 8, chapter III) shows the concept of a data or type abstraction in Bagit. A generic definition of an abstract data type for a stack object is given in Figure 49 (the corresponding definition in Clu was defined in section 5.2 of this chapter).
type stack is
  bag [T is oneof(integer, P, string)] of
  -- P is a user-defined type
begin
  oneview is filtered stack of
    begin push, pop, top end;
  stk is bag of T;
push is function (newitem is T) of;
    begin
      if size(stk) >= limit then overflow
      -- overflow is an exception, may be
      -- a function which is invoked
        else stk:=stk+[newitem];
      end;
pop is function of;
    begin .... end;
top is function of T;
    begin .... end;
end;

Figure 49. An ADT in Bagit
Here, stack is the name of the data abstraction. There is one filter defined in the abstraction, namely "oneview", which makes operations push, pop, and top visible outside the scope of the encapsulation. The definition is generic in the sense that several stack objects of different element types may be declared and used as follows:

```haskell
var int_stack is stack[integer];
P_stack is stack[P];
begin
  int_stack.push(5);
  x:=int_stack.top;
  etc.
end.
```

Note that in the corresponding definition in Clu (section 5.2) the push operation gives the appearance of two parameters, i.e., push=proc(x:cvt,v:T), and at the time of use it has only one argument, e.g. int_stack$push(5). This may be confusing (at least in form). In the example of stack in Bagit this is not the case.

As discussed earlier, in most recent programming languages supporting abstract data types, a data abstraction corresponds to a single abstract data type. There are circumstances where we like an abstraction to define more than one type [35]. This problem is resolved in Bagit via type abstraction.
Type abstraction in Bagit is accomplished by simply allowing a data abstraction to have more than one filter, and consequently define more than one type. Figure 50 shows a generic type abstraction for a stack. Every filter in this case corresponds to a new type of stack with a different behavior.

```
gen stack is
  type stack is
    bag [T is oneof(integer, p, string),
          limit is integer] of
    begin
      myview is filtered stack of
        begin push, pip end;
      yourview is filtered stack of
        begin push, pop, top end;
      stk is bag of T; -- representation
      push is function( newitem is T) of
        begin
          if size(stk) >= limit then overflow
            else stk := stk + [newitem];
        end;
      pop is function of
        begin
          if empty(stk) then underflow
            else delete(last(stk));
            -- remove the last element pushed
        end;
      top is function of t;
        begin
          if empty(stk) then underflow
            else top := last(stk);
        end;
      pip is function of T;
        begin
          if empty(stk) then underflow
            else begin
              pip := last(stk);
              delete(last(stk));
            end;
        end;
    end; -- stack
```

Figure 50. A generic type abstraction in Bagit
Different stacks may be defined as follows:

```verbatim
var  int_stack  is  stack[integer,20].myview;
    str_stack  is  stack[string ,35].yourview;
begin
    int_stack.push(5);
    x:=int_stack.pip;
    str_stack.push('first');
    y:=str_stack.top;
    str_stack.pop;
end;
```

In this example the filters are called myview and yourview. The declaration of a stack object should specify what filter is used (i.e., what type of stack is meant), e.g. the declaration for int_stack specifies an object of type stack whose elements are all of type integer, with at most 20 elements allowed on the stack. Moreover, the filter 'myview' makes the operations push and pip visible and hides the others. The second filter, called 'yourview', makes the operations push, pop, and top visible and hides the rest of the attributes. Note that the two stacks as declared are two different data types, with two different behaviors, and of course different types of elements. In the first one (i.e. int_stack), both operations pop and top are performed in a single operation called pip. Therefore, the statement x:=int_stack.pip; would assign the top element of stack to y and then removes it from the stack. However, in the second type of stack (i.e. str_stack), two separate operations are needed and as a result two statements y:=str_stack.top and str_stack.pop are written to do the job. This shows that the abstraction defines two
types of stack with two different behaviors.

Another problem, the so-called zero type problem, is identified by Rowe [35] as a problem with abstraction mechanism of existing languages. This problem relates to circumstances where the data abstraction defines no type (or zero type), i.e., it is just being used to collect together a bunch of procedures or routines in a subroutine library [35].

In Bagit, a zero type may be defined using the bag filters just like any other kind of abstraction. The only difference is that all attributes of the bag are of type function. For instance consider the following example (Figure 51). It's a zero type abstraction by which two different zero types are defined. Those who are given the 'manager' view of the abstraction can access all functions of the encapsulation, e.g. by the declaration:

```plaintext
var mangr is private_library.manager;
```

Those with a programmer view of the type can only access the functions `list1` and `list2`, and not the rest of the functions. That is, unauthorized persons cannot hire, fire, or update an employee, but only prepare certain kinds of listings of the employee bag. For instance:

```plaintext
var prgmr is private_library.programmer;
```

declares the programmer view of the abstraction which is only allowed to do the following:
prgmr.list1 (employee);

or

prgmr.list2 (employee);

and nothing else. On the other hand, an authorized person or module who has (or knows!) the manager's view of the type is allowed to do any one of the following:

mangr.employ (employee, newperson);
mangr.fire (employee, oldperson);
mangr.update (employee, myself, mysalary);
mangr.list1 (employee);
mangr.list2 (employee);

These views may be defined by the supervisor of the project and submitted to different programmers. Of course there is no way to protect the views against malicious programmers. The language only guarantees that if the views are declared legally, during the execution there is no way to access the objects from unauthorized views.
type private_library is bag of
begin
manager is filtered private_library of
begin employ, fire, update, list1, list2
end;

programmer is filtered private_library of
begin list1, list2 end;

employ is function (employee_bag, new_employee) of;
-- code to employ an employee

fire is function (employee_bag, old_employee) of;
-- code to fire an employee

update is function (employee_bag, key, attr) of;
-- code to update an attribute
-- of an employee, e.g. salary

list1 is function (employee_bag) of;
-- output the employee bag

list2 is function (employee_bag) of;
-- input the employee bag
end;

Figure 51. A zero type example
V. The Runtime Environment

In this chapter we briefly discuss some implementation issues of Bagit. A top-down parsing technique (LL(1)) is suggested which requires no backtracking. The design of symbol table organization and type definition table are also given. The storage allocation scheme for dynamic bags as well as other types of bag is discussed and a solution is proposed.

Naturally, developing an optimized and efficient compiler for a language takes a few years. What we are concerned with is a compiler for Bagit to show that the implementation is feasible. Since the implementation of the language is not the major theme of this research work, we keep the discussions brief.

1. A Compiler-Interpreter Organization

Figure 52 shows a compiler-interpreter organization considered for implementation of Bagit. There are five major components in this configuration: Lexical Analyzer (scanner), the compiler (consisting of Syntax Analyzer, Semantic Analyzer, and Code Generation), the Table Management (consisting of Symbol Table, Type Definition Table and its extensions, Constant Table, etc.), Error Detection and Recovery, and finally an Interpreter.
Figure 52. A compiler-interpreter organization

General Description

The objective is to implement a compiler-interpreter system for Bagit. The system should compile programs written in Bagit into intermediate code (of some abstract machine), checking for syntactic and semantic correctness. If no errors are found, it should then execute this code interpretively.

A) The Lexical Analyzer (Scanner)

The lexical analyzer (or scanner) recognizes all legal tokens and detects illegal ones. It uses blanks and special symbols as delimiters, and skips the comments
Each time it is called (by the syntax analyzer), it returns an internal code value (an integer), the token (a string) and the value of the token (a string). The value of the token is used for identifiers and constants, giving the name of the identifier or the value of the constant. For example:

<table>
<thead>
<tr>
<th>code</th>
<th>token</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>170</td>
<td>identifier</td>
<td>COUNT</td>
</tr>
<tr>
<td>1</td>
<td>all</td>
<td>-</td>
</tr>
<tr>
<td>172</td>
<td>string constant</td>
<td>'STRING'</td>
</tr>
<tr>
<td>13</td>
<td>begin</td>
<td>-</td>
</tr>
</tbody>
</table>

A complete list of the internal codes is given in Appendix E.

A lexical analyzer is developed for Bagit. The program listing is given in Appendix D.

B) The Compiler

The compiler portion of the system is split into three major parts, namely, the syntax analyzer, the semantic analyzer, and the code generation. It serves the following purposes:

1. Establishing whether or not the input is a syntactically correct program. If it encounters
a syntax error, it reports the error to the error handler, passing appropriate information.

A syntax analyzer has been developed for Bagit. The program listing and some sample runs are given in Appendix D.

2. Invoking appropriate semantic routines to perform semantic analysis. The semantic actions are inserted into the LL(1) grammar of the language (described in the next section).

3. Type checking via the semantic actions.

4. Building a compile-time symbol table, type definition table, and a run time table of constants.

5. Generating code for syntactically and semantically recognized constructs.

C) Table Management

The table management part includes all tables which hold relevant information about identifiers and constants encountered in the program. All data structures and associated operations for these tables are defined in this part of the compiler-interpreter.

D) Error Detection and Recovery

Errors are detected during lexical analysis, syntax analysis, semantic analysis, or at run time during
execution. The error handler reports the "cause" of the error rather than the "effects" of it. It repairs and continues the process.

E) The Interpreter

This part scans the code produced by the compiler and performs certain actions according to the instructions in the code, thus executing the program. It is called if and only if the compiler found no errors. The interpreter checks for run time errors, terminating execution if one occurs.

