205

FORM DEFINITION LANGUAGE FOR INTELLIGENT DATA OBJECTS

by

RICHARD P. SEWCZWICZ

B.S., Christian Brothers College, 1970

A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

Dr. Elizabeth A. Unger

ACKNOWLEDGMENTS

To Chuck D. Kissner, for identifying and supporting me as a candidate to the Summer-on-Campus Program.

To Robert G. Reeves, Jr., for his support during my participation in the Summer-on-Campus Program.

To my wife, Denise, and our two sons, Peter and Philip, for their support and understanding during my absence to partake in the program.

To Dr. Elizabeth A. Unger, for her advice in the preparation of this report and accessibility for consultation.

CONTENTS

LIST OF FIGURES

1. CHAPTER ONE: INTRODUCTION

1.1 WHY OFFICE INFORMATION SYSTEMS (OIS)

The goal of Office Information Systems (OIS) has been to accomplish the mission of the office efficiently by reducing costs, enhancing the quality of work, and extend the workers capabilities. In order to achieve this goal, companies in 1982 purchased approximately $6 billion of office automation equipment with projections of $27 billion to be purchased by the end of the decade [Wint85]. These purchases are typically for electronic equipment: word processing systems, electronic data processing (EDP) systems, communications equipment (telephone sets, private branch exchanges(PBX), modems, enhanced services, etc), and manufacturing control systems. For the most part, these systems are not integrated and thereby lead to potential problems and inefficiencies (e.g., manually transferring data from an EDP system to a word processing system). The challenge of OIS is the integration of the above components [Elli80, Tsic82] in order to reduce the complexity of the users interface to the system and control the flow of information to achieve increased efficiency in the office.

1.2 WHAT IS AN OFFICE

An office consists of people interacting in an environment to carry out the mission of a business by handling information dealing with the business [Baum80, Deog83, Elli80, Hamm79]. To achieve its mission in an orderly fashion, office procedures are defined. An office procedure is a structured framework by which the individual tasks and activities performed by office workers are organized [Hamm80]. The procedures insure that the office can operate in a predictable and repeatable fashion. This last point should not be taken to mean that all office procedures are fully specified and deterministic. Most office procedures are composed of both structured and repetitive tasks and by unstructured tasks which are characterized by their judgmental nature (e.g., negotiation, decision making, etc.).

Focusing in on an office, one would typically see a document moving from one activity to the next. The action taken with the document will be determined by the knowledge held by the individual who interprets the information presented by the document. Like office procedures, documents can range between highly structured (e.g., employment form, purchase order, etc) to unstructured (e.g., job resume, letter of reference, etc.). A document can be viewed as a filled form [Tsic82]. The traditional business paper form can be thought of as text which contains values in certain slots [Tsic82]. Not all documents can be viewed as forms. Forms are distinguished by the fact that many of the documents are similar, in that they have the same structure and same printed words, but differ from each other by the values placed in the slots.

Because of the structure of a business form, forms provide a structured approach to office messages [Tsic82]. Forms provide the necessary guide for a user in filling out a request for information, that is to say the user is not left free to his own designs. Forms, therefore, impose structure in communicating messages in the same way that formatted data imposes structure on knowledge [Tsic82]. The exactness of the message is benefical to the receiver of the message because it allows for a clear interpretation of the information being conveyed.

1.3 FORM-BASED SYSTEMS

Because business paper forms provide a structure, define the meaning of information, and provide direction to value insertion, they can be used as the basis for the design of an office automation system. Form-based office automation system have the following apparent advantages over non-form-based systems [Geha82]:

- Forms allow logically related data to be treated as an entity.
- Electronic forms retain many properties of paper forms (e.g., copied, edited, filed, etc.).
- Forms can be traced to determine their location in an office procedure.

- Access rights can be defined and automatically enforced for a form [Woo84].
- Forms can be designed to have routing information associated with the [Maze84, Shu82, McBr83, Zism77].
- Electronic forms also allow for an easier transition from the manual environment to the automated environment because of office worker's familiarity with paper forms.

With electronic forms, the potential exists to move the knowledge of what to do from the office worker to the electronic form. This passive object in the office, the form, can become active [Vitt81] and carry out its mission with minimal manual intervention.

Paper business form users are only capable of using a limited set of operations such as: read, enter values, send, receive, and locate. With the introduction of electronic forms, (from now form will be used to mean electronic form) the set of potential form operations increase. For instance, totals can now be automatically computed based on submitted values, field values can be inserted automatically, and entered data can be verified in real time. The fact that operations can be defined for specific form types gives forms the characteristics of an abstract data type [Tsic82, Geha82, Geha83]. An abstract data type defines a class of abstract objects (forms) and completely specifies the operations available for use with those objects [Lisk74].

1.4 OFFICE MODELING

Prior to the implementation of an OIS, careful analysis of the way an office conducts its business is necessary [Hamm79]. A methodology in Hammer [Hamm79] was proposed with one of the steps being the specification of the current office procedures. These specifications should be expressed in terms that are natural to the application in question. In [Hamm80], Hammer and Kunin describe several criteria that should be considered when designing a specification language. These attributes are summarized below:

- The language must be formal and well defined so that illegal specifications can be automatically detected.

- The language must be readable by office workers with little formal training in the language.
- The language must lend itself to writing descriptions in terms and structures appropriate to the office.
- The language should be high level and non-procedural.
- The descriptions specified by the language must be easy to modify.

In addition to the above five characteristics, an additional three features for a good modeling tool were provided in [Lebe82]. These additional characteristics are:

- Be able to represent the traditional hierarchical structure of an organization.
- The language should be able to express both asynchronous and concurrent events.
- The modeling tool should have a sound theoretical base, preferably if this base can be quantified mathematically.

With these criteria in mind, several specification languages and modeling tools for the office environment have been proposed in the literature. A selected few are described in the following sections.

1.4.1 PETRI NETS

A Petri Net can be viewed as a directed graph represented by two node types. A node drawn as a circle is called a place and a node drawn as a bar is a transition. The places in a Petri Net have the capability of holding tokens. For a given transition, those places that have edges directed into transitions are called input places and those places having edges directed out of this transition are called output places. If all the input places for a transition have a token, then the transition is said to be active and may fire. The firing removes the token from each input place and places a token into each output place [Zism77, McBr83]. Petri Nets represent a good modeling tool for office procedures because of their ability to show concurrency and decision points in a process. Concurrency is

demonstrated by being able to have multiple output places that can be active. This situation is very similar to any number of office procedures that use multi-part forms (e.g., customer copy, accounting copy and shipping copy) that are distributed to several organizations at once.

To enhance the Petri Net model, Zisman [Zism77] proposed the Augmented Petri Net (APN) formalism. The APN combines process and knowledge representations into one model. This merger of the two representations is necessitated by the fact that the Petri Net itself does not dictate when an enabled transition will fire, only when it is enabled to fire. With the addition of a production system which consists of a set of rules which specify a condition and an associated action, the APN will fire only when the production rule at the enabled transition is true and the tokens are present at the input places.

Another extension to the Petri Net for modeling an office environment is proposed by McBride and Unger [McBr83]. In their paper, they propose the use of a control Petri Net with an enhanced token. This token is viewed as an intelligent form. The intelligent form, or token, provides for a representation of information concerning the state of the procedure (e.g., where has it been, where is it going). The control net can then use the token to determine where the token is to be placed next by using the information supplied by the token.

1.4.2 INFORMATION CONTROL NETS (ICN)

ICN is a formalism that is intended to aid office managers and analysts to describe and evaluate procedures [Cook80]. The ICN model allows a user to graphically represent both the control structure and information structure being used for an office procedure. The ICN uses a directed graph augmented with various symbols to represent repositories of information needed to support an activity, decision points, and control and information flow with the use of directed arcs. ICN is supported with a software graphics package [Elli82] which allows the user to build or modify an ICN. In addition to the graphics package, a simulation system is available to evaluate the current procedures and the impact there might be if changes were made [Elli82].

1.4.3 OFFIS

In Konsynski, et. al. [Kons82] the authors propose the use of a specification language based on objects, attributes, and relations. An object is defined to be any entity in the office which cause activities to happen or receive the results from an activity. Attributes are defined to be properties that the objects of the system can possess. Relations are used to define the interconnection between objects and attributes. OFFIS supports a requirements specification language that is used to describe office activities. The system has an analyzer that maintains a database of requirements and insures the requirements specification is consistent and complete.

1.4.4 BUSINESS DEFINITION LANGUAGE (BDL)

BDL is a very-high-level programming language for the description of tasks to be accomplished by the clerical user [Hamm77]. BDL is a programming language; however, one of its intents is to be expressive enough that it can serve as a mode of communication between office workers [Elli80]. In BDL, documents that represent units of work serve as inputs and outputs that can be passed among boxes. Each box represents a processor that reads documents, preforms computations, and produces documents. Boxes are connected with directed arcs along which documents flow. A box can contain subboxes that are connected by arcs within the box. With this method of implementation, BDL lends itself to a top-down design approach.

Within BDL there are three major components known as Form Definition (FDC), Document Flow (DFC), and Documentation Transformation (DTC). FDC is used to describe the form templates necessary to produce a specific document. The DFC is used to define the basic structure of the application. The user describes the hierarchical structure of functions within the application and their dependence on and relationships among each other. DTC expresses what actual computations are to be performed on the document.

1.5 OFFICE INFORMATION SYSTEMS

This section will describe several systems that are being prototyped (or in use) to research the automation of office procedures. The systems that will be profiled will all have in common the fact that they are form-based and use forms as active messages or intelligent objects.

1.5.1 SYSTEM FOR BUSINESS AUTOMATION (SBA)

SBA is a system which allows the end user, as well as programmers, to directly describe their applications on the computer [DeJo80, Zloo77]. SBA uses the concept of two dimensional objects that a user is familiar with: tables, forms, and charts. SBA is built from abstract objects called BOXes. Each BOX is independent of every other and can process concurrently. The SBA BOX has many of the same characteristics as ACTORS [Byrd82, Hewi77]. These attributes are:

- Instances of BOX (ACTOR) types are the executing and communicating objects in the system. Their behavior is defined by a script and they maintain a memory between executions.
- BOXes (ACTORS) communicate with other BOXes (ACTORS) via messages.
- BOXes (ACTORS) execute asynchronously upon arrival of a message.
- BOXes (ACTORS) are modular with well defined interfaces.
- Every object is a BOX (ACTOR) in particular the messages are BOXes (ACTORS).

An outgrowth of the SBA project was Query-By-Example [Zloo80, Zloo81] which SBA uses for database management.

Each SBA BOX has four sections: identifier, input, output, and contents. Input defines from what other BOXes messages can be accepted as messages. Expressions can be associated with the input section. These expressions specify under what conditions the box can be triggered. The output section specifies what objects can be produced and where the outputs are to be sent. The contents section

describes what actions are to done to the input to produce the necessary output.

1.5.2 OFFICETALK AND OFFICETALK-D

OFFICETALK [Elli80] is a system that is based on data objects, displayed as single page forms and files of forms. Communications among work stations is supported by the sending and receiving of forms. An OFFICETALK implementation is accomplished by designing a specific set of blank forms for an activity and entering them into a database. OFFICETALK provides the user with an electronic desktop which simulates a typical desktop in the the sense that it has in and out baskets, one form can partially cover another, and inventories a set of blank forms. When a user wants to create a document, they will select a form from the inventory of blanks which is then fully displayed on the screen. The individual enters data by pointing at the appropriate field and then uses the keyboard to enter the value. Once the document is fully prepared, the user then selects one of several form operations that is available (e.g., send, file, copy, etc.).

OFFICETALK-D (the D means distributed) presents a similar type interface to the user as OFFICETALK [Elli82]. Office workers can provide a script to the system that describes how an activity is to be completed by a single user. A limitation of OFFICETALK, specifically no support for traditional database operations and no distributed capabilities, was overcome with OFFICETALK-D. Within the database, relationships are established between actors and roles. In OFFICETALK-D an actor is the names of people or devices within an office that are responsible for completing an activity. A role is the name of all positions within an office that people could fill (e.g., accountant, supervisor, engineer, etc.). A relation is established between actors and roles, also a relation is made between roles and activities thereby controlling who has authorization to perform which activities. Since activities are defined by scripts OFFICETALK-D has the capability to suspend the normal sequence of activities for an instance of a form and allows the user to directly route the form. This capability is provided to only authorized users. OFFICETALK-D has as its underlying concept Information Control Nets.

1.5.3 OFFICE FORM SYSTEM (OFS)/TRONOTO LATEST ACRONYM (TLA)

OFS and TLA [Tsic82] provide extensive form processing capabilites in order to describe well defined office activities. The design basis for OFS/TLA is that by using forms they can specify activity which is well defined and worth automating.

In OFS a form type represents a data type which is defined for a form [Tsic82]. A form instance is an occurrence of the type which may include added information beyond the form attribute values (e.g., time stamps, history of changes, copy numbers, etc.). In OFS the concept of a form was extended to include different communication medium (e.g., display, print, voice, text, and data). A form template provides the mapping to translate a form instance to the particular communication medium specified. OFS form attributes can only be singled valued, that is, there can be no repeating groups. This is a recognized limitation and was done this way to avoid implementing complicated displays. Another limitation is that OFS only allows single page forms to be designed. It is also recognized that a capabilitiy is needed for multi-paged forms. When a form instance is created, a unique form number is generated to guarantee unique identification. This function is accomplished by a control node issuing groups of unique identifiers to the work stations. This is analogous to distributing uniquely identified blank paper forms in a manual environment. Also, when copies of a form instance are made, copy numbers are generated so as to keep track of the number of copies made, maintain unique identification, and to be able to distinguish the original form from a copy.

