CITI Technical Report 97-1

MICA: A MIB Compiler in Java

David T. Nettleman

Center for Information Technology Integration
University of Michigan
Ann Arbor

ABSTRACT

Management Information Bases (MIBs) define the attributes of objects managed by the Simple Network Management Protocol (SNMP). An SNMP application uses the information in one or more MIBs to learn the attributes of the objects supported by the managed devices in a network. MIBs in source form are not directly usable; they must first be compiled into a format acceptable to the SNMP application. Java's built-in network facilities make it an ideal language for writing SNMP applications, so the need arises for a Java-based MIB compiler. The MICA project is the design and implementation a Java class library that treats a MIB as an abstract data type (ADT). Java-based SNMP applications use this ADT to compile MIBs and extract their information.

September 3, 1997

Center for Information Technology Integration
University of Michigan
519 West William Street
Ann Arbor, MI 48103-4943

MICA: A MIB Compiler in Java

David T. Nettleman

Center for Information Technology Integration
University of Michigan
Ann Arbor

1.0 Introduction

1.1 Motivation

The Java programming language greatly simplifies the task of writing network applications, when compared with earlier languages. In particular, Java provides high-level abstractions that allow the programmer to manage Uniform Resource Locators (URLs) and socket communications quite easily. In addition, Java offers excellent portability. These factors suggest that an increasing number of network applications will be written in Java. The need therefore arises for Java-based tools to support these applications.

One such application is the Simple Network Management Protocol, or SNMP, which can be used to manage devices in TCP/IP-based networks. The semantics of the objects retrieved from a managed device by SNMP are coded in files known as Management Information Bases, or MIBs.

Although several Java-based SNMP applications are available without fee (for example, see "West's SNMP Stack in Java" [West96] and "Java SNMP Control Applet" [Nik96]), none of them has MIB capability. Thus, these applications can retrieve management information from networked devices, but cannot associate semantics with the information so retrieved. This paper describes the design and implementation of MICA, a MIB Compiler in Java. MICA is a freely-available tool intended to provide MIB capability to Java-based SNMP applications.

To appreciate the significance of MIBs, a basic understanding of network management concepts and SNMP is necessary. The rest of this section is devoted to providing the needed background information in these areas.

1.2 Overview of SNMP Network Management

1.2.1 Background

In the early days of TCP/IP networks, network management functions were performed in an ad hoc manner. According to Stallings [Stal96], one of the few tools available for network management during that period was the Internet Control Message Protocol (ICMP). ICMP supports an "echo" mechanism that can be used to determine whether or not communication is possible with a given device. As networks grew in size and complexity, however, the need arose for a more sophisticated management scheme.

The scheme eventually adopted by the Internet Architecture Board (IAB) was SNMP. SNMP is a network management framework for TCP/IP-based internets. The three parts of the framework are:

- A set of rules for defining and identifying the types of information to be managed, known as the Structure of Management Information, or "SMI".
- A set of actual definitions for basic types of information that all managed devices are expected to support. This set of definitions is known as Management Information Base II, or "MIB-II". Like all MIBs, MIB-II is coded in the format specified by SMI.
- An application-layer protocol, known as "SNMP". This protocol defines the format of messages exchanged by the entities in a managed network. (Note that the term "SNMP" can be used to refer either to the application-layer protocol, or to the entire management framework. The intended meaning must be drawn from the context).

This paper deals with Version 1 of SMI and of SNMP.

1.2.2 The Roles of SNMP Entities

Before exploring the three parts of the framework in more detail, some comments on the roles of the entities that implement the framework are in order. Figure 1 shows a high-level view of a managed network. The network consists of one or more network management stations (NMS's) and potentially many managed devices, such as routers, servers, and hubs. Usually, the NMS's themselves are also managed devices.

Although a single NMS per management domain would suffice, it is desirable to have more than one so that network management functions will still be accessible should an NMS (or its network connection) fail. The NMS and the managed device are the two "active" components of an SNMP internet. An NMS sends a request to a managed device; the device responds with the requested information.