2. LL(1) Parsers: A Non-backtracking Top-Down Technique

A top-down parsing method called LL(1) parsing is used to implement the compiler for Bagit. LL(1) parsers are table-driven variants of recursive descent parsers and are suitable in conjunction with semantic actions, in that the semantic actions may be inserted anywhere in the productions. The LL-parser is non-backtracking and predictive which makes it more efficient than ordinary recursive descent parsers. An LL(1) parser is effectively based on an LL(1) grammar which permits deterministic left-to-right top-down recognition with a lookahead of 1 symbol.
LL(1) Grammars

Consider a production \(<A>::=w\) where \(<A>\) denotes a nonterminal symbol and \(w\) is a string of terminals and nonterminals. The production is called 'nullable' if and only if \(w\) is nullable, i.e., the null string may be generated from \(w\). The selection set of the production, denoted by \(\text{SELECT}(<A>::=w)\) is then defined as:

\[
\begin{align*}
\text{SELECT}(<A>::=w) &= \text{FIRST}(w) \quad \text{if } w \text{ is not nullable, and} \\
\text{SELECT}(<A>::=w) &= \text{FIRST}(w) \cup \text{FOLLOW}(<A>) \quad \text{if } w \text{ is nullable, where:}
\end{align*}
\]

\[
\begin{align*}
\text{FIRST}(w) &\text{ is the set of terminal symbols that occur at the beginning of the intermediate strings derived from } w \\
\text{and, } \text{FOLLOW}(<A>) &\text{ is the set of input symbols that can follow an instance of } <A> \text{ in an acceptable input sequence.}
\end{align*}
\]

We can now define an LL(1) grammar as follows: A context-free grammar is called an LL(1) grammar if and only if productions with the same left-hand-side have disjoint selection sets [27]. A procedure for finding selection sets for the productions of a given grammar is given in section 8.2 of [27].

**LL(1)-based Syntax Analyzers**

The steps to develop an LL(1) parser are as follows:

1. Transform the context-free grammar of the language
to an LL(1) grammar. Appendix B gives the context-free grammar of Bagit in BNF. The corresponding LL(1) grammar is given in Appendix C. In order to transform a context-free grammar of a language to an LL(1) grammar the following steps are usually taken [27]. For a detailed description see Appendix C of [27].

a) Process the repeating factors. For instance,

<id list> ::= <id> { , <id> }

may be replaced by:

<id list> ::= <id> <repeat id>
<repeat id> ::= null

 ::= , <id> <repeat id>

b) Left Factoring.

Suppose the grammar contains the two productions

<stmt> ::= if <B> then <stmt>

 ::= if <B> then <stmt> else <stmt>

The grammar cannot be LL(1) because both the sample productions have if in their selection sets. Left factoring replaces them with:

<stmt> ::= if <B> then <stmt> <something>
<something> ::= null

 ::= else <stmt>

c) Corner Substitution.

Consider the following grammar with starting
symbol \(<A>\):

\[
\begin{align*}
<A> & ::= a \\
& ::= <B> c \\
<B> & ::= a <A> \\
& ::= b <B>
\end{align*}
\]

The grammar is not LL(1) because the selection sets of the first two productions both contain \(a\). Substituting for \(<B>\) in the second production will result in:

\[
\begin{align*}
<A> & ::= a \\
& ::= a <A> c \\
& ::= b <B> c \\
<B> & ::= a <A> \\
& ::= b <B>
\end{align*}
\]

However, the new grammar is not LL(1) yet. But a left factoring on the first two productions makes it an LL(1) grammar.

d) Left Recursion.

As a simple example consider the following grammar where the first production is self left recursive:

\[
\begin{align*}
<S> & ::= <S> a \\
& ::= b
\end{align*}
\]

It is proved [27] that left recursive grammars cannot be LL(1). The given grammar effectively
generates the strings: b, ba, baa, baaa, etc.
This could be replaced by:

\[ <S> ::= b <list> \]
\[ <list> ::= \text{null} \]
\[ ::= a <list> \]

The general procedure for removing left recursive productions is found in [27]. It is also shown that steps a) through d) are adequate to rewrite a given non-LL(1) grammar so that it is LL(1).

2) Find the selection sets for all the productions.

3) When the grammar is transformed to an LL(1) grammar and all selection sets are found, parsing becomes very easy. The general procedure is as follows:
   a) Initially, the parser stack contains the start symbol S on its top.
   b) If the stack top contains a terminal symbol t, then the input token must be a "t", else error. If the two match, then advance the read head and pop the terminal symbol t from the stack.
   c) If the stack top contains a nonterminal symbol A, then examine the input token currently under the read head, and check the selection sets of productions with the left-hand-side A. If the selection set indicates production A::=w, then remove A from the stack and push the string w onto the stack; otherwise it is an error.
d) Stop the process when stack is empty. Accept the input program when the stack is empty and the last token is processed (i.e., the end-marker of the input is reached).

The advantages of an LL(1) parser are the simplicity of the method and the fact that the analyzer always makes the "right" decision and no backtracking is necessary. Moreover, the semantic actions can be inserted anywhere in the productions and when they reach the top of the stack, the proper routine is invoked.

3. Table Management

3.1 Type Definition Table

The type definition table contains the relevant information about the types defined in a program. It has several extensions which are given in the next section. The following shows the information kept in (a row) of the table for every type defined in the program.
Type Definition Table (TDT)

<table>
<thead>
<tr>
<th>level (of visibility)</th>
<th>first (attribute)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bag-label</td>
<td>rest (of the attributes)</td>
</tr>
<tr>
<td>type-name/attribute-name</td>
<td></td>
</tr>
<tr>
<td>definition of type</td>
<td></td>
</tr>
<tr>
<td>size (of a value or a descriptor)</td>
<td></td>
</tr>
<tr>
<td>B/U (Built-in/User-defined)</td>
<td></td>
</tr>
<tr>
<td>ptr to TDT (for further information)</td>
<td></td>
</tr>
<tr>
<td>ptr to TAT (Type Argument Table)</td>
<td></td>
</tr>
<tr>
<td>ptr to SD (Scalar Descriptor)</td>
<td></td>
</tr>
<tr>
<td>ptr to RE (Range Extension)</td>
<td></td>
</tr>
<tr>
<td>ptr to FD (Function Descriptor)</td>
<td></td>
</tr>
<tr>
<td>ptr to FAD (Formal/Actual Decomposition)</td>
<td></td>
</tr>
</tbody>
</table>

where:

1. 'level' indicates the level of visibility of an attribute of a bag type. If the level is assigned -1, the attribute is not visible and is therefore not allowed to be accessed. It also indicates the level of nesting when bags are nested inside one another.

2. The 'bag-label-first' and 'bag-label-rest' are used to handle labelled bags. They point to the first attribute of the label and the rest of the attributes of a label (if any) respectively.
3. The 'size' indicates the space needed for objects of that type. In case the size is not known at compile time (e.g. for dynamic bags), the size of a descriptor is stored in this field. The size of the descriptor is known at compile time.

4. The rest of the information is mostly pointers to extensions of TDT or is self explained.

A complete example of TDT and its extensions is given in a later section.

3.2 Extensions of Type Definition Table

The TDT contains the general information about types. However, when different types are defined in a program, some extra information is needed. In all of the following tables, NEXT indicates the next entry to the table.

**Range Extension (RE)**

This table is used to keep the lower and upper bounds of a subrange type, e.g. m .. n. An entry of RE is shown below.

<table>
<thead>
<tr>
<th>base value</th>
<th>limit value</th>
<th>NEXT</th>
</tr>
</thead>
</table>
Scalar Descriptor (SD)

This table is used to hold the information about a scalar type. The SD table has a descriptor for every such type and a Scalar Extension (SE) to hold the values of the type. This information is known at compile time.

Scalar Extension (SE)

Function Descriptor (FD)

The function descriptor and its extensions are used to keep the information about functions defined in a data or type abstraction. It is used in the decomposition process of an abstraction and when new types are defined later on (composition of a type). The following tables show the Function Descriptor (FD) and its extensions.
The Function Descriptor contains three pointers. The first points to the (formal) parameters of the function (if any). These parameters are kept in the Func-Parameter table. The second pointer indicates the type of the result which is either pointing to TDT when the type is known, or to TAT (Type Argument Table) when a type is passed as a parameter. If the function does not return any result to the calling routine, this pointer is null. The last pointer points to the object code table where the object code of the function is kept.

Formal/Actual Decomposition (FAD)

Any type or data abstraction will be decomposed into the following tables as shown:
The interface of an abstraction A is kept in the Decomposed
table. It contains three indices to where the information
is kept about filters, formal/actual parameters (if any),
and formal/actual type parameter (if any) of A. This
information is further decomposed into other tables, namely
Visibility table and Choices table. When a variable is
declared to be of type A, the actual information about the
new type is mapped to the decomposed (formal) information of A, and a new type will be maintained in TDT.

'Filters' is a table which holds the name of a filter, and a pointer to the Visible table where the visible objects of A are saved. If there are more filters associated with the abstraction A, the 'next filter' field points to the next filter.

An abstraction may have some variable parameters passed to it (via the interface). The 'Formal/Actual Parameter' table keeps these parameters. The 'value' field will be filled out at the time of instantiation of the variable of type A.