OFS is basically a passive system; nothing is done until the user initiates an action. TLA, on the other hand, is active by allowing the user to specify conditions and actions to be taken on what is referred to as sketches. Sketches are form-like objects that represent the form that the procedure will eventually manipulate [Hogg81]. TLA, like OFS, does not assume any knowledge of the system state other than what is in its own form file or mail tray. Automatic procedures are defined by a set of sketches which will describe what needs to be done. Some of the sketch types available to the user are: pre-condition, action, and destination. The user interface to TLA is very similar to that of SBA

or OBE. If a user wishes to specify something beyond the capabilities of sketches, then specific programs must be provided for the application.

1.6 INTELLIGENT DATA OBJECTS

Based on the current research being done in the application area of OIS we are proposing the implementation of Intelligent Data Objects (IDO). The IDO will need to have the following capabilities to be of benefit in an office environment.

- Define and provide for a set of form operations.
- Perform field operations within a form automatically.
- Generate message to other nodes to collect data items from data bases not located at a users work station.
- Determine the forms next location based on the current state of the IDO.

In order to support the implementation of the IDO a Form Definition Language will be described in the next set of chapters. The Form Definition Language is targeted at a technical individual, not at the clerical user level. The language will be relatively non-procedural, by that is meant, the user will specify what is to be done and not how it is done.

The following chapters will describe the design and implementation of certain portions of the language. A brief description of each chapter follows:

- Chapter 2 will describe the overall requirements of a Form Definition Language in order to be useful.
- Chapter 3 will present the syntax diagrams that define the Form Definition Language.
- Chapter 4 discusses the tools available to aid in the construction of the FDL compiler.

- Chapter 5 will provides a summary of FDL and further areas of development that can be undertaken.

2. CHAPTER TWO: FDL - REQUIREMENTS

2.1 FORM DEFINITION LANGUAGE (FDL)-OVERVIEW

In support of the Intelligent Data Object (IDO), a Form Definition Language (FDL) is proposed. The purpose of the language is to allow a user to specify what data fields are to be on a form and what actions are to be taken with them. The intent of the FDL is to allow an applications programmer, knowledgeable in office procedures, to specify, in a non-procedural way, a form to be used as an IDO. Before specifying the requirements of the FDL a discussion of what is meant by a non-procedural language is needed. Non-procedural means that the forms designer will provide the specification of what needs to be done and not how it is done. Zisman in [Zism77] indicates that the "simplicity of the statement is somewhat misleading." Zisman [Zism77] states that non-procedurality is a relative term and that as knowledge is increased in the application area then, that knowledge will be reflected in the language. The proposed language specification in the next sections and chapters is only a starting point from which to evolve to a good non-procedural FDL.

2.2 FORM DEFINITION LANGUAGE - REQUIREMENTS

In [Geha83] several requirements for a form definition language are specified. The proposed Form Definition Language requirements described in the following sections follow many of those same requirements.

2.2.1 DATA FIELD DEFINITION

Data field definition refers to the naming of fields that will define a form type and the variable information contained within the form type. When providing a name for the field the form specifier also needs the capability to associate the following items with the field name:

- data type (integer, character, etc.)
- data length
- user prompts (help messages)

- textual information (typically preprinted formation found on a paper form)
- data state (required or optional)
- assign a default value
- designation of input validation checks

2.2.2 ACCESS RIGHTS

Access rights refers to the identification of who is authorized to do what concerning a form. Access rights can be associated with three different levels, namely;

- form type level
- form instance level
- form field level

Certain individuals may not be allowed to access a particular form type, this type of restriction does not allow access to the next two levels. A particular person may have access to a form type but only a limited set of form operations is available to them. An individual's access to data fields may also need to be restricted. Since the list of individuals can be quite extensive several papers [Geha83, Geha82, Woo84, Elli82] have proposed the use of roles to identify access rights. A role is defined as a name given to a function performed in an office environment. Thereby, with the association of roles to form operations and data fields, access to the form and its contents can be made secure.

2.2.3 ELECTRONIC SIGNATURES AND LOCKING

The FDL must provide access to an abstract data type for allowing a user to specify where an electronic signature needs to be placed [Geha82, Geha83, Lum82]. Signing an electronic form must be possible if electronic mail systems are to replace the existing paper mail system for business transactions. This is necessary so that the receiver of a message has proof that it originated from the identified sender. In addition, an electronic signature must be message dependent, as well as signer dependent [Rive83]. This is needed because the recipient of a message may either modify the

contents or attach the electronic signature to another message (cut and paste). Therefore, after an electronic signature is applied, fields need to be identified as unalterable in order to prevent tampering.

2.2.4 ACTION SCRIPTS

Action scripts describe what data items are to be provided by the user, what fields are to be provided automatically by the system, identify what fields are to be computed and how they are to be computed. In addition to data entry features, the form designer must be allowed to specify a control structure using the data values on the form to guide the user in the next steps of completing the form. These actions need to be associated with each visit a form makes to a role. The form specifier should also be able to incorporate rountines [Tsic80] written to support tasks too complicated for the FDL to implement.

2.2.5 ROUTING SCRIPTS

Routing scripts will characterize the flow of forms throughout the office until their logical conclusion. The routing scripts will specify the destination, under what conditions a form will be forwarded (conditional and unconditional), and identify when a form is to wait for other forms of the same type before proceeding [Zloo77, Maze84]. The destination should be expressed by the use of logical naming conventions [Lum82, Shu82, Maze84] or specific names [Shu82]. The routing script should be able to specify concurrent distribution of the same form type.

2.2.6 DATABASE ACCESS

The form designer will need to be able to specify an interface to a database with a minimal amount of description. The user will provide this description without specifying the physical location of the database needed. The specification database access is required so as to support automatic data insertion, translate logical names to physical locations for routing, and the translation of role names to qualified names.

2.2.7 DISPLAY

The FDL will not provide for allowing a user to specify the physical appearance (i.e. positioning of text material, font size of a character, highlighting, etc.). In Gehani [Geha83] a DISPLAY section is specified is his proposed FDL but indicates that "heuristics" will have to be used to insure properly positioned information. For this project it is proposed that a Form Design subsystem be specified to support this display function; this approach is consistent with several published articles [Lum82, Shu82, Yao84, Zloo80, Bern82].

3. CHAPTER THREE: FDL - SYNTAX

3.1 FORM DEFINITION LANGUAGE (FDL)-SPECIFICATION

In the following sections, specifics of the FDL will be provided. The details of the language will be presented with both the use of syntax diagrams and written description.

3.2 FORM TYPE SPECIFICATION

Each form must have a form identifier (i.e., name), a block to define it and the word END. Figure 1 describes this general structure. Each new form type must have a unique name. The name will be checked against a list of existing form names to determine if the name has been used with a previously defined form type. This check will be accomplished by accessing a relation that contains all previously defined forms (each tuple is a form definition). The block consists of several mandatory and optional items, including data declaration and actions to be taken with the data declared. Every form must contain data declarations. If a form is defined with no other specifications, the eventual output of the FDL compiler will be a new database relation assuming that the relational database model is supporting the Intelligent Data Object (the term relation will be used to refer to the Intelligent Data Object (IDO) database model). The keyword END will signal the end of a form definition.

3.2.1 DATA DECLARATION

Data declaration is accomplished by defining each data item once in a data dictionary and with subsequent reference to these definitions. This approach is used in the implementation of the Office Data Input system (ODIN) [DiPi83, Ferr82] and FORMANGER [Yao84]. In McLeod's paper [McLe76], it suggests the implementation of a high level language to specify the domain of an object, the ordering of domains with regard to other values in the same domain, and an action clause if the domain specification is violated. Because one data item has the potential of representing the same type of object on various forms, the concept of specifying the domain of a data item and placing it in a data dictionary for use by a variety of forms is very beneficial.

In order to specify a data field for a form, it first must be specified in the data dictionary via a data domain specification similar to that described by McLeod [McLe76] and Ferrans [Ferr82]. It is not the intent of this report to specify the domain language, but to specify what FDL will expect to receive from the data dictionary.

Adding a data item into the data dictionary will require the user to specify for each item its name, type, and length as a minimum. In addition to these items, the user should be allowed to specify valid number ranges, denote a set of acceptable values, define error messages when a violation of domain is detected, and provide appropriate help messages. Also, the user should have the ability to specify data items that are constructed from other data items. A data item of this configuration is called a structure. Typically a structure will represent a collection of like objects (e.g., date is year, month, day or address is number,street name, city, state, zip code). The user will then reference the structure name and receive all associated elements as opposed to referencing each item separately.

Once the data dictionary contains the data items needed for a form the form designer can then declare them. Figure 2 displays the syntax diagram for providing data declaration. Data declaration is identified by the keyword DATA. With this declaration each data field and those fields that can be used to uniquely identify an instance of a form can be specified. Two other declarations can be made, role identifications and specifying names for each part of a multi-part form.

3.2.1.1 DATA FIELD DECLARATION

Data fields identify the objects that are to be collected on a form. The data field name must reference a data item in the data dictionary. If the designer specifies a data field name that is more descriptive for his application then it must be followed by the keyword DD (See Figure 3). The dd identifier following DD will reference the data dictionary. If the user specifies a data field name that is defined in the data dictionary as a structure then all data items within the structure are members of the form. This is the equivalent of declaring each data item separately in the data declaration. If neither the data field identifier or the dd identifier is present in the data dictionary an error condition exists and

the user is so notified.

3.2.1.2 REPEATING GROUPS

To handle a larger population of forms, the concept of repeating groups must be allowed to be specified. Typically, the design of forms have been hierarchical in structure, only in the simpler forms does the data structure tend to be flat [Lum82]. One of the limitations of OFS [Tsic82] was its inability to support repeating groups. A number of systems, as seen in the literature [Shu82, Bern82, Ferr82, Yao84], have allowed the user to specify repeating groups. In FDL the analyst will specify the maximum length (See Figure 4) a repeating group can be. This approach models closely the design of paper forms and should simplify the implementation of a display function. A repeating group identifier will not reference a data item name in the data dictionary. A repeating group identifier is local only to the form. To reference an element within a repeating group, the syntax is "repeating group identifier.data field name" (See Figure 4).

3.2.1.3 SPECIAL DATA TYPES

Special data types identify certain attributes a data field may have besides length and type. Listed is a set of special data types and their associated syntax diagram is illustrated in Figure 5.

- REQ (required) specifies the data field must contain a valid entry before a form instance can be considered complete.
- OPT (optional) specifies that the data field does not need to contain an entry before a form instance is considered complete.
- SIGN (signature) specifies that an electronic signature needs to be provided. Before a signature is applied a check must be made to see if that individual is designated to have the role in question. This assumes that there exists a relation that associates names of individuals and their assigned roles.

3.2.1.4 KEY FIELD DECLARATION

The key field declaration identifies each data field that is required to uniquely identify a form instance. Key fields once entered cannot be altered and will be available to all declared roles. The key field data will be entered by the user or provided by the system. All key identifiers must be previously declared. If the system is to provide the key value, then the keyword FUNCTION must be provided with the appropriate process name. The function specified by the designer must be available to the compiler or else an error condition exists. See Figure 6 for the syntax diagrams specifying the key. Data fields declared within a repeating group cannot be used as a key.

3.2.1.5 ROLE DECLARATION

Role declaration refers to the identification of functions that are being performed in the office that will affect the form. The identification of roles implies identification of the sites that a form instance can be routed to. Figure 7 diagrams the syntax for role declaration. A role may be defined as a data field only if the data field has a special data type of SIGN. A system maintained relation will exist that defines all the roles within the system. If a declared role identifier is not present in the relation, an error is present. The keyword ALL implies that all roles have the potential of receiving this form. The keyword EXCEPT excludes those roles identified from receiving the form being specified.

3.2.1.6 MULTI-PART DECLARATION

Multi-part declaration refers to the naming of various copies a form may have [Shu82]. This is analogous to a multi-part paper form with each part uniquely named (e.g., customer copy, shipping copy, accounting copy, etc.). Figure 8 is the syntax diagram for multi-part declaration. The multi-part identifier must be unique and not previously declared.

3.2.2 ACTION LIST

The action list contains those statements that specify what is to be done with the fields previously declared. Any identifier detected in this specification not previously declared will be identified as an error. In the next sections, access rights, data entry rules, and user help will be described.

3.2.2.1 ACCESS RIGHT SPECIFICATION

Access rights refers to the giving of permission to a user to perform certain functions associated with a form. The access rights can be assigned at three levels; form type, form instance, form fields. Access rights for form type were assigned at the time of role declaration. Form instance and field access rights will be assigned (or denied) with FIELDS-RIGHTS and FORM-OPS. TEMPLATE controls what can be displayed on a particular multi-part name. Figure 9 specifies the syntax for access rights.

3.2.2.1.1 TEMPLATE

The purpose of TEMPLATE is to define what field names can be included on one part of a multi-part form. If no TEMPLATE (See Figure 9) statement is provided, then it is assumed that all parts of form can contain the data fields previously declared. If a multi-part identifier and data field identifier is used that were not previously declared, then an error has occurred. ALL and EXCEPT have been previously defined in section 2.1.5 and have the same meaning.

3.2.2.1.2 FIELD-RIGHTS

FIELD-RIGHTS (See Figure 9) specify which roles are allowed to update which data fields. All identifiers (role, data field) must be previously declared. ALL and EXCEPT have been previously defined.

3.2.2.1.3 FORM-OPS

The purpose of the FORM-OPS statement (See Figure 9) is to define which form operations can be performed by which declared roles. At this time system provided form operations have not been defined. If no FORM-OPS statement is submitted then the system will assume all roles declared have access to all form operations being supported by the system. An assumption is being made that a relation exists that identifies all allowable form operations. An error exists if a role identifier is used that was not previously declared.