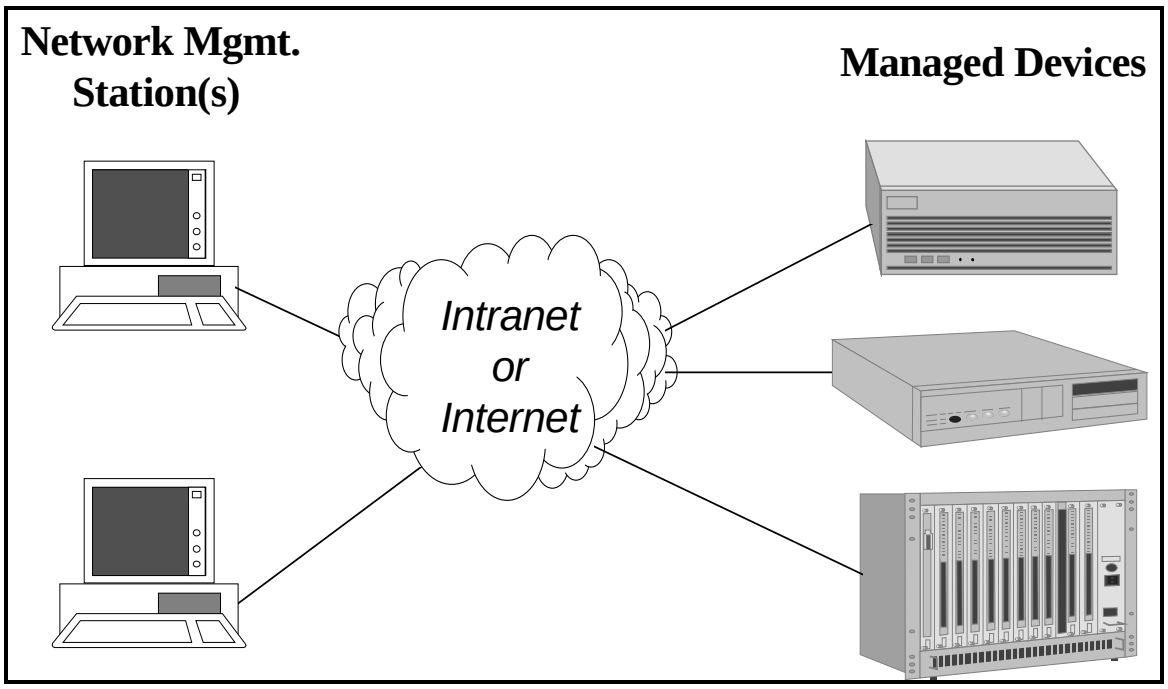

Figure 1: High-Level View of a Managed Network

Figure 2 shows a schematic view of an NMS and a managed device.