The Type Argument Table (TAT) stores the information about the choices of type that the abstraction expects. Different choices of type are kept in a table called Choices.

**Internal Representation of Bags**

Every user-defined type will eventually be decomposed into the built-in types of Baqit. The following are the internal names for the built-in types:

```
int    -- integer
real   -- real
bool   -- boolean
Fstr   -- Fixed length string
```
Vstr -- Variable length string,  
      size of descriptor is 2.

scal -- scalar

rang -- range

func -- function, size of descriptor=3

Besides, several kinds of bags are internally recognized  
which are defined below.

Cbag -- attributed bags

Lbag -- Labelled bags which are attributed  
      bags with some labels defined on  
      the bag. Therefore,  
      Lbag = Cbag + label.

Abag -- Dynamic bags, e.g., bag of T.

Obag -- Obags define a simple data  
      abstraction (ADT). Therefore,  
      Obag = Cbag + one filter  
      They are not parameterized.

Sbag -- Sbags represent superbags which  
      may have two or more filters and  
      may be parameterized. Hence,  
      Sbag = Cbag + filters + param

Dbag -- Derived bags (Dbags) are derived  
      from an Sbag. Every Dbag is  
      treated like an Obag.
Examples

This section is concluded with two examples. The first example shows the Type Definition Table (TDT) and its extensions for several types defined in a program. Some of the table headings were abbreviated to fit the table in a page. The second example shows a type abstraction, its decomposition, and its composition (use). In both examples, it is assumed that the built-in types are saved in a fixed part of TDT.

Example 1

type
  p is integer;
  s is string (20);
  t is string;
  u is ('mo','tu','wd','th','fr','sa','su');
  v is 'mo' .. 'fr';
  w is 1..25;
  x is bag of
      begin
            x1 is p;
            x2 is u;
      end;
  y is bag of x;
  z is bag (z1,z2) of
      begin
            z1 is s;
            z2 is bag (z22) of
               begin
                    z21 is integer;
                    z22 is w;
               end;
            z3 is x;
      end;
Example 2

Given the following generic type abstraction for a stack:

```plaintext
type stack is
dag [t is oneof (integer, real, P), limit is integer] of
begin
  myview is filtered stack of
  begin push, pip end;
```

### Table: TDT (Fixed Part of TDT for Built-in Types)

<table>
<thead>
<tr>
<th>Level</th>
<th>Label</th>
<th>Name</th>
<th>Def</th>
<th>Size BU</th>
<th>TDT</th>
<th>TAT</th>
<th>SD</th>
<th>RE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>p</td>
<td>int</td>
<td>1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>s</td>
<td>Fstr</td>
<td>20</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>t</td>
<td>Vstr</td>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>u</td>
<td>scal</td>
<td>3</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>v</td>
<td>scal</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>w</td>
<td>rang</td>
<td>2</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>x</td>
<td>Chag</td>
<td>4</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>x2</td>
<td>u</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>v</td>
<td>Abag</td>
<td>des</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>z</td>
<td>Lbag</td>
<td>27</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>z1</td>
<td>s</td>
<td>20</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>z2</td>
<td>Lbag</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>z21</td>
<td>int</td>
<td>1</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>z22</td>
<td>x</td>
<td>4</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>z3</td>
<td>x</td>
<td>4</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
yourview is filtered stack of
   begin push,pop,top end;
stk is bag of t;

-- push,pip,pop, and top as
-- defined in Figure 50
end;

var sl is stack[integer,25].myview;
s2 is stack[P,20].yourview;

then the TDT and its extensions would look as follows (some
of the table headings were abbreviated and not shown in
their entirety in order to fit the table on one page):

<table>
<thead>
<tr>
<th>level</th>
<th>name</th>
<th>def</th>
<th>size</th>
<th>B/U</th>
<th>TDT</th>
<th>TAT</th>
<th>FD</th>
<th>FAD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Stack</td>
<td>Sbag</td>
<td>des</td>
<td>U</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>stk</td>
<td>Abag</td>
<td>des</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>push</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>pop</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>top</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>pip</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>$0001</td>
<td>Dbag</td>
<td>des</td>
<td>U</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>stk</td>
<td>Abag</td>
<td>des</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>push</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>pop</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>top</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>pip</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>$0002</td>
<td>Dbag</td>
<td>des</td>
<td>U</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>stk</td>
<td>Abag</td>
<td>des</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>push</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>pop</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-1</td>
<td>top</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>pip</td>
<td>func</td>
<td>3</td>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In the second example the two Dbags are derived from the Sbag definition of stack and are internally named $0001$ and $0002$. The -1's in the Type Definition Table indicates that the corresponding attribute cannot be accessed.

3.3 Symbol Table Organization

Symbol table holds relevant information about identifiers encountered in the source text. These are organized in the following manner:

<table>
<thead>
<tr>
<th>Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>name</strong></td>
</tr>
<tr>
<td>--------------</td>
</tr>
<tr>
<td>table-top</td>
</tr>
<tr>
<td>LL</td>
</tr>
<tr>
<td>Scope Marker</td>
</tr>
</tbody>
</table>

where:
- 'level' is the lexical level which identifies which activation record the identifier belongs to,
- 'disp' is the displacement of the identifier in the activation record,
- 'others' is considered for future use.
- Scope Marker determines the scope of variables in Symbol Table when several blocks are nested
inside one another, and LL and table-top point to the next available cell in their respective tables.

As a result the pair \((\text{level}, \text{disp})\) corresponds to the address of an identifier. A binary search method to access the symbol table is recommended. This method is moderately easy to implement and provides fast access to identifiers \(O(\log n)\), where looking up the table is a major operation. Since Bagit performs strong type checking, effectively for any occurrence of an identifier in the text of an expression or function call, there should be a search in the symbol table (and consequently Type Definition Table). As a result, searching becomes a major operation for symbol table.

3.4 Constant Table

Constants encountered in the text of a program (e.g. in a \texttt{const} declaration) are saved in a constant table. It suffices to give the structure of the table followed by an example.
For instance, given the declarations:

```plaintext
const pi  is real 3.14;
max   is integer 25;
dash is string '-----';
sw    is boolean true;
n    is max;
```

The tables would look like:

```
const table

<table>
<thead>
<tr>
<th>const id</th>
<th>const type</th>
<th>const value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>real</td>
<td>3.14</td>
</tr>
<tr>
<td>max</td>
<td>int</td>
<td>25</td>
</tr>
<tr>
<td>dash</td>
<td>str</td>
<td>'-----'</td>
</tr>
<tr>
<td>sw</td>
<td>bool</td>
<td>true</td>
</tr>
<tr>
<td>n</td>
<td>int</td>
<td></td>
</tr>
</tbody>
</table>
```

integers

```
25
```

reals

```
3.14
```

strings

```
'-----'
```

boolean

```
true
```
4. Storage Allocation

4.1 Static Storage Allocation

All data types of Bagit except dynamic bags and variable length strings are allocated storage at compile time. We briefly discuss them in this section.

For built-in data types the storage allocation (in words) is as follows:

- integer: 1
- real: 1
- boolean: 1
- Fstring: length of string from TDT
- range: 2 (size of descriptor)
- scalar: 3 (size of descriptor)

A Cbag has a known size at compile time. As it is shown in TDT, the size of every Cbag at level i is the sum of the sizes of its attributes at level i-1. If one of the attributes is an Abag or Vstring, the size of the descriptor is considered which is known at compile time.

For Lbags the storage is allocated at compile time. They are Cbags to which some labels are associated (i.e. Lbag=Cbag + labels). The information about labels is kept in TDT and the storage is allocated for the Lbag as if it were a Cbag.

Sbags, Dbags, and Obags are also like Cbags with the
difference that some of the attributes are functions. In that case, the size of the descriptor of the function (i.e. 3) will be considered.

4.2 Dynamic Storage Allocation

A stack-heap based storage management is used for Bagit, which is shown below.

| Fixed | Stack ----> Free | <---- Heap |

where:

Fixed data structures are stored at Fixed part, the stack (of activation records) is used for temporary locations, parameter passing, recursion, Cbags, etc., and heap is a block of storage within which pieces are allocated and freed (e.g. for Abags) in some relatively unstructured manner.

The strategy is to put all objects whose sizes are known at compile time in the stack. For dynamic bags, there is a descriptor in the activation record, which points to the heap where actual objects are kept. Naturally a garbage collector will be necessary to provide space in the heap. An overflow condition is when the garbage collector rearranges the heap and yet the heap and the stack collapse.

Storage Allocation for Variable Length Strings (Vstr)

Variable length strings (Vstr) have a descriptor in
the activation record. The descriptor contains the current length of the string and the position of the first character of the string in the heap:

```
<table>
<thead>
<tr>
<th>descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
</tr>
<tr>
<td>current length</td>
</tr>
</tbody>
</table>
```

string storage (in the heap)

**Storage Allocation for Dynamic Bags**

Dynamic bags (Abags) have a dynamic nature. They can expand or shrink dynamically as objects are added to or removed from the bag at run-time.

The general approach for allocating space to such dynamic structures has been to allocate a descriptor (whose size is known at compile time) in the stack, and store the actual elements in a heap. For instance, in the case of adjustable arrays the descriptor (called dope vector) may contain the fixed information about the array such as number of dimensions, size of each element, lower bounds, upper bounds, and the origin where the first element will be stored in the heap.