3.2.3 DATA ENTRY RULES

The syntax rules developed to describe how data is to be entered is described in the next sections. Also, while defining the data entry rules, the user can specify the types of validation checks that will be required to guarantee correctness of the form. Also, recall that contained within the data dictionary a set of validation rules may have been specified for a data field. When a forms designer references a data item in the data dictionary that has validation checks the form, being specified, will automatically inherit those checks. Because of the extensive nature of error checking that can take place, it is not the intent of this report to provide syntax diagrams for error checking, but to discuss the topic in general.

In Ferrans [Ferr82] a language identified as SEDL was discussed that implemented a wide range of integrity checks. In this paper four classes of error checks could be defined for a form. These classes of error checks are:

- Domain Checks - error checks made on data fields at data entry time.
- Structure Checks - checks made on the overall structure after all data elements of a structure have been entered.
- List Elements (repeating groups) Checks - checks made on list elements after all elements have been entered.
- Form Checks - checks made on the entire form to provide overall form integrity.

In Ferrans [Ferr82] a significant amount of detail is given with regard to the syntax used to provide for the above classes of checks. His paper [Ferr82] is possibly a starting point for further development of error checking constructs. In McLeod [McLe76], BNF specifications are provided for the implementation of domain checks. In Gehani [Geha83], he describes the use of PRE and POST to specify a condition that must be met either before and/or after data entry to determine whether a value is accepted. In general business applications using paper forms, the use of PRE and POST may be sufficient to do validation checking.

The syntax diagram for data entry is provided in Figure 11. If the data field identifies a repeating group then all elements must be referenced using the form "data identifier.element" or "data identifier[n].element name". The former references all elements of that name within a repeating group; whereas, the latter representation references a specific row within the repeating group. Each statement is further described in the following sections.

3.2.3.1 ENTER

The ENTER statement is used to specify that a data field is to be provided by a user at data entry. The data field identifier must be previously declared and the role that is providing data entry must have been given field access rights to the data field. If the data field identifies a repeating group, all elements must be referenced using the form "data identifier.element name".

3.2.3.1.1 SEQUENCE IDENTIFIER

The sequence identifier, Figure 12, specifies which field must be entered first at data entry. If no sequence identifier is provided, then the ordering of input is not significant for that data field. Two keywords are available for this function, BEFORE and AFTER.

3.2.3.1.2 VALIDATION RULES

Validation rules define the acceptable data values that can be placed in a data field. The validation rules insure that the data integrity of the form is maintained. Validation rules can be as simple as defining an acceptable range of values to as complicated a defining inter-data field checks. Because of the extensive domain of validation checking, validation rules will not be specified for this report. Validation rules will not be specified for this report.

3.2.3.2 INSERT

The function of the INSERT statement is to specify which data fields will be entered via an access to a database. At the time the form is being compiled a check will be made with the database manager supporting the IDO to verify that the relation name exists, that the specified relation keys are sufficient to uniquely access a tuple, and that the data field in question can be acquired in the

relation. An underlying assumption to this access method is that the data dictionary that specifies the domain of a data field is integrated with the database management system. Prior to verification with the data dictionary, the data field name will be translated to the domain name used in the data dictionary name.

3.2.3.3 CALC

The purpose of CALC is to allow the user to specify mathematical expressions to derive a data field from other data fields. The data field identifier must be previously declared and be either an integer or real data type, if these conditions are not met then an error condition exists. The data fields must be of the same data type or else on error condition exists. The syntax for expressions are described in Section 2.9.

3.2.3.4 LOCK

The function of LOCK is to specify a set of fields which cannot be altered when the condition specified is met. It is expected that this statement will be used after a signature is applied. In order to unlock the fields an authorized individual will need to perform the necessary form operation. The LOCK facility provides an additional level of security to prevent tampering.

3.2.4 USER HELP SPECIFICATION

The user help specification statements will allow a user to specify default values and user help messages. In Figure 13 the statements for the user help specification are documented.

3.2.4.1 HELP

The HELP statement allows a user to specify a message to assist the user at data entry. If there is data dictionary help message for a data item, the form HELP statement will have precedence and its message will be displayed.

3.2.4.2 ASSIGN

The purpose of ASSIGN is to allow the designer to specify a default value for a data field. During the compile process, all declared data fields will be set to blanks or zero, depending on the type of the

data field. The ASSIGN statement will override this process for the data field specified. The compiler will perform type checking and, if there is a mismatch, an error condition exists.

3.2.5 ROUTING SPECIFICATION

Routing specifications will allow a forms analyst to specify under what conditions a form will be routed to a specific location. The conditions can be unconditional or based on data values contained in the data fields of the form. The syntax for specifying routing is shown in Figure 14. The form designer will specify each site a form is expected to visit (a site is identified by a role name), the destination and conditions required to deliver it, and the actions required to be performed for each visit to a site. When the forms designer is specifying the route of a form, either an ICN or APN should be prepared in advance to insure all conditions and paths are properly specified.

In Mazer [Maze84], two types of routing exist for normal cases were described, type routing and instance routing. Type routing is specified at form definition time and pertains to all instances of the form. Instance routing is provided by the user when an instance of a form is created. In addition to these two routing types, override routing is discussed. Override routing is used to suspend the normal routing specification to handle exceptional situations. The specification of routing in FDL will pertain to type routing. The override and instance routing are viewed to be outside the scope of FDL. An assumption that is being made about routing is associated with each site (which is identified as a role) is that the equivalent of an in-basket and an out-basket. Forms directed to a site will be deposited in the appropriate role in-basket. After the site has completed its processing the form will be placed in the out-basket were it is be evaluated and forwarded. If a form that is delivered to a site and future actions are dependent on the arrival of another form, the system should hold these until all forms with the same key value arrive. Once this has occurred, the forms can then be deposited into the in-basket.

3.2.6 ROUTING SCRIPT

The routing script is identified by the keyword ROUTE (See Figure 14). Imbedded within the routing script are routing rules for each site an IDO is to visit. This specification is preceded by the keyword SITE. For each occurrence an IDO visits a site the actions to be taken must be specified, this is accomplished with the VISIT statement.

3.2.6.1 VISIT

The VISIT statement is used to specify what actions are to taken at a particular visit occurrence. Visit occurrences are identified with an unsigned integer. A visit identified as zero has special meaning. VISIT [0] states that this is where the form instance can originate. The compiler will evaluate each visit occurrence in ascending order. If a sequence of visit occurrences are missing (e.g., 0-1,5-8), the compiler will warn the user that a visit occurrence is missing. Associated with each visit is a set of actions to be taken. The visit rules are defined in the following sections. If no rules are specified for a visit then routing is terminated.

3.2.6.1.1 OUT-BASKET ACTIONS

To forward a form to the next site the SEND statement (See Figure 15) is specified. When not followed by the keyword WHEN, SEND will unconditionally forward a form to a site. If the boolean expression is evaluated to be true, the form will be routed to the specified role name. If the boolean expression is evaluated as false, the send operation will not be executed and the next out-basket statement will be evaluated. If the keyword OTHERWISE follows the boolean expression, then those out-basket statements following the keyword OTHERWISE will be evaluated. The function ERROR is used to alert that an abnormal condition exists in the form and that automatic routing for the form instance is halted. If this condition occurs, manual intervention will be required. Below is an example of SEND with both unconditional and conditional routing specifications.

UNCONDITIONAL:

SEND FORM-1 TO ACCOUNTING

CONDITIONAL:

SEND TO ACCOUNTING WHEN TOTAL >= 10000

or

SEND TO ACCOUNTING WHEN TOTAL >= 10000
OTHERWISE {SEND TO PURCHASING}

If no form name is provided, it is assumed to be that of the form identifier that followed the keyword FORM. All data elements associated with the form will be forwarded. If a multi-part identifier is provided, then only those data fields associated with the multi-part identifier are forwarded to the next site. The symbol "&" indicates an "and" condition. This specifies that each multi-part identified is to be forwarded to the site specified. The COPY attribute signifies that a copy of a form or multi-part is to be forwarded to a site. When a copy is made, it has no official status within the system. That is, it cannot be used to modify the contents of a data base. The purpose of COPY is to convey information about some set of events that has been accomplished (e.g., I have signed your request, here is your copy). In order for a copy to be sent to a site, the site must be declared to have permission to receive the form or the multi-part.

The role name selection identifies the destination of a form. The destination can be specified a number of ways. These methods are described below:

- role name identifier - if the role name is separated by the symbol "&" this means that the form will be routed concurrently to the site identified.

- PREVIOUS - indicates that the site that had it last is now the destination.
- ORIGIN - indicates that the site that created the form instance is the destination.
- USER data field identifier - indicates that the next site is identified by the contents of the specified data field. At run-time the routing system must verify the validity of information so as to convert it to an address. Also a run-time check is required to verify that the role associated with the contents is valid for the form.
- APPROVAL - is a special routing process for a form. An assumption is being made that a relation exists that specifies under what conditions a form will be routed for approval. APPROVAL implies sequential routing. The relation requires the following attributes: condition, data field name, and role name. The domains of each are: condition-boolean operators, data field name-data value, and role-identifier. For instance, purchases typically require a number of signatures based on the amount being purchased, therefore a relation could be described as:

CONDITION	TOTAL	ROLE
>	0	supervisior
=>	10000	manager
=>	100000	director
etc.	etc.	etc.

Therefore, if the form purchase order had a total of 10,500 the purchase order instance would be routed automatically to the first two roles for approval.

Each SEND statement will be executed in a sequential manner (as encountered). Checks will be required to verify that all identifiers have been previously declared.

3.2.6.1.2 IN-BASKET ACTIONS

In-basket actions are shown in Figure 16. The RECEIVE statement is used to specify that before a

form is to be deposited in the in-basket, it must wait until the arrival of the other forms specified before it is released to the in-basket. All forms or multi-part names require a match on key fields and values The use of the symbol "&" indicates an "and" condition. The RECEIVE statement is optional. If the receive statement is the only statement in the visit specification, then this terminates routing of the form.

3.2.7 CONDITION SPECIFIER

The purpose of the condition specifier is to allow the user to specify the execution of certain statements based on the values in the specified data field(s). If the condition is true, then the first set of statements are executed. If the condition is false and OTHERWISE is not specified, the statement directly following the WHERE statement will be executed. If OTHERWISE is specified, then those statements will be executed. The syntax diagram for the condition specifier is described in Figure 17. The syntax diagram for the boolean expression is illustrated in Figure 18. In Figure 18 a special function called ERROR is specified. The purpose of ERROR is to signal the user that an abnormal condition exists for the form instance being operated on. If ERROR is invoked processing of the form instance should be halted with an appropriate error message. In Figure 18 a function named SIGNED is described. The purpose of this function is to verify if an electronic signature has been applied to the data field being specified. The SIGNED keyword can only be used with data fields that have been designated as a special data type of SIGN. The only logical operator that can be associated with SIGNED is equal (=).

3.2.8 EXPRESSIONS

Expressions (See Figure 19) are used to define arithmetic operations to be in the CALC statement and within a boolean expression. No specialized functions, such as SUM, AVG, MIN, etc, but are recognized as being needed, are not specified for this report.

3.3 USING FDL

Before specifiying a form using FDL the form designer should use one of the formal modeling tools for describing office procedures. Some of these tools that are available are summarized in section 4.0. This step is required to insure that all data items, identification of who uses the form and what actions are taken with, and the repositories of information needed to support the procedure are fully specified. At the completion of this step the form designer consults with the existing data dictionary or if one does not exist, begin planning its implementation. The analyst must identify all data items required to support the form are present in the data dictionary. Missing items must be added using the data dictionary specification language (this language was not described in this report). Associated with each data item in the data dictionary is the name, length, type, domain range, etc. Once these steps are completed the form designer can begin to define the form using FDL.

In figure 20 is a sample form used at Kansas State University. The purpose of this form is for a graduate student to document their program of study. Some assumptions taken with this example include the existence of a data base that contains student records and a translation of logical names to physical addresses.

Once this form has been completed, the student then proceeds to collect the necessary signatures from the advisory committee. The advisory committee is made up of three professors. One of these professors is identified as the major professor and the other two professors are identified as committee members. Once the advisory committee accepts the program of study, acceptance is signified by a signature, it is then forwarded to the department head for signature. When the department head signs the program of study it is submitted to the dean of the graduate school for approval. After all the necessary signatures are collected the program of study is placed into the student's permanent file.