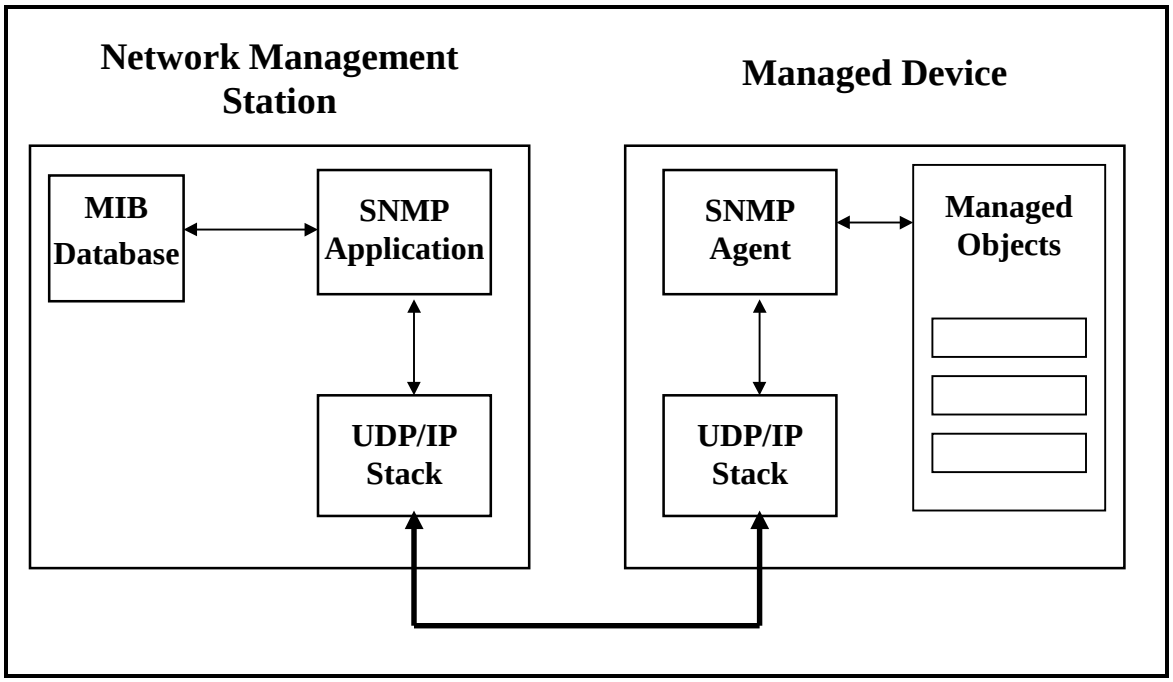

Figure 2: Schematic View of SNMP Entities

An NMS provides a user with access to network management functions. It executes an SNMP application, which accepts network management requests from the user and translates them into SNMP messages. The application sends the message to UDP port 161 of the target device and waits for the device to respond. It then formats the response and presents it to the user.

In order to provide the user with a menu of the types of information available at each device, the SNMP application consults a MIB database. This database is formed by compiling the MIB modules associated with each type of device in the network. MIB-II, which contains definitions applicable to all managed devices, is also usually compiled into the database. The MIB compiler may be integrated into the SNMP application or may be a separate program.

Each managed device is responsible for maintaining an appropriate set of objects, which are instances of the object types defined in one or more MIB modules. An NMS can access a device's objects only by sending requests to the SNMP agent executing at the device. The agent uses the object identifier(s) in the request to determine which of the device's objects are to be acted upon. Since the object identifiers are well-known (they are published in MIB modules), this arrangement allows an NMS to refer to a device's objects without needing to know anything about how the device implements those objects.

Having considered the roles of the entities in a managed network, let us now examine the three parts of the SNMP management framework: SMI, MIB-II, and the SNMP Protocol.

1.2.3 Structure of Management Information

The purpose of SMI is to define a scheme by which each class of management information can be named and described. Each such class is termed an "object type". The names and descriptions of object types are coded in MIB modules. Version 1 of SMI is defined in RFC 1155 [Rose91a].

An object type is identified via a sequence of integers known as an object identifier, or "OID". SMI defines a hierarchy to be used when assigning OIDs to object types. The sequence of integers comprising an OID reveal that object type's place in the hierarchy, in much the same way that the Dewey decimal system can be used to classify books.

Since SMI specifies the format of MIBs, it must be thoroughly understood before writing a MIB compiler. Section 2 of this report therefore covers SMI in greater detail.

1.2.4 MIB-II

The IAB has defined a standard MIB to contain a core set of object types common to most managed devices. This MIB is called "MIB-II", and is defined in RFC 1213 [Rose91c]. It includes 190 object type definitions, covering such things as device configuration (e.g., the device's name, number and type of network interfaces, routing table contents), status (such as the state of a TCP connection), and statistics (number of datagrams sent/received, number of IP packets successfully fragmented by the device, etc.). The object types are organized by "group"; if a given group, such as TCP, does not apply to a device, then the device need not implement the object types in that group.

MIB-II provides a degree of uniformity by standardizing the names and interpretations of common types of information. Vendors often supplement MIB-II by providing "enterprise MIBs" for their devices, which define object types for information specific to their devices.