However, dynamic bags have a different ordering defined on their objects which must be implemented efficiently. In order to do that, a descriptor is allocated in the stack which contains the following information:
The actual objects which are in the heap, are accessed via an indexed structure as follows:

![Diagram of indexed structure]

The INDEX is conceptually an ordered set of pointers which point to the heap where objects of the dynamic bag are stored. When a new ordering is to be defined on the objects of bag B, the ordering of the pointers of INDEX will be changed, but the actual objects in the heap remain in the same locations. As a result, the INDEX itself has a dynamic nature.

A variation of B+ tree called a B++ tree is used to implement the INDEX.

**B, B+, and B++ trees**

A B tree of order m is a tree which satisfies the following properties [19]:

- **First (object of the bag)**
- **Last (object of the bag)**
- **Current object (of the bag)**
- **Current size (of the bag)**
- **Size (of each object of the bag)**
(1) Every node has at most $m$ sons.
(2) The root has at least two sons.
(3) Every node, except for the root and the leaves, has at least $m/2$ sons.
(4) All leaves appear on the same level, and carry no information.
(5) A nonleaf node with $k$ sons contains $k-1$ keys.

B trees make it possible both to search and to update a large file with guaranteed efficiency, in the worst case, using comparatively simple algorithms [19].

A typical node of a B tree looks like the following:

```
[ p0, k1, p1, k2, p2, ..., p_{j-1}, k_j, p_j ]
```

where $k_1 < k_2 < \cdots < k_j$ are keys, and $p_0, p_1, \ldots, p_j$ are pointers. Operations find, insert, delete, etc. are explained by Knuth [19].

However, B trees have bad performance for sequential processing where a 'next' operation is needed. In a $B+$ tree, all keys reside in the leaves. The upper level nodes are organized as a B tree. The leaf nodes are linked together left-to-right, forming a linked list (called a sequence set). The sequence set provides the 'next' operation. The following shows the structure of a B+ tree:
For a dynamic bag, the 'pred' operation is also a major one. To add this capability to B+ trees, it is modified to provide the 'pred' operation by making the sequence set a doubly-linked structure. The modified structure is called a B++ tree and has the following structure:

The B++ trees are used in the implementation of dynamic bags as discussed before. The labels of the objects of the bag are used to traverse the index tree, and the sequence set contains pointers to the heap. Moreover for fast access to the elements, the descriptor points to the first, current, and last nodes of the sequence set so that if the structure of the bag is not changed, one can
directly access these elements without going through the index tree. Also note that, the first (last) object of the bag corresponds to the leftmost (rightmost) node of the index tree.
VI. Conclusion

In the past decade abstraction mechanisms of programming languages have been a major theme of research in computer science. Today, abstract data types are among the principle design issues of most recent programming languages.

However, abstraction has been treated in these languages as an isolated concept, i.e., just another linguistic construct provided in the language to define an abstraction.

Moreover, as we mentioned before, every data abstraction in these languages corresponds to a single abstract data type. There are circumstances where we want a single data abstraction to define more than one type [35]. Also, there are circumstances where we want a data abstraction to define a zero type [35].

Bagit provides an encapsulation mechanism (bag) and an information hiding mechanism (filter) which are fundamental design issues of the language.

Effectively, without these two mechanisms, no significant programs can be written in Bagit. That is, abstraction in the language is more than an isolated concept. It is applied in a uniform and consistent manner.

A data abstraction in Bagit is defined by a bag which
encapsulates the representation and valid operations on the data type, and a filter which defines an interface to the abstraction.

Solution to the Typing of Objects Problem

A type abstraction is a data abstraction which has two or more filters (interfaces), every filter defining a new type. As a result, unlike other programming languages, a type abstraction can define more than one type (without any additional constructs in the language).

A zero type is a type or data abstraction where all its attributes are of type function. It simply encapsulates a group of functions to be used as a private library of functions.

Solution to the VHLL Functional Abstraction

Functions in Bagit can return objects of a built-in type or type bag. Therefore, traditional functional abstraction is preserved in the language by hiding the implementation of the function. Moreover, since functions can return bags as their result, a function may represent several objects at one time. Finally, a zero type abstraction provides higher level functional abstraction by allowing an instance of the zero type to represent a group of functions.

Solution to the Loop Iterator Problem

Functions which return bag objects may be used in conjunction with loop constructs and bag expressions,
providing control abstraction. Such functions separate the process of selecting the next object from its use. In this case, they are user-defined generators for control abstraction. The two built-in generators (suchthat and range) also provide control abstraction by hiding unnecessary details of how an object is to be selected and used.

**Solution to the Associative Referencing Problem**

Associate referencing is considered one of the most important features of Very High Level Languages. In Bagit, a labelled bag is the supporting data structure for associative referencing. By hiding the irrelevant information of how an object is accessed, and only specifying what the object is, access abstraction is achieved.

Another kind of abstraction (which may be called information abstraction) in Bagit is to hide certain information of a bag via a filter. This way, several views of a bag may be defined. While, these information are hidden from one module, they may be visible to other modules.

Consequently, the uniformity of the language is achieved by consistent use of the bag constructor and its filters.

Since bags are generalized sets, and Bagit is a bag-oriented language, it fits in the class of set-oriented
languages which are considered to be Very High Level Languages [21].


APPENDICES
Appendix A. Collective Syntax Diagrams of Bagit

id:

unsigned integer:

unsigned number:

character string:

unsigned constant:

constant:

variable:
abstract type:

```plaintext
Mllag)  generic clause  of  begin
filters  field list  end
```

filters:

```plaintext
filter id  is  filtered  type id  of
begin  field id  end
```

ADT instance:

```plaintext
type id  actual clause
filter id
```

actual clause:

```plaintext
1

| type id

| variable id

| constant

| 1
```

filtered type:

```plaintext
filtered  variable id  label  of
field id
filtered bag
```
filtered bag:

block:

program:
Appendix B. BNF Syntax of Bagit

<program> ::= <id> is <block>.

<block> ::= <const dcl part> <type def part>
        <var dcl part> <stmt part>

<const dcl part> ::= null |
        const <const dcl> { ; <const dcl>};

<const dcl> ::= <id> is <const part>

<const part> ::= <type> <constant> |
        <constant id>

<constant id> ::= <id>

<id> ::= <letter> { <letter or digit or underscore> }

<letter or digit or underscore> ::= <letter> |
        <digit> |
        <underscore>

<constant> ::= <unsigned number> |
        <constant id> |
        <sign><unsigned number> |
        <character string> |
        <sign><constant id>

<unsigned number> ::= <unsigned integer> |
        <unsigned integer>.<unsigned integer>

<unsigned integer> ::= <digit> { <digit> }

<sign> ::= + | -

<character string> ::= ' { <character> } '

?type def part> ::= null |
        type <type def> { ; <type def> } ;

?type def> ::= <id> is <general type>

<general type> ::= <type> |
        <abstract type>

?type> ::= <simple type> |
        <bag type>

<simple type> ::= <scalar type> |
        < subrange type> |
        string ( <unsigned integer> )

?type id> ::= <id>

<scalar type> ::= ( <character string>
        { , <character string> } )
<subrange type> ::= <constant> .. <constant>

<bag type> ::= <simple bag type> | <structured bag type>

<simple bag type> ::= bag of <type>;

<structured bag type> ::= bag <interface> of
begin <field list> end

<interface> ::= null | <label>

<label> ::= ( <field id> { ,<field id> } )

<field id> ::= <id>

<field list> ::= <field def> { ; <field def> }

<field def> ::= <field id> [,<field id>] is <field type>

<field type> ::= <type> | <pseudo type>

<abstract type> ::= bag <generic clause> of
begin <filters> <field list> end

<generic clause> ::= null | [ <type argument part> <var argument part> ]

?type argument part> ::= null | <type argument def>
{ <type argument def> }

?type argument def> ::= <id> is <choice type>;

<choice type> ::= oneof ( <type id> { , <type id> } )

<var argument part> ::= <var argument def> { ; <var argument def> }

<var argument def> ::= <id> [,<id>] is <type id>

<pseudo type> ::= <function def> | <type function>

<function def> ::= function <parameter> of
<type of result>; <block>

<parameter> ::= null | <(id list)>}

?type of result> ::= null | <type>

?id list> ::= <id group> { ; <id group> }

<id group> ::= <id> [, <id>] is <type of group>
<type of group> ::= <type id> | <choice type>
<type function> ::= type ( <variable id> )
<variable id> ::= <id>

<var dcl part> ::= null | var <var dcl> { ; <var dcl> }
<var dcl> ::= <id> [, <id>] is <all types>
<all types> ::= <type> | <pseudo type> | <ADT instance> | <filtered type>
<ADT instance> ::= <type id> | <type id <actual clause> | <type id> . <filter id> | <type id> <actual clause> . <filter id>
<filter id> ::= <id>
<actual clause> ::= [ <actual type> <actual var> ] | null
<actual type> ::= null | <type id> [, <type id>]
<actual var> ::= null | <var or const> [, <var or const>]
<var or const> ::= <variable id> | <constant>
<filtered type> ::= filtered <variable id> <new label> of <body>
<new label> ::= null | <label>
<body> ::= <field id> | <filtered bag>
<filtered bag> ::= begin <filtered fields>
   { , <filtered fields> }
   end
<filtered fields> ::= <fields> | <fields> <type of fields>
<fields> ::= <field id> [, <field id> ]
<type of fields> ::= <type> | <filtered bag>
<stmt part> ::= begin <statement> { ; <statement> } end
<statement> ::= <simple stmt> | <structured stmt>
<structured stmt> ::= <in stmt> | <if stmt> | 
    <block stmt> | 
    <block id> : <block stmt>