Using Figure 20 as a model the form designer, using FDL, identifies the form name, data fields, key fields, and roles that will make up this form. The FDL code below defines these areas:

```
FORM ms-prog-of-study:
        DATA: {         curriculum: {REQ},
                student-name-last DD(name): {REQ},
                student-name-first DD(name): {REQ},
                student-name-middle DD(name): {REQ},
                plan-type: {REQ},
                ssn: {REQ},
                courses [20] {    deptname: {REQ},
                                  course-number: {REQ},
                                  course-name: {REQ},
                                  course-credits: {REQ},
                                  school-name},
                major-prof: {REQ},
                commem-one: {REQ},
                commem-two: {REQ},
                dept-head-name: {REQ},
                major-prof-sig: {REQ,SIGN},
                commem-one-sig: {REQ,SIGN},
                commem-two-sig: {REQ,SIGN},
                dept-head-sig: {REQ,SIGN},
                date-dept-head-sig,
                dean-of-graduate-school-sig: {REQ,SIGN},
                date-dean-sig,
                total-credit-hrs}
        KEY:  {ssn}
        ROLES: {grad-student,
                major-professor,
                committee-mem,
                department-head,
                dean-of-graduate-school,
                graduate-file};
```

Next to be defined are the data fields that will be accessible to each of the roles. In this example there is no need for TEMPLATE since there is no need to restrict viewing of the data fields. Access rights are defined below:

```
FIELD-RIGHTS:       grad-student      EXCEPT{     courses,
                                      major-prof-sig,
                                      commem-one-sig,
                                      commem-two-sig,
                                      dept-head-sig,
                                      date-dept-head-sig,
                                      dean-of-graduate-school-sig,
                                      date-dean-sig},
                major-professor{ major-prof-sig},
                committee-mem{        commem-one-sig,
```

```
                                commmem-two-sig},
department-head{                department-head-sig,
                                date-dept-head-sig},
dean-of-graduate-school{dean-of-graduate-school-sig,
                                date-dean-sig};
```

After the field rights have been defined the data entry rules can be specified.

```
ENTER:      ssn,
        student-name-last,
        student-name-first,
        student-name-mid;
INSERT:     courses.deptname,
        courses.course-number,
        courses.course-name,
        courses.course-credits,
        courses.school-name FROM student-record (ssn);
ENTER:      major-prof,
        commem-one,
        commem-two;
CALC: total-credit-hrs FROM courses[1].course-credits +
                            courses[2].course-credits + ..... +
                            courses[20].course-credits;
ENTER:      commem-one-sig,
        commem-two-sig,
        department-head-sig,
        date-dept-head-sig,
        dean-of-graduate-school,
        date-dean-sig;
LOCK: EXCEPT{       commem-one-sig,
            commem-two-sig,
            department-head-sig,
            date-dept-head-sig,
            date-dean-sig,
            dean-of-graduate-school-sig}
            AFTER (major-prof-sig = SIGNED);
```

The following segment of code describes the routing for this form type. The routing for this form is quite simple as signatures are collected the form is forwarded to the next site. If for some reason an individual elects not to sign the form it is then routed back to the student to reconcile the problem.

```
ROUTE:
```

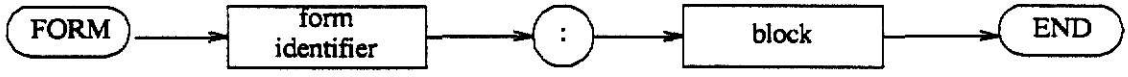
```
SITE: grad-student{
        VISIT[0]
                {SEND TO {major-professor} USER{major-prof}}
        VISIT[1]
                { }
    }
SITE: major-professor{
        VISIT[1]
                {SEND TO{committee-mem USER commem-one}
                 WHEN (major-prof-sig == SIGNED)
                 OTHERWISE {SEND TO{grad-student USER ssn}}}
        }
SITE: committee-mem{
        VISIT[1]
                {SEND TO{committee-mem USER commem-two}
                 WHEN (commem-one-sig == SIGNED)
                 OTHERWISE {SEND TO{grad-student USER ssn}}}
        }

SITE: committee-mem{
        VISIT[1]
                {SEND TO{department-head USER dept-head-name}
                 WHEN (commem-two-sig == SIGNED)
                 OTHERWISE {SEND TO{grad-student USER ssn}}}
        }
SITE: department-head{
        VISIT[1]
                {SEND TO{dean-of-graduate-school}
                 WHEN (dept-head-sig == SIGNED)
                 OTHERWISE {SEND TO{grad-student USER ssn}}}
        }
SITE: dean-of-graduate-school{
        VISIT[1]
                {SEND COPY TO{grad-student USER ssn}
                 WHEN (dean-of-graduate-school == SIGNED);
                 SEND TO{graduate-file USER ssn}
                 WHEN (dean-of-graduate-school == SIGNED)
                 OTHERWISE {SEND TO{grad-student USER ssn}}
        }
    ;
END
```

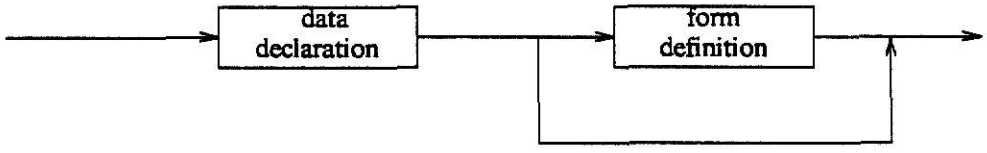
In the process of describing this form with FDL it was noted that without the existence of a function like SUM or further definition of referencing repeating groups calculating total credits required substantial input specification. To perform the addition each row in the repeating group would have to be each time, in this case 20 times. Obviously this method is unacceptable and needs further

definition.

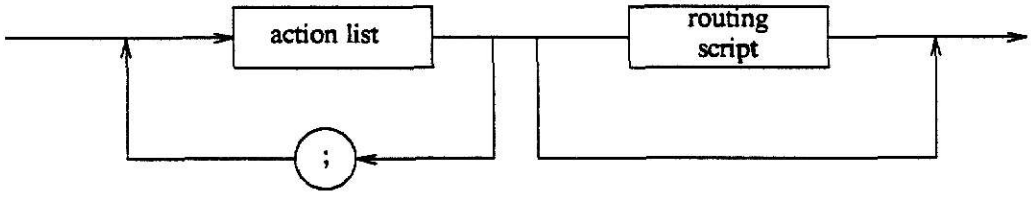
form type



block



form definition



action list

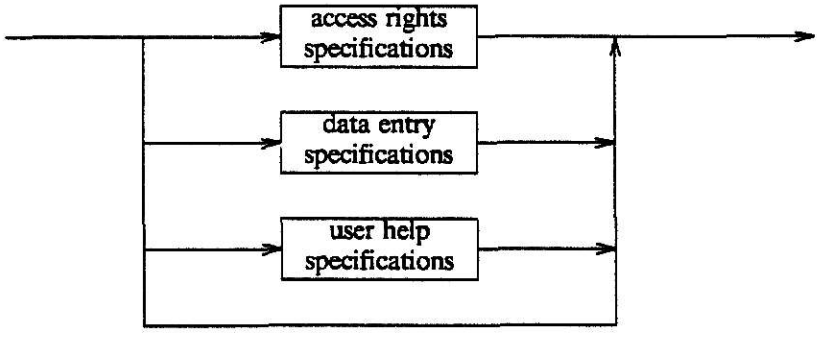
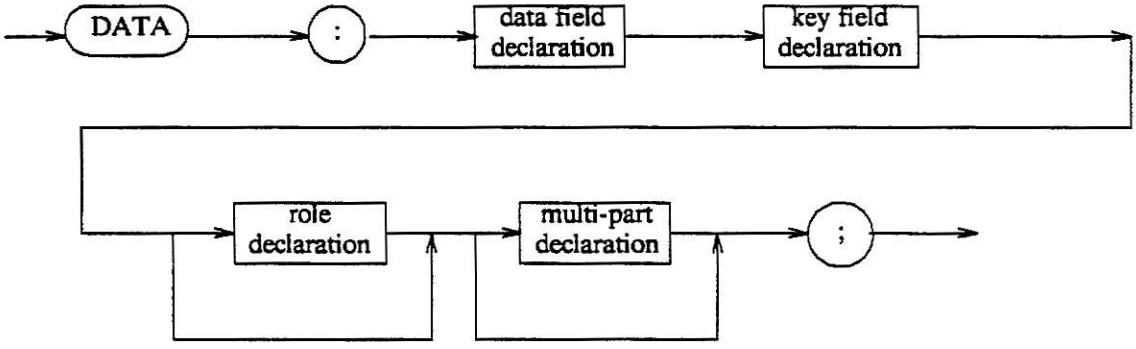


Figure 1. FORM TYPE SPECIFICATION

data declaration



data field declaration

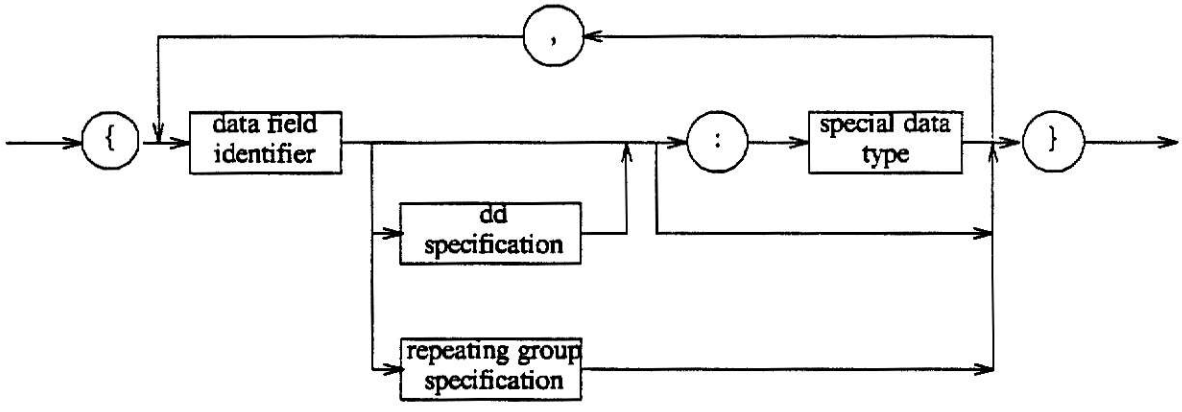


Figure 2. DATA DECLARATION

dd specification

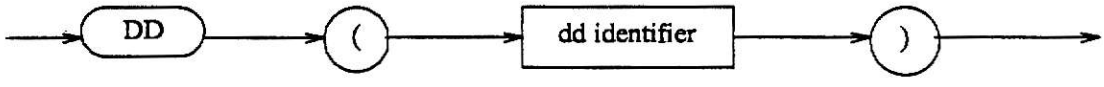
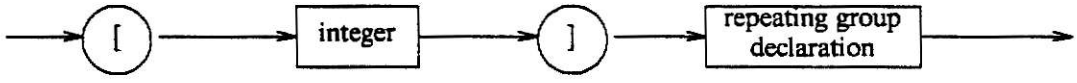


Figure 3. DD SPECIFICATION

repeating group specification



repeating group declaration

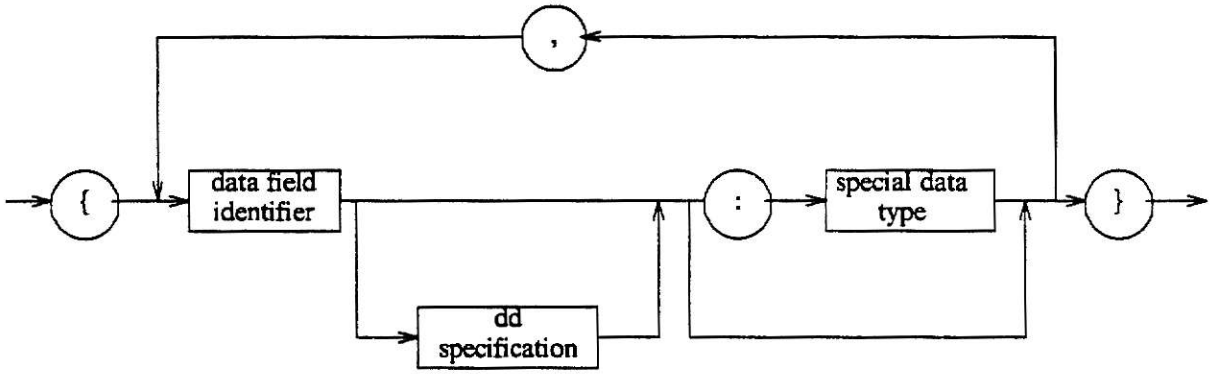


Figure 4. REPEATING GROUP SPECIFICATION

special data type

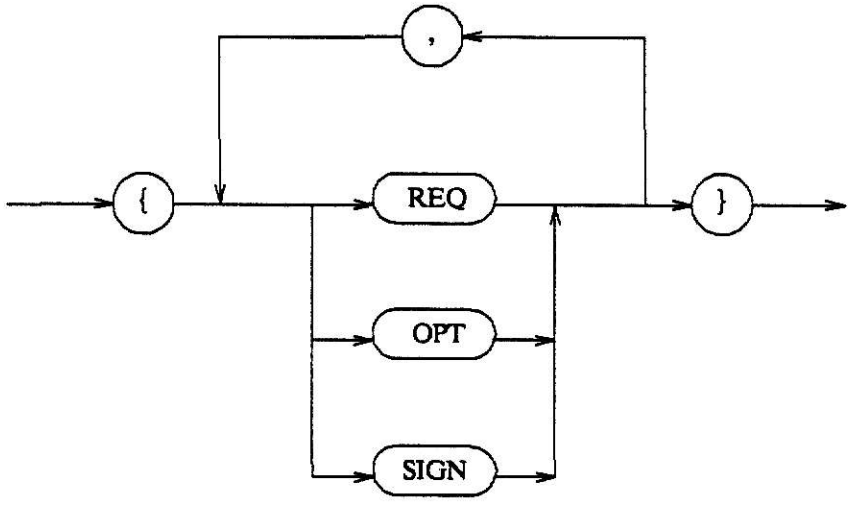
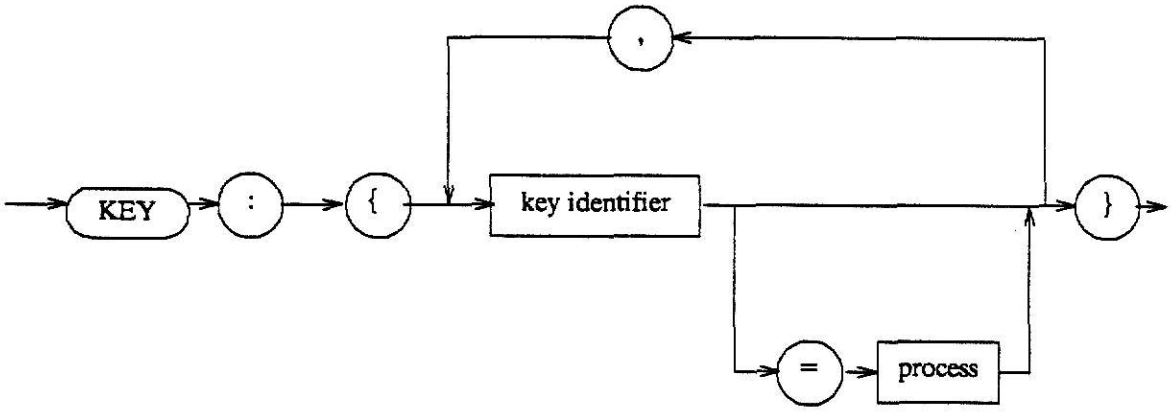