<block id> ::= <id>

<simple stmt> ::= <assignment stmt> | <exit stmt> | 
    <function stmt> | <empty stmt>

<empty stmt> ::= null

<exit stmt> ::= exit <block id>

<assignment stmt> ::= <variable> := <expr> | 
    <function id><field select>:=<expr>

<function id> ::= <id>

<field select> ::= . <field id> { . <field id> } | null

<function stmt> ::= <simple function stmt> | 
    <function id>. <simple function stmt>

<simple function stmt> ::= <function id> | 
    <function id> ( <expr>[<expr>])

<block stmt> ::= <compound stmt> | <repetitive stmt>

<compound stmt> ::= begin <statement> {;<statement>} end

<in stmt> ::= in <structured bag id> {,<structured bag id>} do <statement>

<structured bag id> ::= <id>

<if stmt> ::= if <expr> then <statement> | 
    if <expr> then <statement> 
    {,<expr> then <statement>} 
    else <statement>

<repetitive stmt> ::= <until stmt> | <forall stmt>

<until stmt> ::= until <expr> do <statement>

<forall stmt> ::= forall <variable id> in <expr> do 
    <statement> 
    forall <variable id> in <expr> 
    suchthat <expr> do <statement>


<expr> ::= <simple expr> | <simple expr> <relational oprtr> <simple expr>
<relational oprtr> ::= <: | > | <= | >= | in
<simple expr> ::= <term> | <sign><term> | <simple expr><adding oprtr><term>
<adding oprtr> ::= + | - | or
<term> ::= <factor> | <term><mult oprtr><factor>
<mult oprtr> ::= * | / | div | mod | and
<factor> ::= <unsigned constant> | <variable> | (<expr>) | <bag factor> | not <factor> | <function designator>

<unsigned constant> ::= <constant id> | <unsigned number> | <character string>
<variable> ::= <variable id> | <field designator>
<field designator> ::= <bag variable> . <id or label>
<bag variable> ::= <variable>
{id or label} ::= <id> | <actual label>
<actual label> ::= ( <actual keys> {, <actual keys> } )
<actual keys> ::= ? | <character string> | <variable id> | <actual label>
<filters> ::= <filter definition>|;<filter definition>]; null
<filter definition> ::= <filter id> is filtered <type id> of <visible part>
<visible part> ::= <field id> | begin <fieldS> end
<bag factor> ::= [ <range expr> ] | <retrieve expr>
<range expr> ::= null | <expr> .. <expr>
<function designator> ::= <function id><func param> <func field>
\[<\text{func param}> ::= \text{null} \mid (<\text{expr}> \{,<\text{expr}>\})\]
\[<\text{func field}> ::= \text{null} \mid . <\text{field id}>\]
\[<\text{retrieve expr}> ::= \text{all} <\text{variable id}> \text{ in } <\text{expr}> \text{ suchthat } <\text{expr}>\]
Appendix C. The LL(1) grammar and Selection Sets

<program>::=<id> is <block>. {id}

<block>::=<const dcl part>
<type def part>
<var dcl part>
<stmt part>

<const dcl part>::= null
::=const <const dcl>
<repeat const dcl>

<repeat const dcl>::= null
::=<const dcl>
<repeat const dcl>

<const dcl>::=<id> is <const part>;
::=<id>
<repeat const dcl>

<const part>:::=integer <constant>
::=real <constant>
::=boolean <constant>
::=string <constant>
::=<id>

<rest string>::= null
::= (; , )

<constant>::=<unsigned number>
::=<id>
::=<character string> [' ]
::=<sign><num or id> [+ , -]

<num or id>::=<unsigned number>
::=<id>

<unsigned number>::=<unsigned integer>{digit}

<fraction>

<fraction>::=null
 ::= .<unsigned integer> [ . ]

<sign>::= + [+]
 ::= - [-]
<type def part>::=null [var,begin]
 ::=type<type def> {type}
 <repeat type def>
<repeat type def>::=null [var,begin]
 ::=<type def> [id]
 <repeat type def>
<type def>::=<id> is <general type>; [id]
<general type>::=<simple type> {id}
 ::=<bag type> [bag]
<bag type>::=bag <bag definition> [bag]
<bag definition>::=<label> of [{()}
 <structured>
 ::=<generic clause>of {} [of]
 <structured>
 ::=of <of tail> [of]
<of tail>::=<type> {(),digit, ,string, id,real,integer, boolean}
 ::=<structured> [begin]
<structured>::=begin <field list>end [begin]
<field list>::=<field def> [id]
 <repeat field def>
<repeat field def>::=null [end]
::=<field def> [id]
  <repeat field def>

<field def>::=<id><repeat id> is
  <field or filter>;

<field or filter>::=<field type>
  [(),digit,",string,
    id,bag,function,
    type,integer,real,
    boolean]

::=filtered <id> of
  <visible part>

<visible part>::=<id>
  ::=begin <fields> end

<type>::=<simple type>
  [(),digit,",string,
    id,integer,real,
    boolean]

::=bag <nonabst bag def>
  [bag]

<generic clause>::=[<generic arg>]
  [{}]

<nonabst bag def>::=of
  <simple or struc>
    ::=begin <fields> end

<simple or struc>::=<type>
  [(),digit,",string,
    id,bag,integer,
    real,boolean]

::=<structured> [begin]

<simple type>::=<scalar type>
  [{}]

  ::=<unsigned integer>
  ..<unsigned integer>

  ::=<character string>...
  <character string>

  ::=string<rest string> [string]

  ::=real [real]
::=integer {integer}
::=boolean {boolean}
::=<id><simple tail> {id}
<simple tail>::=null {; , )}
::=..<id> {..}
<scalar type>::=({character string} ()
    <rest of scalars>) {}
<rest of scalars>::=null {}
::=,<character string> [,}
    <rest of scalar>

<label>::=(<id><repeat id>) {}
<repeat id>::=null {},is,do,end,;}
    ::=,<id><repeat id> {
<field type>::=<type> {,digit,'string, id,bag,integer, real,boolean}
    ::=<pseudo type> {function,type}
<generic arg>::=<gen def> {id}
    <repeat gen def>
<repeat gen def>::=null {}
    ::=;<gen def> {;}
<gen def>::=<id><repeat id>is {id}
    <gen type>
<gen type>::=<choice type> {oneof}
    ::=<simple type> {}
<choice type>::=oneof(<id or type> {oneof}
    <repeat id or type>)
<id or type>::=<id>
    ::=integer
    ::=real
    ::=boolean
    ::=string <rest string>
<repeat id or type>::=null
    ::=<id or type>
    <repeat id or type>
<pseudo type>::=<function def>
    ::=<type function>
<type function>::=type ( <id> )
<function def>::=function <parameter> of<type result><block>

<parameter>::=null
    ::=(<id list>)
<type result>::=;
    ::=<type>;
{id list>::=<id group>
    <repeat id group>
<repeat id group>::=null
    ::=;<id group>
    <repeat id group>
{id group>::=<id><repeat id> is
    <type of group>
?type of group>::=<gen type>

\[
<\text{var dcl part}> ::= \text{null} \\
\quad ::= \text{var} <\text{var dcl}> \\
\quad \quad <\text{repeat var dcl}>
\]

\[
<\text{repeat var dcl}> ::= \text{null} \\
\quad ::= <\text{var dcl}> \\
\quad \quad <\text{repeat var dcl}>
\]

\[
<\text{var dcl}> ::= <\text{id}> <\text{repeat id}> \text{ is} \\
\quad <\text{all types}>;
\]

\[
<\text{all types}> ::= <\text{pseudo type}> \\
\quad ::= <\text{filtered type}> \\
\quad \quad ::= <\text{bag}<\text{nonabst bag def}> \\
\quad ::= <\text{scalar type}> \\
\quad '::= <\text{unsigned integer}>.. \\
\quad \quad <\text{unsigned integer}> \\
\quad ::= <\text{character string}>.. \\
\quad \quad <\text{character string}> \\
\quad ::= <\text{string}<\text{rest string}> \\
\quad ::= <\text{id}<\text{all tail}> \\
\quad ::= <\text{integer}> \\
\quad ::= <\text{real}> \\
\quad ::= <\text{boolean}>
\]

\[
<\text{all tail}> ::= \text{null} \\
\quad ::= ..<\text{id}> \\
\quad ::= .<\text{id}> \\
\quad ::= <\text{actual clause}> \\
\quad \quad <\text{filter id part}>
\]

\[
<\text{filter id part}> ::= \text{null} \\
\quad ::= .<\text{id}>
\]
<actual clause>::=[<actual type var>] {}

<actual type var>::=<idorconst> [id, digit, ',
<repeat idorconst>

<repeat idorconst>::=null []

 ::=, <idorconst> [,,
<repeat idorconst>

<idorconst>::=<id> [id]