Figure 5. SPECIAL DATA TYPE

key field declatation



process

Figure 6. KEY FIELD DECLARATION

role declaration

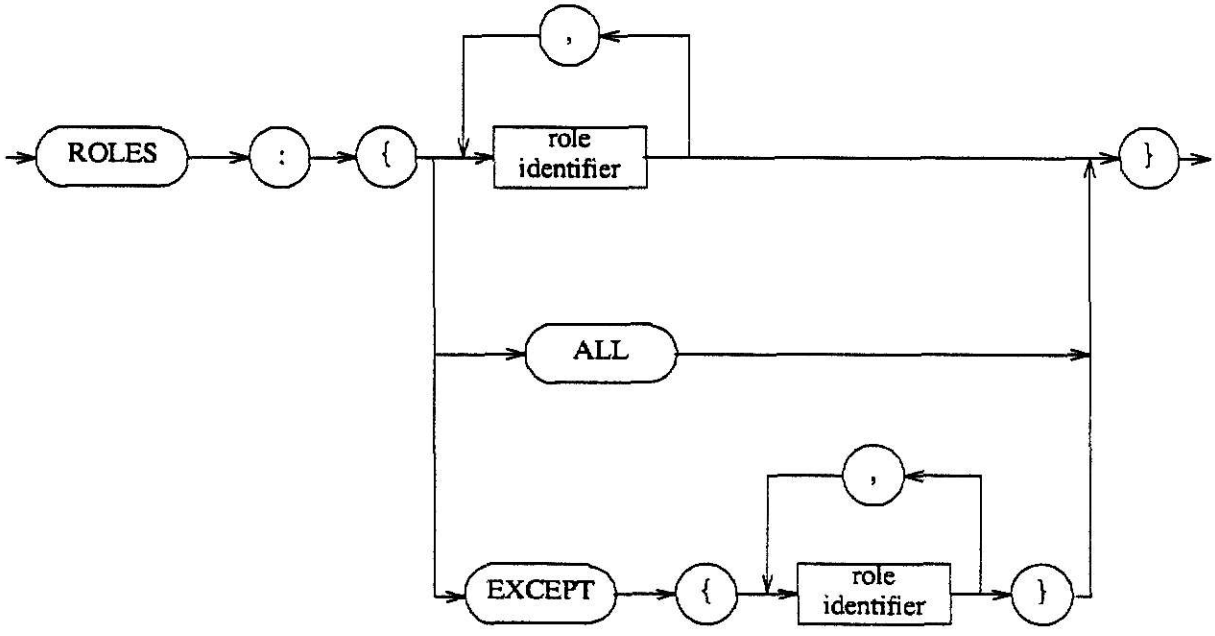


Figure 7. ROLE DECLARATION

multi-part declaration

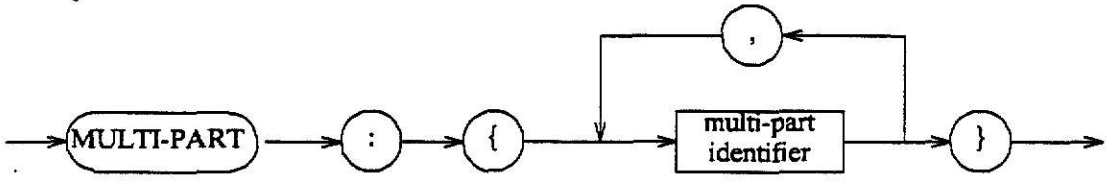


Figure 8. MULTI-PART DECLARATION

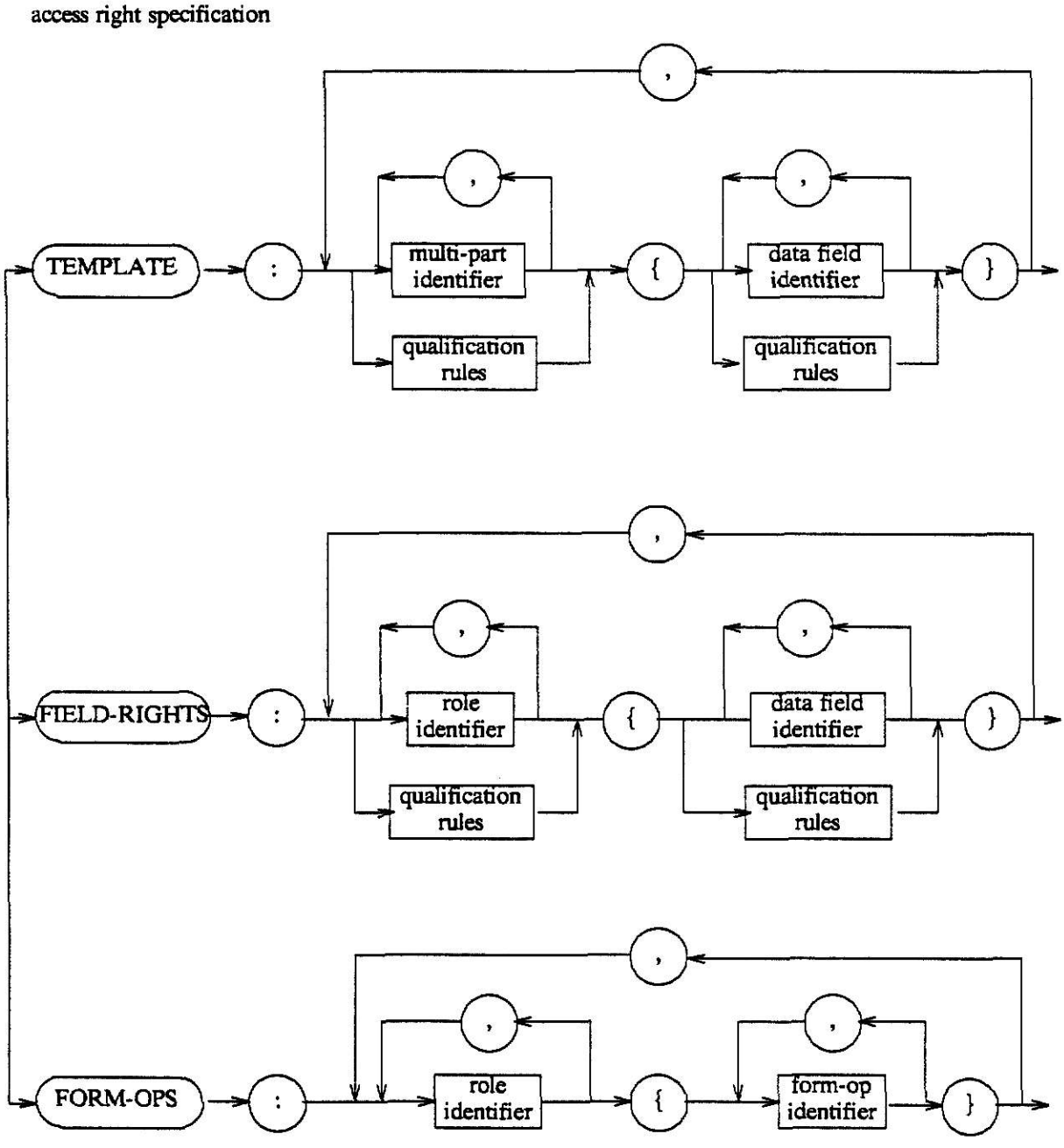
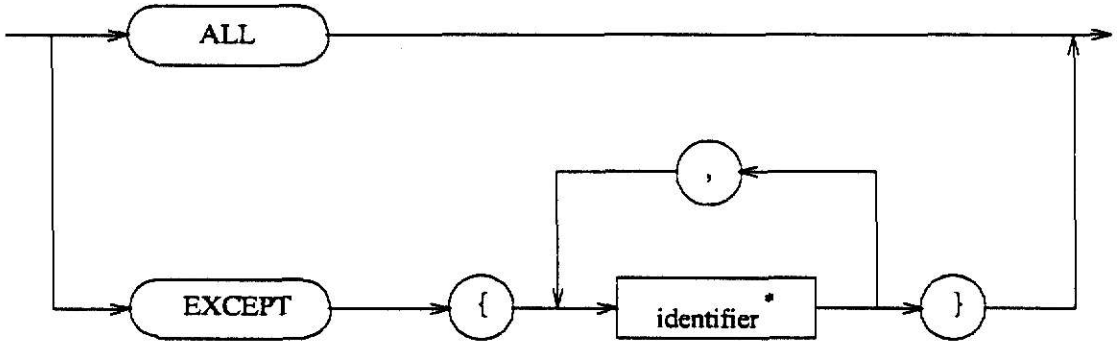


Figure 9. ACCESS RIGHT SPECIFICATION - FOR QUALIFICATION RULE SEE FIGURE 10

qualification rules



* identifier can be either:

role
multi-part
data field

Figure 10. QUALIFICATION RULES

data entry rules

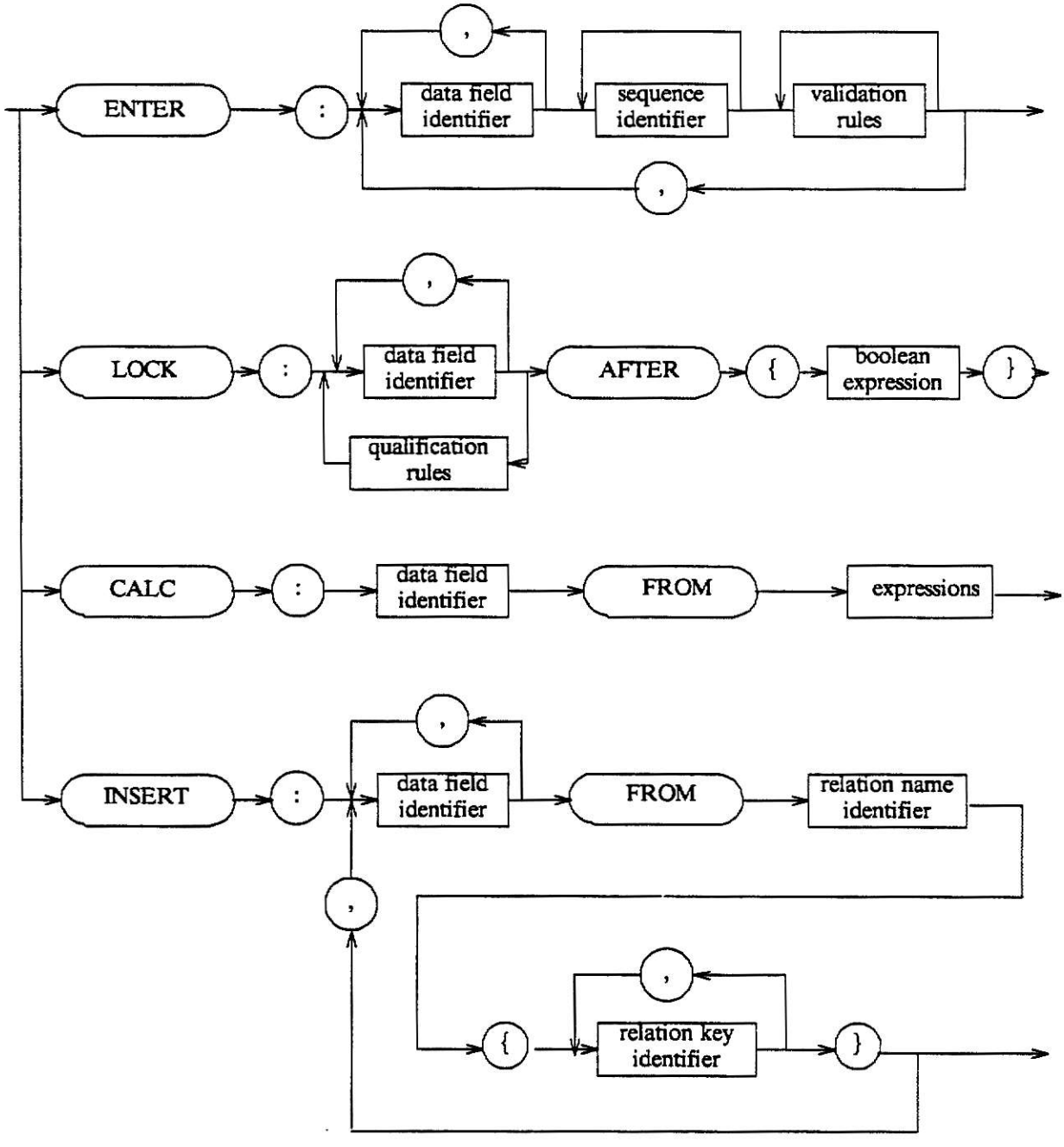


Figure 11. DATA ENTRY RULES

sequence identifier

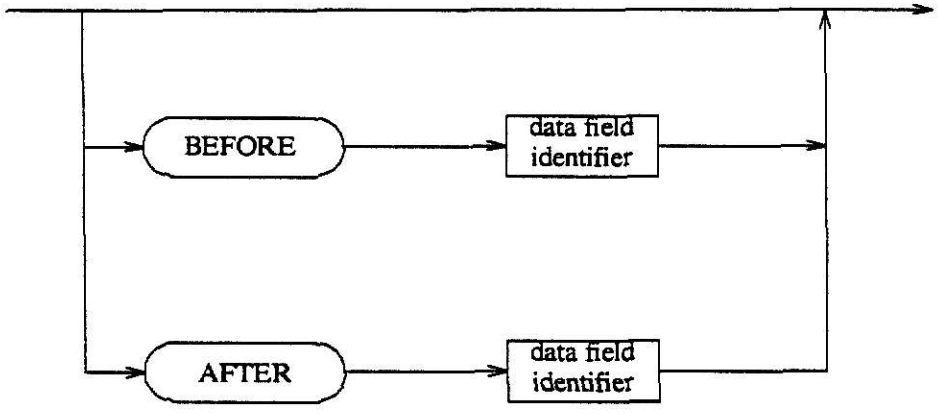


Figure 12. SEQUENCE IDENTIFIER

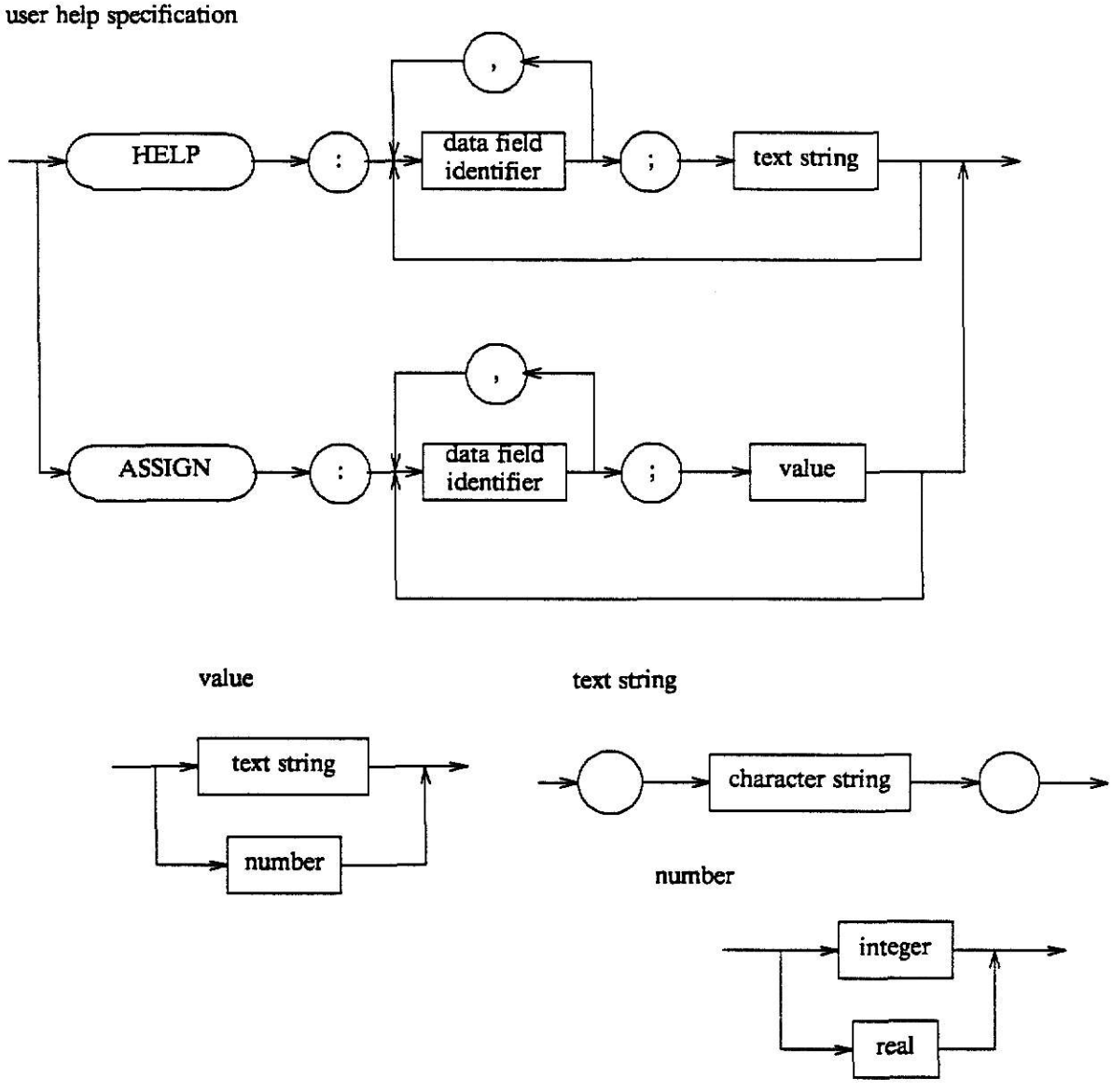
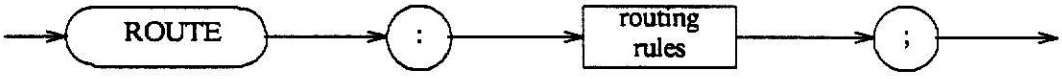
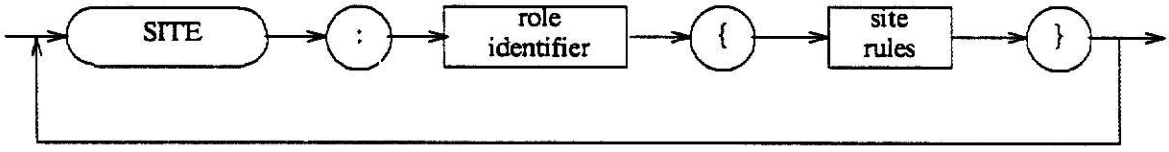


Figure 13. USER HELP SPECIFICATION

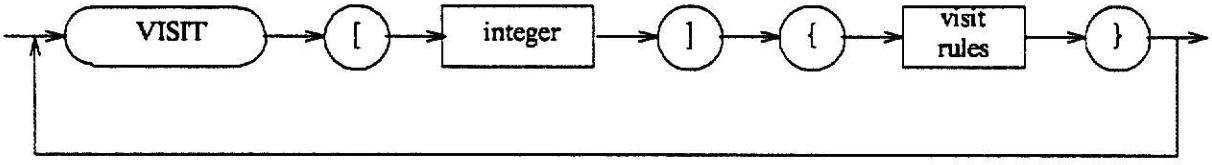
routing script



routing rules



site rules



visit rules

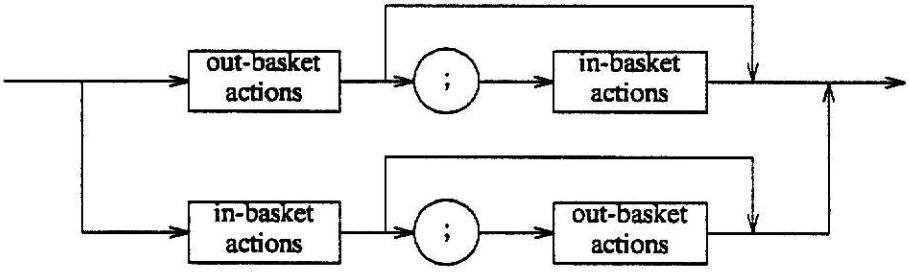


Figure 14. ROUTING SCRIPT

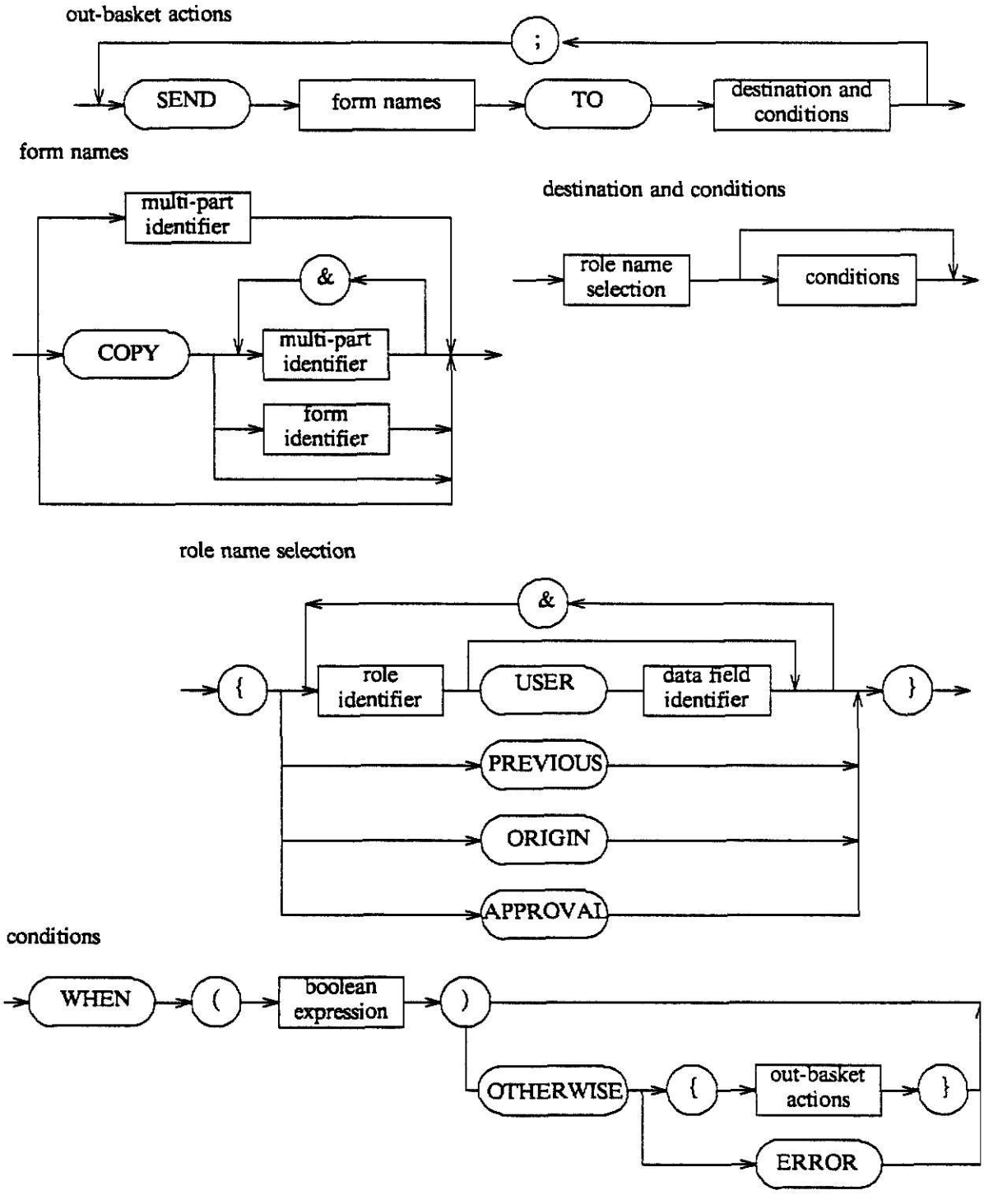


Figure 15. OUT-BASKET ACTIONS

in-basket actions

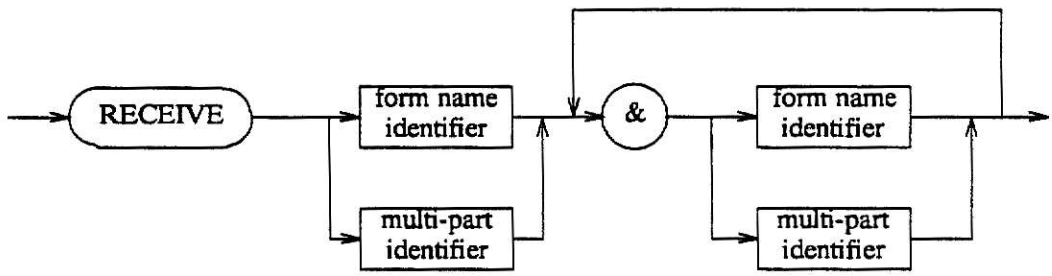
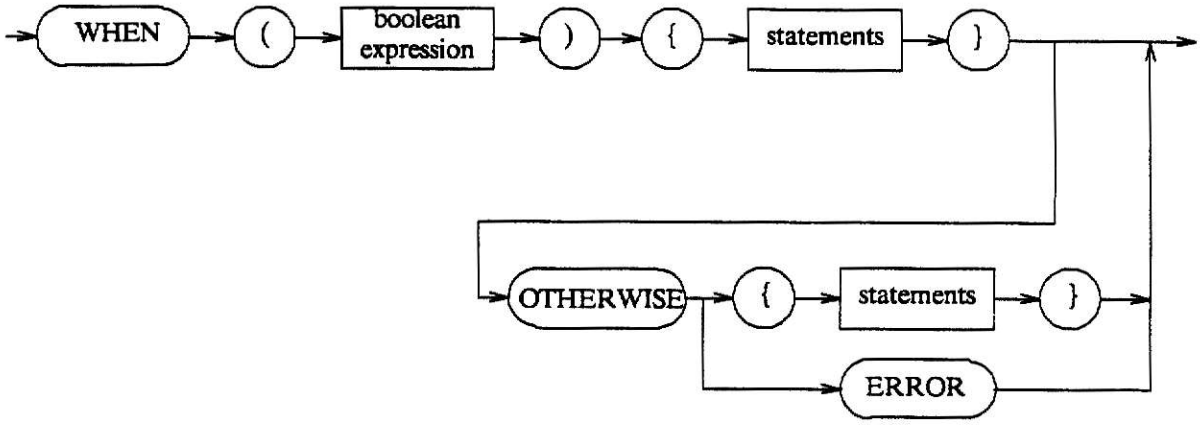