 ::=<unsigned number> [digit]

 ::=<character string> ['

<filtered type>::=filtered<new label> {filtered}

 of <body>

<new label>::=null [of]

 ::=<label> []

<body>::=<id> [id]

 ::=<filtered bag> [begin]

<filtered bag>::=begin<filtered field> {begin}

 <repeat filtered end>

<repeat filtered>::=null [end]

 ::=<filtered field> [id]

<repeat filtered>

<filtered field>::=<fields> [id]

 <possible field type>

<fields>::=<id><repeat id> [id]

<possible field type>::=null []

 ::=is [is]

<possibilities>::=<type> [

, digit, ', string, id, bag, boolean,
integer, real]}

 ::=<filtered bag> [begin]
<stmt part>::=begin <statement> 
   <repeat statement> 
   end 
<repeat statement>::=null 
   ::=;<statement> 
   <repeat statement> 
<statement>::=<in stmt> 
   ::=<if stmt> 
   ::=<block stmt> 
   ::=<exit stmt> 
   ::=<id><stmt tail> 
   ::=null 
<stmt tail>::= : <block stmt> 
   ::=. <dot tail> 
   ::= <expr part> 
<dot tail>::=<actual label> 
   <rest of variable>::=<expr> 
   ::=<id><follow id> 
<follow id>::=. <follow dot> 
   ::=<expr part> 
<follow dot>::=<actual label> 
   <rest of variable>::=<expr> 
   ::=<id><rest> 
<rest>::= :=<expr> 
   ::=. <follow dot> 
<expr part>::= :=<expr> 
   ::=(<expr><repeat expr>) 
   ::=null
\[\text{<in stmt>::=in}<id><\text{repeat id}>do \text{<statement>}\]

\[\text{<if stmt>::=if}<expr>\text{then}<\text{statement}>\]
\[\text{<else clause>::=null} \]
\[\text{:=<repeat then> else}<\text{statement}>\]

\[\text{<repeat then>::=null} \]
\[\text{:=,<expr>then}<\text{statement} \]
\[\text{<repeat then>}\]

\[\text{<block stmt>::=<compound stmt> \}
\[\text{:=<repetitive stmt>}\]
\[\text{<compound stmt>::=begin <statement> \}
\[\text{<repeat statement> end} \]

\[\text{<repetitive stmt>::=<until stmt>}\]
\[\text{:=<forall stmt>} \]

\[\text{<until stmt>::=until <expr> do \}
\[\text{<statement>} \]

\[\text{<forall stmt>::=forall <id> in <expr> \}
\[\text{<execute> \}

\[\text{<execute>::=do}<\text{statement} \]
\[\text{:=suchthat}<expr>\text{do \}
\[\text{<statement>} \]

\[\text{<exit stmt>::=exit}<id> \]

\[\text{<repeat expr>::=null} \]
\[\text{:=,<expr><repeat expr>} \]

\[\text{<expr>::=<simple expr><rest of expr>} \]
\[+, -, \text{digit}, ', (, \text{not}, [\text{all}, \text{id}] \]
<rest of expr>::=null
   ::=<relational oprtr>
      <simple expr>
   <simple expr>::=<term><list expr>
      ::=<sign><term>
         <list expr>
   <list expr>::=null
      ::=<adding oprtr>
         <term><list expr>
   <relational oprtr>::= <
      ::= =
         ::= <
         ::= >
         ::= <=
         ::= >=
         ::= in
   <adding oprtr> ::= +
      ::= -
         ::= or
   <term>::=<factor><list term>
   <list term>::=null
      ::=<mult oprtr><factor>
         <list term>
<mult oprtr>::= *
    ::= / [*/]
    ::= div [div]
    ::= mod [mod]
    ::= and [and]

<factor>::=<unsigned number> [digit]
    ::=<character string> ['']
    ::=< ( <expr> ) > [{}]
    ::=not <factor> [not]
    ::=[ <range expr> ] [{}]
    ::=all <id> in <expr> suchthat <expr> [id]
    ::= <id><factor tail> [id]

<factor tail>::=null
    ::= .<following dot> [.]
    ::=(<expr><repeat expr>) [{},]

<tail>::=null
    ::= . <id> [.]

<following dot>::=<actual label> 
<rest of variable> {}
    ::=<id><following id> [id]

<following id>::=null
    ::=<*>(digit,/,div,mod,+, \,-,or,<>,=,<=,=,<=,=,<, >,in;,,,end,),do, then,else],...,
suchthat}
::= <id or label>
<rest of variable>

<range expr>::=null
::= <expr> .. <expr>

<rest of variable>::=, <id or label>
<rest of variable>
::= null

<id or label>::= <id>
::= <actual label>

<actual label>::= (<actual keys>
<repeat keys>)
<repeat keys>::= null
::=, <actual keys>
<repeat keys>
<actual keys>::= ?
::= <character string>
::= <id>
::= <actual label>
Appendix D. Program Listing: Lexical and Syntax Analyzers

program compilebag (source, tokens, grammar);

const
  idlength = 12;
  maxstk = 50;
  productlimit = 250;
  start = 449;
  rhs = 9;
  setsize = 27;

  {parser stack, max size}
  {max no. of productions}
  {starting symbol, <program>}
  {max size of right hand side}
  {max size of selection sets}

var
  line, token, value: string;
  special: string[27]; {special characters}
  delimiters: set of char; {see main program body}
  ch, firstchar: char;
  chs, chss: string[1];
  reswords: string[161]; {reserved words}
  endmarker, illegal: boolean;
  errors: boolean; {set if any syntax}
    {error detected}
  comment, necessary: boolean;
    {necessary is true when it is necessary}
    {to read a new character and prepare }
    {the next token on line.}
  code, i: integer;
  source, tokens, grammar: text;
  stack: array[1..maxstk] of integer; {parser stack}
  sp: 0..maxstk; {stack pointer}
  productions: array[1..productlimit, 1..rhs] of integer;
  selsets: array[1..productlimit, 1..setsize] of integer;
  maxprod: 1..productlimit;
  bottomarker: boolean; {stack bottom marker}
  found: boolean;
  p: integer;

{----------------------------------------}
{ }
{ SCANNER }
{ }
{----------------------------------------}

{----------------------------------------}
{ General Description: }
{ The scanner (or Lexical analyzer) scans the input }
{ program (written in Bagit) and encodes it into a }
{ form suitable for the compiler. It must recognize }
all legal tokens and detect illegal ones. Moreover, it should skip all comments (starting with --) and blank lines. In case of illegal tokens, the scanner reports the error and immediately looks for the next token so that the parser can always assume tokens are correct.

When it is called, the scanner returns three values:

- **code**: an integer number corresponding to the token. It will be used by the syntax analyzer.
- **value**: For identifiers, numeric constants, and string constants, giving the name of the identifier or the value of the constant.
- **token**: contains the token that has been recognized. It will contain the reserved keywords such as begin, end, ...

If the end of input program is encountered, the scanner returns the code for 'endmarker'.

The output of the lexical analyzer is a stream of tokens which are passed to the syntax analyzer. In this program, not all routines are commented. Only when the function of a module is not clear from its text, appropriate comments would help.

---

```
procedure scanner(var token,value:string; var code:integer);

procedure getnextchar;
begin read(source,ch); chs[1]:=ch;
    line:=concat(line,chs);
end;

procedure putlineout;
begin readln(source);
    writeln(tokens,line); line:='';
end;

procedure charconst;
begin  {check for string const}
    value:=concat(value,chs); ch:='';
    while not(eoln(source) or (ch='''')) do
        begin getnextchar;
            value:=concat(value,chs);
        end;
    if ch <> ''' then (no matching ' scanned)
```
begin
  writeln(tokens,'*** illegal string:',value);
  illegal:=true;  ch:=' ';
end else
begin  code:=172;  token:='string const';
if not eoln(source) then (* may be followed by another ' *)
begin  getnextchar;
if ch='''' then
begin
  delete(value,length(value),1);
  charconst; (recursive call)
end;
end;
end;
necessary:=(ch='''') and eoln(source) or (ch=' ');
end;

procedure itispechar;
begin (recognize a special char)
  value:=' ';
  (firstchar is the first char of token)
  if firstchar in [':'','..','>','<','-'] then
begin
  if eoln(source) then ch:=' ' else getnextchar;
  case firstchar of
    ':' : if ch='=' then token:=':=';
    '..' : if ch='.' then token:='..';
    '>=' : if ch='=' then token:='>=';
    '<=' : if ch='>' then token:='<=';
    '=-' : if ch='-' then
      begin  comment:=true;
        while not eoln(source) do
          getnextchar;
      end;
    end;
  end;
necessary:=ch in ['='','..','-','>','<',' '];
end else necessary:=true;
if firstchar='''' then charconst
else code:=pos(token,spechars)+200;
(200 is added to map the code to the appropriate range for special characters.)
end;

procedure maybenumber;
begin
  (see if all symbols are digits)
procedure maybeidentifier;
var id:string;
begin
{check if all symbols are valid}
i:=1; value:="";
while ( (i<length(token)) and 
(token[i] in ['0'..'9']) ) do i:=i+1;
if( i<length(token)) or 
not( token[length(token)] in ['0'..'9']) ) then 
begin illegal:=true;
   writeln(tokens,"*** illegal number:",
   token);
end else 
begin value:=token; code:=171;
   token:="numeric constant";
end;