Figure 16. IN-BASKET ACTIONS

condition specifier



statements

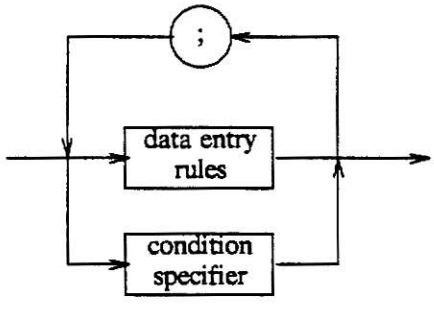
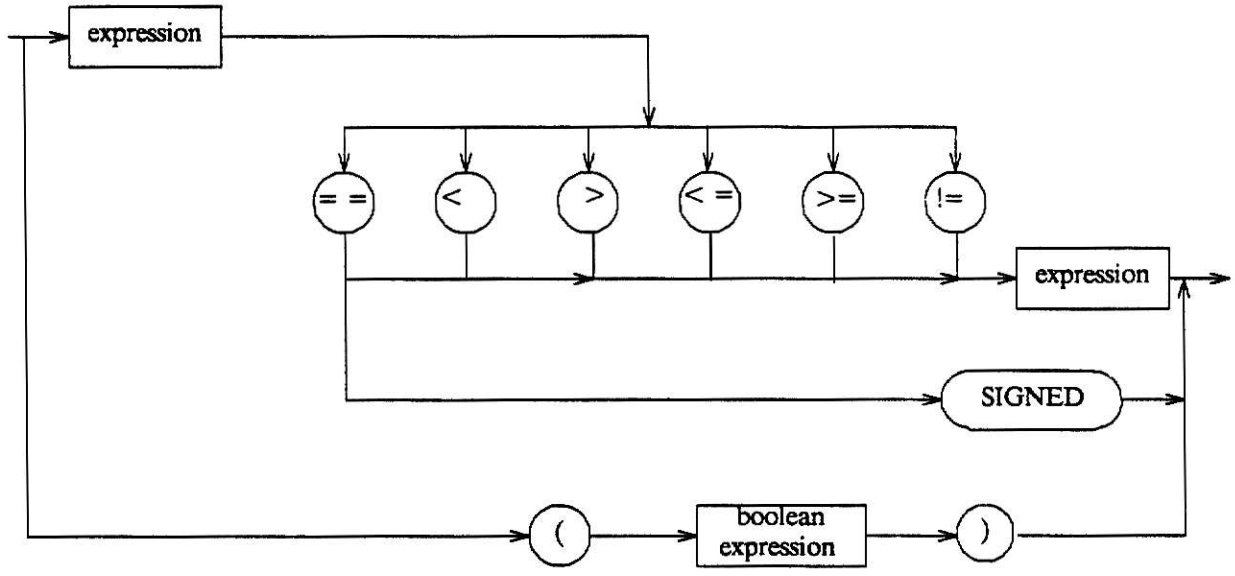


Figure 17. CONDITION SPECIFIER

boolean term



boolean expression

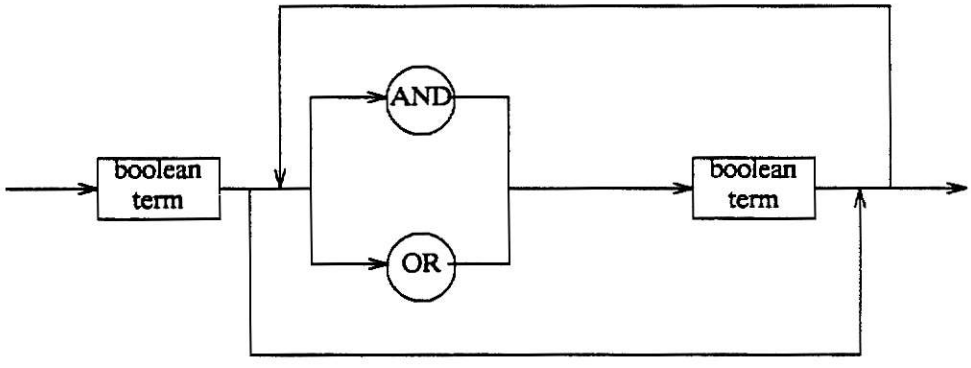
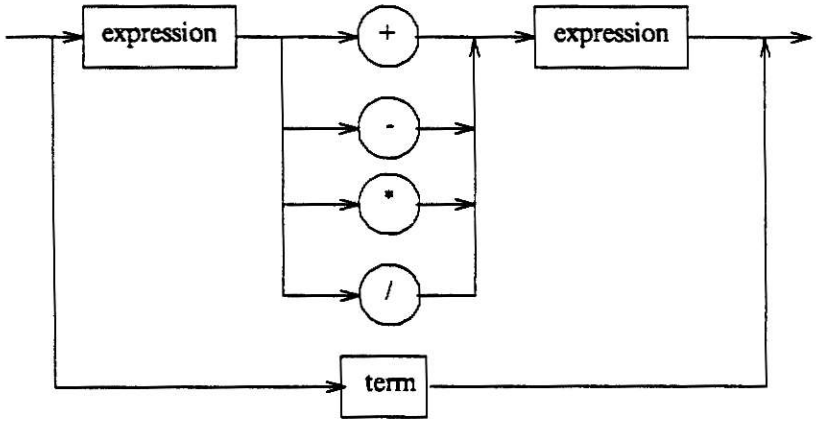


Figure 18. BOOLEAN EXPRESSIONS

expression



term

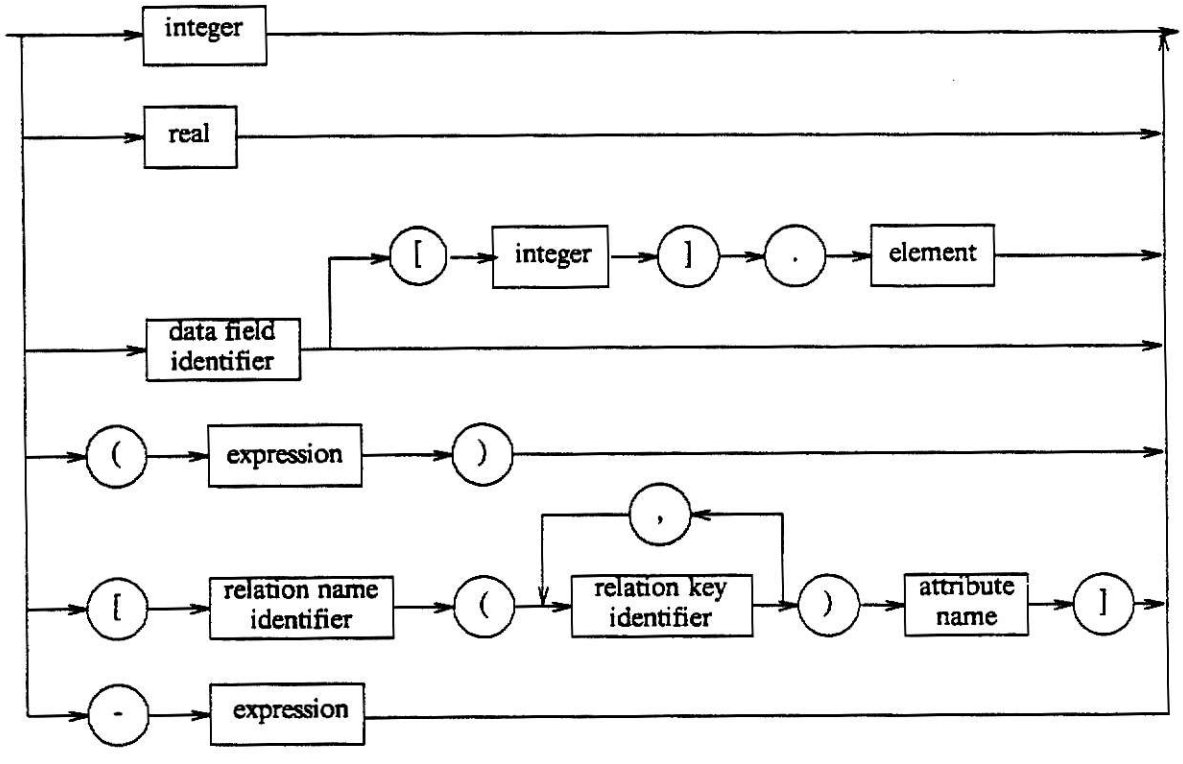


Figure 19. EXPRESSIONS

SUBMIT ORIGINAL AND 5 COPIES

KANSAS STATE UNIVERSITY

THE GRADUATE SCHOOL

PROGRAM OF STUDY FOR THE MASTER'S DEGREE

CURRICULUM

Indicate Plan:
Master's Thesis ☐
Master's Report ☐
Non Thesis-Report ☐

Last Name | First | Middle | Social Security Number

MAJOR COURSES

Department Name	Course Number	Course Name	Credits

Total Credits

SUPPORTING COURSES

Department Name	Course Number	Course Name	Credits

Total Credits

Advisory Committee

Major Professor — Name Typed _____ Signature _____

Committee Member — Name Typed _____ Signature _____

Committee Member — Name Typed _____ Signature _____

Approved by Head of Department _____ Date _____ Dean of Graduate School _____ Date _____

Typed copies of the program signed by major professor, at least two other committee members and the department head are forwarded to the Dean of Graduate School. (Head of department signs twice if he is a committee member.) Transfer of credits should be indicated and the name of the school given.

Rev 7-71

Figure 20. KSU PROGRAM OF STUDY

4. CHAPTER FOUR: FDL - IMPLEMENTATION

4.1 COMPILER CONSTRUCTION - LEXICAL AND SYNTACTICAL ANALYSIS

Chapter three provides a comprehensive set of instructions to allow a form designer to describe the contents of a form and what actions are to be taken. In this chapter, some of the considerations that are required to implement an FDL compiler are discussed. For this report the discussion about compiler construction for FDL will be limited to lexical and syntactical analysis.

4.1.1 GRAMMAR DEFINITIONS

Barrett [Barr79] stated that a grammar is a four-tuple containing a terminal alphabet, a nonterminal alphabet, a set of production rules made up of the terminal and nonterminal alphabet, and a start symbol. It was also stated that the set of terminals and the set of nonterminals were disjoint. Grammars can be categorized into four general classes [Barr79]; unrestricted, context-sensitive, context-free, and right-linear. Unrestricted is the most general class and is outside the scope of the above definition. Each other class of grammar is formally defined as [Barr79];

- Context-sensitive - Each production is of the form of x -> y where x and y are members of the union of terminals and nonterminals, such the x contains contains at least one member from the set of nonterminals and that the length of x is less than or equal to the length of y.
- Context-free - Each production is of the form x -> y where x is a member of nonterminals and y is a member of the union of terminals and nonterminals and y may be an empty string. Right linear - Each production has the form A -> xB or A -> x, where A and B are members of nonterminals and x is a member of terminals.

FDL is of the class context-free because each production production rule (refer to the syntax diagrams) has as its left side only member of the nonterminal set (e.g., form type -> FORM form-identifier : block END).

4.1.2 LEXICAL ANALYSIS

Lexical analysis is the process of analyzing a input stream to construct elementary units to be passed to a parser, these elementary units are typically called tokens. A token is a member of the terminal set. Some of the members of the token set for FDL are SEND, COPY, {, }, identifiers, etc. With the specification of the syntactic diagrams, the token (terminal) set was defined.

With the specification of tokens, the language designer can begin design and implementation of a lexical analyzer. At this point, the language designer can begin the design of a new analyzer or use an existing tool that is available to automatically construct such an analyzer. Unless the application is so unusual, the probable choice is to select an existing tool. One such tool is call LEX which is a facility of the UNIX (tm) operating system.

LEX [Lesk75] is a tool that accepts user submitted rules and associated actions to create a lexical analyzer. When a rule is recognized, the action is taken as specified by the designer. The rules are specified as regular expressions. The lexical analyzer generated by LEX is an interpreter representing a deterministic finite automaton (DFA). A DFA is a labeled directed graph with nodes called states and the labeled edges are called transitions. To be a DFA two conditions must be met; it has no transition on input from an empty string and for each state and with a input symbol there is a most one edge labeled with that input symbol leaving the state. A typical rule submitted to LEX may be of the form:

[A - Za - z][A - Za - z0 - 9]*

This regular expression matches all alphanumeric strings with a leading alphabetic character, this expression recognizes an identifier. The associated action with this rule may be the placement of the identifer into a symbol table and passing a pointer to the parser. LEX also provides for the handling of ambiguous situations by using a set predefined rules within LEX.

4.1.3 SYNTACTICAL ANALYSIS

Syntactical analysis is the process by which tokens, received from the the lexical analyzer, are examined to determine whether the submitted sentence is derivable from the defined production rules. The function that provides this capability is called a parser. Parsing can be accomplished by either one of two methods - top-down or bottom-up. The top-down approach determines the correctness of a sentence by beginning with the root node (start symbol) and working toward the leaves (terminal) of the derivation tree. The bottom-up approach begins with the leaves of the tree and works toward the root of the tree. A grammar that can be parsed using a top-down approach is known as LL(k), with k identifying the number of lookahead tokens. A grammar that can be parsed using bottom-up is known as LR(k).

4.1.3.1 LL(K) PARSERS

A LL(K) parser can be constructed with the use of push-down automaton that is deterministic. This can be accomplished by the construction of a LL(K) selector table. The selector table examines the symbol on top of the stack, the next K input tokens and based on this information, selects the next production rule to apply. If during the construction of the selector two or more production rules occupy the same cell, the grammar is considered ambiguous and not LL(K). The language designer must then respecify the offending production rules to correct this situation. Another parsing method is called the recursive decent parser. Recursive decent parsers use procedure calls. In order for this method to work correctly the source grammar must be LL(1) or the parser may enter an infinite loop. Both implementations require the removal of left recursion from the production rule set. Algorithms that derive these types of parsers are presented in Aho, et. al. [Aho77] and Barrett, et. al. [Barr79].

4.1.3.2 LR(K) PARSERS

A LR(K) parser has an input, a stack, and a parsing table. The parsing table consists of two parts, a action function and a go to function. The parser determines the current state on the top of the stack and the current input token. It then examines the ACTION table to determine what to give the current state value on the stack and the input token. An entry in the ACTION table can be: shift,

reduce, accept, error. The go to function takes a state and the grammar symbol to determine the next state. A shift is when a input token and the next state are pushed onto the stack. A reduce is when the number of tokens on the right side of an identified production rule is popped from the stack and the exposed state on the stack and symbol on the left side of the rule are used to determine the next state to be placed on the stack. Accept is when parsing is complete and a correct sentence has been detected. Error indicates that an invalid sentence has been detected. The construction of the parsing table is described in both Aho, et. al. [Aho77] and Barrett, et. al. [Barr79]. If the derivation of the parsing table for a grammar produces a shift/reduce conflict, that is, the parser cannot determine what to do based on the current state of the machine, then the grammar is not LR(K). The language designer then must respecify the rules to remove the ambiguity from the grammar.

4.1.4 YET ANOTHER COMPILER - COMPILER (YACC)

Because the algorithms for creating selector tables and parsing tables are well defined and the amount of manual effort required to produce a number of automatic parsing generators have been constructed, one of these generators is known as YACC and is a facility of the UNIX operating environment. The language designer specifies input to YACC as production rules in a BNF format. Along with the production rules, the start symbol, tokens, and precedence rules are identified. The output of YACC is a program capable of parsing the grammar described. YACC works with LEX to receive tokens for the grammar. The parser constructed by YACC is LALR (Look Ahead Left Right). The significant difference between a LR(K) parser and a LALR parser is the size of the tables produced. For further details concerning the construction of the LALR tables refer to Aho, et. al. [Aho77]. In addition to providing the production rules, the user may specify calls to routines (provided by the language designer) that take some action when either a part or entire production rule is identified. In order to handle ambiguous grammars YACC provides several rules for handling ambiguous situations [John78]. YACC provides two rules by default, they are; if a shift/reduce conflict is detected reductions are deferred in favor of shifting and if a reduce/reduce conflict is detected the rule stated first in the grammar specification is used. Both types of conflicts are

summarized in the output provided by YACC. Another set of rules for resolving conflict is the establishment of precedences and associativities for expressions.

With LEX and YACC it is possible to quickly implement a language and the experiment with the specification as opposed to being bogged down with the details of writing of the parsers.

4.1.5 SYMBOL TABLES

During the compilation process, information concerning the names appearing in the source is collected. This information is entered into a data structure called a symbol table. The data collected for each new name varies from implementation to implementation. Typically the information relates to the type, form (simple, structure, etc.), default values, etc. The primary issues in symbol table design are the format of the entries, method of access, and place of storage. It is not the intent of this report to analyze symbol table implementation but state that one is required to be defined along with its associated operations to support the compiler.

4.2 FDL IMPLEMENTATION

The implementation of FDL utilitizes LEX, YACC, the INGRES data base management systen, and application software using the C programming language. The syntax diagrams described in Figures 1-8 and Figures 14-19 have been converted to a BNF specification for input to YACC. The FDL compiler currently supports the syntax associated with data declarations and routing scripts.

The FDL compiler generates two tables. The first table describes where a form is to be forwarded and under what conditions this is to occur. The second table contains the boolean expression that must be evaluated to determine whether to route a form to a site based on the contents of the form. Prior to the construction ot these tables syntactic and semantic checks are made to insure that the defined routing is logical. The implementation details of FDL are provided in a separate report.

5. CHAPTER FIVE: SUMMARY

5.1 CONCLUSIONS

In the previous chapters a Form Definition Language (FDL) is specified to support the implementation of an Intelligent Data Object (IDO). An IDO is a form that has the capabilities to make decisions for itself based on the state of the form. With FDL a forms designer can specify the contents of an IDO and associate various actions to be taken with the content of the IDO. Some of these actions that can be specified are:

- Automatic calculation of data fields using the contents of other data fields or accessing a data field in a data base.
- Allow users, by the use of roles, to only see or update data fields they are authorized. The concept of "locking" the form after certain events is provided for so that receivers of documents can be reasonably assured that they have not been tampered with.
- To achieve a higher degree of automation, the designer can specify under what conditions a form is to be forwarded and to whom.

To support FDL, a data base management system was assumed to exist to support the compilation process. For instance, to support the function APPROVAL, it is assumed that a set of prespecified conditions for approval have been defined along with a system maintained directory of organization members is accessible. Another important data base item required is the ability to translate a logical named destination to a physical address to support automatic routing.

The constructs used to define statements were picked to model office terminology as closely as possible. For instance, ENTER, INSERT, COPY are used. Though the intent of the language design was targeted at someone with an understanding of programming, hopefully the language can evolve and attract a larger community of users.

5.2 FUTURE DIRECTIONS

The specified FDL is first cut at what should be contained in a form definition language. In this section, some recommendations are made as to what can be done to improve the usefulness of FDL. One area of consideration is the development of a graphics tool to allow a user to draw a route and specify conditions directly. This tool then can generate the necessary data structures and object modules to support routing; within the area of routing the development of an algorithm to detect abnormal situations (like a form being routed infinitely) at the time the form is being compiled would be useful. Further work needs to be done with the specification of repeating groups. The current process is awkward but does get the task done. Also, a consideration might be given to providing more triggering specifications. FDL, as specified, provides only one, wait for the arrival of two or more forms before releasing.

BIBLIOGRAPHY

[Aho77] Aho, A.V. and Ullman, J.D., "Principles of Compiler Design", Addison-Wesley Publishing Company, Reading, MA, 1977.

[Barr77] Barrett, W.A. and Couch, J.D., "Compiler Construction: Theory and Practice", Science Research Associates, Inc., Chicago, IL, 1979.

[Baum80] Baumann, L.S. and Coop, R.D., "Automated Workflow Control: A Key To Office Productivity", Proceedings For AFIPS Office Automation Conference, March 1980.

[Bern82] Bernal, M., "Interface Concepts For Electronic Forms Design And Manipulation", Office Information Systems, edited by N. Naffah, North-Holland Co., 1982.

[Byrd82] Byrd, Roy J. and Smith, Stephen E. and deJong, S. Peter, "An Actor- Based Programming System", ACM 0-89791-075-3/82/006/0067.

[Cook80] Cook, Carolyn L., "Streamlining Office Procedures--An Analysis Using The Information Control Net Model", AFIPS National Computer Conference, 1980, pp. 555-565.

[DeJo80] deJong, S.P. and Byrd, R.J., "Intelligent Forms Creation In The System Form Business Automation (SBA)", Research Report RC 8529, Computer Science Dept. IBM T. J. Research Center, Yorktown Heights, New York 10598, October 1980.

[Deog80] Deogun, Jitender S., "Conceptual Development of Office Automation Models", Proceedings of the 16th Annual International Conference On System Sciences, Vol. 1, 1983.

[DiPi83] DiPirro, J.E. and Ferrans, J.E. and Juszczak, C., "A Form Management System For Switching Database Administration", Proceedings IEEE International Conference On Communications, Boston, MA, June 19-22, 1983.

[Elli80] Ellis, Clarence A. and Nutt, Gary J., "Office Information Systems and Computer Science", Computing Surveys, Vol. 12, No. 1, March 1980.

[Elli82] Ellis, Clarence A. and Bernal, Marc, "Officetalk-D: An Experimental Office Information System", ACM 0-89791-075-3/82/006/0131, 1982.

[Ferr82] Ferrans, James C., "SEDL - A Language For Specifying Integrity Constraints On Office Forms", Proceedings ACM-SIGOA Conference On Office Information Systems, Philadelphia, PA, June 21-23, 1982, pp. 123-130.

[Geha82] Gehani, Narain, "The Potential Of Forms In Office Automation", IEEE Transactions On Communications, Vol COM-30, No. 1, January 1982, pp. 120-125.

[Geha83] Gehani, N.H., "High Level Form Definition In Office Information Systems", The Computer Journal, Vol. 26, No. 1, 1983.

[Hamm77] Hammer, M. and Howe, W.G. and Kruskal, V.J., and Wladowsky, I., "A Very High Level Programming Language for Data Processing Applications", Communications of the ACM, Vol. 20, No. 11, November 1977.

[Hamm79] Hammer, M. and Zisman, M., "Design and Implementation of Office Information Systems", NYU Symposium on Automated Office Systems, May 17-18, 1979, pp. 13-23.

[Hamm80] Hammer, Michael and Kunin, Jay S., "Design Principles Of An Office Specification Language", Proceedings AFIPS Office Automation Conference, March 1980.

[Hewi77] Hewitt, Carl and Baker, Henry Jr., "Actors and Continuous Functionals", Library For Computer Science, Massachusetts Institute of Technology, MIT/LCS/TR-194, December 1977.

[Hogg81] Hogg, J. and Nierstrasz, O. and Tsichritzis, D., "Form Procedures", Omega Alpha, D. Tsichritzis, Ed., CSRG Tech. Rep. 127, Univ. of Toronto, 1981.

[John78] Johnson, S.C., "YACC: Yet Another Compiler - Compiler", Computing Science Technical Report #32, Bell Telephone Laboratories, July 31, 1978.

[Kons82] Konsynski, Benn R. and Bracker, Lynne C. and Bracker, William E., "A Model For Specification Of Office Communications", IEEE Transactions On Communications, Vol. Com-30, No. 1, January 1982, pp. 27-36.

[Lebe82] Lebensold, J., and Radhakrishnan, T. and Jaworski, W.M., "A Modelling Tool for Office Information Systems", 1982 ACM 0-89791- 075-3/82/006/0141.

[Lesk75] Lesk, M.E., "Lex- A Lexical Analyzer Generator", Computing Science Technical Report #39, Bell Telephone Laboratories, October 1975.

[Lisk75] Liskov, B. and Zilles, S., "Programming With Abstract Data Types", SIGPLAN, April 1974.

[Lum82] Lum, V.Y. and Shu, N.C. and Tung, F. and Chang, C.L., "Automating Business Procedures with Form Processing", Office Information Systems, North Holland Publishing Company, 1982.

[Maze84] Mazer, M.S. and Lochovsky, F.H., "Logical Routing Specification in Office Information Systems", ACM Transactions on Office Information Systems, Vol. 2, No. 4, October, 1984, pp. 303-330.

[McBr83] McBride, R. A. and Unger, E. A., "Modeling Jobs In A Distributed System", ACM 0-89791-123-7/83/012/0032, 1983.

[McLe76] McLeod, D.J., "High Level Domain Definition in a Relational Date Base System", Proceedings 1976 ACM SIGMOD-SIGPLAN Conference on Data, Salt Lake City, Utah, March 22-24, 1976.

[Rive83] Rivest, R.L. and Shamir, A. and Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, Vol. 26, No. 1, January 1983.

[Shu82] Shu, N.C. and Lum, V.Y. and Tung, F.C. and Chang, C.L., "Specification of Forms Processing and Business Procedures for Office Automation", IEEE Transactions on Software Engineering, Vol. SE-8, No. 5, September 1982.

[Tsic82] Tsichritzis, D.C., "Forms Management", Communications of the ACM, Vol. 25, No. 7, July 1982. pp. 453-478.

[Tsic80] Tsichritzis, D., "OFS: An Integrated Form Management System", Proceedings Of The ACM International Conference On Very Large Data Bases, 1980.

[Vitt81] Vittal, John, "Active Message Processing: Messages As Messengers", North - Holland Publishing Company, IFIP, 1981.

[Wint85] Winter, C., "The Next Big High-Tech Boom in Office", Chicago Tribune, April 28, 1985, Sec. 7, p. 3-4.

[Woo84] Woo, C.C. and Lochovsky, F.H., "Authorizations in a Computer-Based Office Information System", CH2019-7/84/0000/0081, 1984 IEEE.

[Yao84] Yao, S.B. and Hevner, A.R. and Shi, Z. and Luo, D., "FORMANAGER: An Office Forms Management System", ACM Transactions on Office Information Sysatems, Vol. 2, No. 3, July, 1984, pp. 235-262.

[Zism77] Zisman, Michael D., "Representation, Specification, and Automation of Office Procedures", A Dissertation In Decision Sciences, University of Pennsylvania, Business Administration, 1977.

[Zloo81] Zloof, M.M., "QBE/OBE: A Language For Office And Business Automation", IEEE Computer, May 1981, pp. 13-22.

[Zloo77] Zloof, M.M., "Query By Example: A Data Base Language", IBM Systems Journal, Vol. 16, No. 4, December 1977.

[Zloo80] Zloof, M.M., "A Language For Office and Business Automation", IBM Research Report RC8091, Yorktown, New Jersey, January 1980.

FORM DEFINITION LANGUAGE FOR INTELLIGENT DATA OBJECTS

by

RICHARD P. SEWCZWICZ

B.S., Christian Brothers College, 1970

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

An Abstract of a Masters Report

An Intelligent Data Object is defined as an instance of an intelligent abstract data type. In this report the IDO is used to describe an office form. To support the implementation of the IDO a collection of subsystems are being designed. Collectively these subsystems are known as the IDO Management System (IDOMS). Within this report the Form Definition Language subsystem is defined.

The Form Definition Language (FDL) provides to the user a set of language constructs to describe a form and the actions that can be performed on it. The FDL allows the user to make data declarations, describe data entry requirements, and the route a form can take based on the contents of defined data fields. The FDL is described with a set of syntax diagrams and associated text to describe the function of each FDL statement.

Also within the report a review of office procedure specification languages and Office Information Systems are presented. This review provides the foundation for the design of FDL.