{check for reserved words}
id:=copy(token,1,length(token));
{every reserved word id enclosed}
{in a # sign.}
i:=pos(concat('#',id,'#'),reswords);
if i<>0 then code:=i 
else begin
   value:=token;
   code:=170;
   token:="identifier";
end;

necessary:=(eoln(source) and 
   not (ch in delimiters) or (ch=' '));
end;

procedure identifytoken;
begin
illegal:=false;  firstchar:=token[1];
chss:=firstchar;
if firstchar in ['A'..'Z', 'a'..'z']
  then maybeidentifier
else if firstchar in ['0'..'9']
  then maybenumber
else if pos(chss,spechars)<>0
  then itispechar
else begin
  illegal:=true;
  necessary:=true;
  writelnitokens,
  'lc** illegal char in'
  ,token);
end;

procedure preparetoken;
{ this routine prepares for the next token }
{ It reads characters until a nonblank one }
{ is reached. Skips blank lines and if end }'
{ of line is true, prints the line out. }
begin
  ch:=' '; necessary:=false; comment:=false;
  if not eof(source) then begin
    while ((ch=' ') and not eoln(source)) do getnextchar;
    if eoln(source) and (ch=' ') then begin
      putlineout; preparetoken;
      end;
    end;
end;

procedure nextoken;
begin
  if necessary then preparetoken;
  if endmarker then begin
    token:='endmark';
    code:=173; value:='';
    putlineout;
  end else begin
    token:=chs;
    while not (eoln(source) or
      (ch in delimiters)) do begin
      getnextchar;
      if not (ch in delimiters)
        then token:=concat(token,chs);
    end;
identifytoken;
if illegal or comment then nextoken;
end;
end;

begin (SCANNER)
  nextoken;
end;

procedure checksyntax;
procedure readgrammar;
var i, j : 0..productlimit;
{read in the LL(1) grammar and}
{corresponding selection sets.}
begin
  j:=0;
  while not eof (grammar) do
    begin
      j:=j+1;  i:=0;
      repeat
        i:=i+1;
        read(grammar,productions[j,i]);
      until (productions[j,i] = -1);
      readln(grammar);  i:=0;
      begin

repeat  
i:=i+1;
readln(grammar,selsets[j,i]);
until  
(selsets[j,i] = -1);
end;
maxprod:=j;
end;

procedure advance;
begin
    scanner(token,value,code);
end;

procedure pop;
begin
    sp:=sp-1;  bottomarker:=( sp=0 );
end;

function tos : integer;
begin
    tos:=stack[sp];
end;

procedure push (p : integer);
var i,j : integer;
begin
    j:=1;
    {push pth production onto the stack}
    { -1 shows end of the right hand side}
    while (productions[p,j] <> -1) do j:=j+1;
    j:=j-1;
    if j <> -1 then
        for i:=j downto 2 do
            begin
                sp:=sp+1;
                stack[sp]:=productions[p,i];
            end;
end;

function inselset (p:integer) : boolean;
var i:integer;
    there:boolean;
    {see if the token is in one of the}
    {selection sets of pth production.}
    { -1 indicates end of sel. set. }
begin
    i:=1;  there:=false;
    while (selsets[p,i] <> -1) and not there do
        if code=selsets[p,i] then there:=true
            else i:=i+1;
    inselset:=there;
end;

function endmark : boolean;
begin
  endmark:=(code=173);  \{173 is end of tokens\}
end;

begin \{SYNTAX CHECKER\}:
  \{initialize the parse stack\}
  sp:=1; stack[sp]:=start;
  readgrammar;  advance;
  while not endmark do
  begin
    if tos<start then \{i.e. terminal\}
      if tos=code then \{i.e. input=tos=terminal\}
        begin pop;
            advance;
        end
      else begin
          writeln(tokens,tos,'expected, not ',
                  code);
          errors:=true;
          (* repair *)
        end
    else begin
      writeln(tokens,brks,'error');
      errors:=true;
      advance;
    end;
  end;
  if endmark and bottomarker and not errors
then writeln(tokens,'no errors detected');
else writeln(tokens,'errors detected');
end;

{===============================
} 
{ }
{ MAIN BODY }
{ }
{ }
{ OF }
{ }
{ COMPILER }
{ }
{===============================

begin 
{COMPILEBAG}

necessary:=true; 
line:=" "; 
endmark:=false;
chs:=" "; 
chss:=" "; 
errors:=0;
reset(source,'info:source.text');
rewrite(tokens,'info:tokens.text');
reset(grammar,'info:grammar.text');
delimeters:=":+,,-, *, /, (, ), [ , ] , { , } , :, '.', '>', '<', '>=', '<=', '==', '!=', '='; 
spechars:="+-*/(=)"; 
reswords:=concat('#alittand#bag#begin#boolean#constW, 
'div#do#else#end#exit#filterediV, 
'forall#function#if#in#is#integerW, 
'mod#not#of#oneof#or#real#stringiV, 
'suchthat#then#type#und#until#vartV); 
checksyntax; 
close(source); 
close(tokens,lock); 
close(grammar); 
end.


test1 is
   const pi is real 3.14;
   var r is real;
   p is real;
begin
   input (r);
   p:=2*pi*r;
   output('circle',p);
end.

no errors detected, program OK
test2 is
type
date is bag ofegin{align*}
\text{day} &\text{ is } 1..31; \\
\text{month} &\text{ is } 1..12; \\
\text{year} &\text{ is } \text{integer}; 
\end{align*}
end;
-- a labelled bag
year_info is bag (year) of
begin
\text{year} \text{ is } 50..82;
\text{status} \text{ is } \text{bag of } -- \text{nested bag}
\begin{align*}
\text{married} & \text{ is } \text{boolean}; \\
\text{children} & \text{ is } \text{integer}; 
\end{align*}
end;
\text{salary} \text{ is } \text{real};
\text{pos} \text{ is } \text{string};
end;
employees is bag (lastname, firstname, ssn) of
begin
\text{lastname}, \text{firstname} \text{ is } \text{string(20)};
\text{ssn} \text{ is } \text{integer};
\text{history} \text{ is } \text{bag of } \text{year_info};
\text{sex} \text{ is } \text{('male','female')};
\text{birth} \text{ is } \text{date};
end;
var
\text{employee} \text{ is } \text{bag of } \text{employees};
\text{persons} \text{ is } \text{employees};
begin
\text{input } (\text{employee});
\text{forall } \text{persons in } \text{employee do}
\begin{align*}
\text{persons.history.("82").salary} &\text{ := } 4500;
\end{align*}
end.

no errors detected, program OK
test3 is

    type complex is bag of
        begin
            re is integer;
            im is integer;
        end;

    ops is bag of
        begin
            add, mul is complex;
        end;

    var w is integer;
    z is ops;
    u, v is complex;

    -- functions can return bags, e.g.:
    complex_ops is
        function (x, y is complex) of ops;
        begin
            in complex_ops do
                begin
                    add.re := x.re + y.re;
                    add.im := x.im + y.im;
                    mul.re := x.re*y.re - x.im*y.im;
                    mul.im := x.re*y.im + x.im*y.re;
                end;
            end; -- complex_ops

    begin
        -- assume u and v are initialized variables
        w := complex_ops (u, v).add;
        -- OR --
        z := complex_ops (u, v);
    end.

    no errors detected, program OK
test5 is
-- define a generic type abstraction for stack
type stack is
  bag[T is oneof(integer, real, string),
     limit is integer] of
begin
  -- define two filters on the type
  myview is filtered stack of
    begin push,pip end;

  yourview is filtered stack of
    begin push,pop,top end;

  -- representation
  stk is bag of T;

  -- legal operations on stack
  push is function(newitem is T) of;
    begin
      if size(stk)>limit then overflow
        else stk:=stk+[newitem]; -- insert
    end;

  pop is function of;
    begin
      if empty(stk) then underflow
        else stk:=stk-last(stk); -- delete
    end;

  top is function of T;
    begin
      if empty(stk) then underflow
        else top:=last(stk);
    end;

  pip is function of T;
    begin
      if empty(stk) then underflow
      else begin
          pip:=last(stk); -- get top
          stk:=stk-last(stk); -- pop
        end;
    end; -- of stack

-- declare different stack objects
var int_stack is stack[integer,20].myview;
str_stack is stack[string,35].yourview;
begin
  -- assume x and y are of appropriate types
  int_stack.push(5);
  x := int_stack.pip;

  str_stack.push("first");
  y := str_stack.top;
  str_stack.pop;
end.

no errors detected, program OK
generic_swap is
    var u,v is integer;
    swap is
        function(x,y is oneof(integer,real,P)) of;
        var temp is type(x);
        begin
            temp:=x;
            x:=y;
            y:=temp;
        end;
    begin
        u:=5;
        v:=6;
        swap(u,v);
    end.

no errors detected, program OK
generic_swap is
  var u, v is integer;
  swap is
    function(x, y is oneof(integer, real)) of;
    var temp is type(x);
    begin
      temp := x;
      x := y;
      y := temp;
    end;
begin
  **** ERROR: = : bad input
  u := 5;  v := 6;
  swap(u, v);
end.

**** 1 ERRORS detected
Grades is

var id,s1,s2,s3 is integer;
grade is string(1);
ave is real;

begin
  -- recognizing until
  until false do
  begin
    input(id,s1,s2,s3);
    recognizing if-then and exit
    if id <= 0 then exit;
    -- nested ( )'s
    ave:=(((s1*0.25)/(2+(s2*0.25)/3+(s3*0.25)/4))+15)*2;
    recognizing if-then-then-else
    if ave >= 90 then grade:='A'
      ave >= 80 then grade:='B'
    ave >= 70 then grade:='C'
    ave >= 60 then grade:='D'
      else grade:='F'
    output(id,s1,s2,s3,ave,grade);
  end;
end;

no errors detected, program OK
## Appendix F. The Internal Codes

<table>
<thead>
<tr>
<th>Code</th>
<th>Token</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>all</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>and</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>bag</td>
<td>-</td>
</tr>
<tr>
<td>13</td>
<td>begin</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>boolean</td>
<td>-</td>
</tr>
<tr>
<td>27</td>
<td>const</td>
<td>-</td>
</tr>
<tr>
<td>33</td>
<td>div</td>
<td>-</td>
</tr>
<tr>
<td>37</td>
<td>do</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>else</td>
<td>-</td>
</tr>
<tr>
<td>45</td>
<td>end</td>
<td>-</td>
</tr>
<tr>
<td>49</td>
<td>exit</td>
<td>-</td>
</tr>
<tr>
<td>54</td>
<td>filtered</td>
<td>-</td>
</tr>
<tr>
<td>63</td>
<td>forall</td>
<td>-</td>
</tr>
<tr>
<td>70</td>
<td>function</td>
<td>-</td>
</tr>
<tr>
<td>79</td>
<td>if</td>
<td>-</td>
</tr>
<tr>
<td>82</td>
<td>in</td>
<td>-</td>
</tr>
<tr>
<td>85</td>
<td>is</td>
<td>-</td>
</tr>
<tr>
<td>88</td>
<td>integer</td>
<td>-</td>
</tr>
<tr>
<td>96</td>
<td>mod</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>not</td>
<td>-</td>
</tr>
<tr>
<td>104</td>
<td>of</td>
<td>-</td>
</tr>
<tr>
<td>107</td>
<td>oneof</td>
<td>-</td>
</tr>
<tr>
<td>113</td>
<td>or</td>
<td>-</td>
</tr>
<tr>
<td>116</td>
<td>real</td>
<td>-</td>
</tr>
<tr>
<td>121</td>
<td>string</td>
<td>-</td>
</tr>
<tr>
<td>Line</td>
<td>Token</td>
<td>Description</td>
</tr>
<tr>
<td>------</td>
<td>---------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>128</td>
<td><code>suchthat</code></td>
<td>-</td>
</tr>
<tr>
<td>137</td>
<td><code>then</code></td>
<td>-</td>
</tr>
<tr>
<td>141</td>
<td><code>type</code></td>
<td>-</td>
</tr>
<tr>
<td>146</td>
<td><code>und</code></td>
<td>-</td>
</tr>
<tr>
<td>150</td>
<td><code>until</code></td>
<td>-</td>
</tr>
<tr>
<td>156</td>
<td><code>var</code></td>
<td>-</td>
</tr>
<tr>
<td>170</td>
<td><code>identifier</code></td>
<td>name of id</td>
</tr>
<tr>
<td>171</td>
<td><code>numeric const</code></td>
<td>its value</td>
</tr>
<tr>
<td>172</td>
<td><code>string const</code></td>
<td>char. string</td>
</tr>
<tr>
<td>173</td>
<td><code>endmark</code></td>
<td>-</td>
</tr>
<tr>
<td>201</td>
<td><code>+</code></td>
<td>-</td>
</tr>
<tr>
<td>202</td>
<td><code>-</code></td>
<td>-</td>
</tr>
<tr>
<td>203</td>
<td><code>*</code></td>
<td>-</td>
</tr>
<tr>
<td>204</td>
<td><code>/</code></td>
<td>-</td>
</tr>
<tr>
<td>205</td>
<td><code>(</code></td>
<td>-</td>
</tr>
<tr>
<td>206</td>
<td><code>=</code></td>
<td>-</td>
</tr>
<tr>
<td>207</td>
<td><code>)</code></td>
<td>-</td>
</tr>
<tr>
<td>208</td>
<td><code>,</code></td>
<td>-</td>
</tr>
<tr>
<td>209</td>
<td><code>;</code></td>
<td>-</td>
</tr>
<tr>
<td>210</td>
<td><code>&lt;</code></td>
<td>-</td>
</tr>
<tr>
<td>211</td>
<td><code>.</code></td>
<td>-</td>
</tr>
<tr>
<td>212</td>
<td><code>&gt;</code></td>
<td>-</td>
</tr>
<tr>
<td>213</td>
<td><code>[</code></td>
<td>-</td>
</tr>
<tr>
<td>214</td>
<td><code>:</code></td>
<td>-</td>
</tr>
<tr>
<td>215</td>
<td><code>]</code></td>
<td>-</td>
</tr>
<tr>
<td>216</td>
<td><code>..</code></td>
<td>-</td>
</tr>
</tbody>
</table>
Appendix G. Language Reference Card

In a computer programming language, both data and operations on the data must be represented. The data in Bagit is described by definitions and declarations. The operations are described by operators and functions. Data values can be represented as constants or as values of variables. Variables must be associated with a type.

**CONSTANT**

A constant definition associates an identifier with a constant. The association does not change throughout the lexical scope of the definition.

**STANDARD TYPES**

A type definition associates an identifier with a type. The type may be one of the following standard built-in types or a structured type, bag.

a) `integer` and operations: `+`, `-`, `*`, `div`, `mod`

b) `real` and operations: `+`, `-`, `*`, `/`

c) `string` and operations: `concat`, `substr`, `length`, `index`, `delete`, `insert`

d) scalar type (which represents an ordered set of manifest string constants) and operations: `next`, `pred`

e) subrange type which indicates a restriction on an enumerable type, giving the smallest and largest values in a range allowed for values of the type.

f) `boolean` and operations and `,`, `or`, `not`

**STRUCTURED TYPE**

The structured type of Bagit is called a bag. An
attributed bag is a compound structure consisting of a finite number of components called attributes which may be of differing types. A particular attribute is accessed by the usual dot notation. A labelled bag is the supporting structure for associative referencing. It is an attributed bag where a combination of its attributes is chosen to label the bag. In a dynamic bag all objects must be of the same type. It may hold zero or more instances of a built-in or a user-defined type. There is a hidden ordering associated with any dynamic bag. The ordering is based on the labels of the objects in the bag if objects are labelled, or on a first-come, first-order basis.

The intrinsic functions first, last, next, pred, size, current, delete, empty, and EOB (End Of Bag) are associated with any dynamic bag. A particular object in a dynamic bag may be accessed using the intrinsic functions or an actual label (if objects are labelled). Each component of the actual label specifies the value expected to be found in the corresponding attribute. This value may be: don't care (?), literal ('v'), variable (v), or numeric (n). The aggregate operations union (+), intersection (*), difference (-), inclusion (in), equality (=), inequality (<>) , sub-bag (<), and super-bag (> ) are available on dynamic bags.

ABSTRACT DATA TYPES

Bag is also the encapsulation mechanism of Bagit which
together with an information hiding mechanism called a filter define a data or type abstraction. Bag encapsulates the representation of the type, the valid operations on the type, and a filter. The filter defines an interface to the abstraction by naming the visible operations. An abstract data type consists of a bag and a filter. A type abstraction is a data abstraction which includes more than one filter, each filter defining a new type.

**VARIABLES**

A variable declaration associates an identifier with a type. The type may be directly described in the variable declaration, or it may be a type identifier representing a previously defined type.

**FUNCTIONS**

Functions of Bagit are conceptually viewed as a pseudo-type. A function is defined in the variable declaration part of a block (program or function), or in a data/type abstraction. A function in Bagit is permitted to return a value of a standard type, or of type bag. Every function is invoked by a function statement.

**INPUT / OUTPUT**

Aside from the traditional input/output statements (input and output), Bagit provides a screen-oriented input/output facility for bags. A prompt defines a form on the screen to capture data or display it using the intrinsic functions "keyin" and "display" respectively.
Several prompts may be defined for a bag and kept in a separate bag which is constructed apart from the compiler.

**CONTROL STRUCTURES**

The assignment, exit, and function statements are simple statements in Bagit. Structured statements are made up of other statements and specify conditional, repetitive, or sequential execution of their component statements. The exit statement is used to terminate one or more nested repetitive statements. Repetitive execution of statements is specified by the forall-do and until-do statements. The forall-do clause provides for iteration over objects of a dynamic bag. The until-do statement provides loops controlled by a condition. The conditional execution of statements is specified by the if-then-then-else construct which is semantically equivalent to an if-then-elseif-else construct. Sequential execution of statements is specified by the compound statement which brackets the statements with begin-end.

**GENERATORS**

There are two built-in generators in Bagit. The suchthat generator may be used to qualify specific objects of a bag. The range generator (...) may be used to generate the values in the range of two given variables which are constants of an enumerable type or actual labels of objects in a bag. Moreover, all functions of Bagit which return a dynamic bag as their result are viewed as user defined
generators. These generators may be used in a bag expression or in conjunction with iterative statements.