1.2.5 SNMP

The SNMP protocol is defined in RFC 1157 [Case90]. Network management stations, executing SNMP applications, function as clients. The managed devices, executing SNMP agents, act as servers. An explicit goal of the designers was to keep the protocol as simple as possible, especially for the agents. Their plan was to keep the functionality required of agents to a minimum, to allow agents to be implementable even on small devices. This is one reason why SNMP specifies the use of a connectionless (unreliable) transport layer.

Clients and servers using SNMP communicate by exchanging messages. Clients can issue messages to retrieve or set the values of managed objects at a server (device). The client's message indicates the type of request and the object ID(s) to be acted upon. The server responds with a message indicating the success or failure of the request, and if successful, the values of any requested objects.

SNMP must also deal with the subject of object instance identification. Although SMI specifies the format of OIDs for object *types*, it does not describe the format of OIDs for the *instances* corresponding to those object types; that task is delegated to SNMP. SNMP forms the OID of an object instance by appending a suffix to the OID of the instance's object type. The form of the suffix varies depending on the object type involved. This will be further explained in section 2.2, "Identification of Managed Objects".

1.3 Related Work

Several non-Java MIB compilers are available without fee. These include SMICng [Perkins94] and SNMX [SNMX], both of which are written primarily in C. Advent Network Management, Inc. has developed a Java-based MIB compiler [Advent], but its source code is not freely available.

The remainder of this paper is organized as follows. Section 2 describes SMI in detail. Section 3 discusses the design of MICA. Section 4 describes how to use MICA with an SNMP application. Section 5 offers conclusions and suggestions for future work. The lexical and parsing rules used by MICA are listed in Appendix A, followed by code copyright notices in Appendix B, and lastly acknowledgements and references.

2.0 Structure of Management Information

2.1 Purpose

In order for the SNMP implementations of various vendors to interoperate, a global scheme for identifying and describing managed objects is needed. SMI specifies the mechanisms to be used for this purpose.

2.2 Identification of Managed Objects

The International Organization for Standardization (ISO) and the International Telecommunication Union (ITU, formerly known as CCITT) have agreed upon a naming tree that can be used to uniquely identify any object. Here, the term "object" means not just the managed objects used by SNMP, but also such things as standards organizations, the standards themselves, vendors, etc. Each object to be identified is placed at a unique position in the tree, and the path from the root to the object is used to form the object's OID. Each node in the tree is also given a mnemonic for human convenience.

Figure 3 shows a portion of the naming tree. All object types from which managed objects are instantiated appear at leaves of the tree. The figure shows the path from the root to two (arbitrarily chosen) leaves: "sysUpTime" from MIB-II and "alMcpuRtVer" from an enterprise MIB of Alantec Corp (sysUpTime represents the length of time since the device was last initialized; alMcpuRtVer represents the software version running on the motherboard CPU of an Alantec PowerHub device). The number in parentheses following each node's name indicates the node's child number with respect to its parent. These numbers, separated by periods, are used to form an OID for each node. For example, the OID for the "sysUpTime" object type is 1.3.6.1.2.1.1.3, and that for the "alMcpuRtVer" object type is 1.3.6.1.4.1.390.2.1.2. Since the SNMP management framework is "owned" by the IAB, the OIDs of all MIB object types begin with the prefix of the "internet" node: 1.3.6.1.

The authority to add nodes to the naming tree is delegated. For instance, the IAB has authority to manage the "internet" subtree. The IAB has defined a "private" node in that subtree, and delegated the management of the subtree under that node to the Internet Assigned Numbers Authority (IANA). The IANA has allocated an "enterprises" node in the "private" subtree, under which nodes for vendors and other organizations are allocated. The IANA allocates these nodes by assigning a unique child of the "enterprises" subtree to each vendor or organization that applies for one (although the "enterprises" subtree currently has over 3000 children, only the one assigned to Alantec Corp. is shown in the figure). This arrangement allows vendors to uniquely identify object types that are specific to their products, since each vendor can manage its own portion of the naming tree as it sees fit. The MIBs in which vendors publish the definitions of their object types are known as "enterprise MIBs".

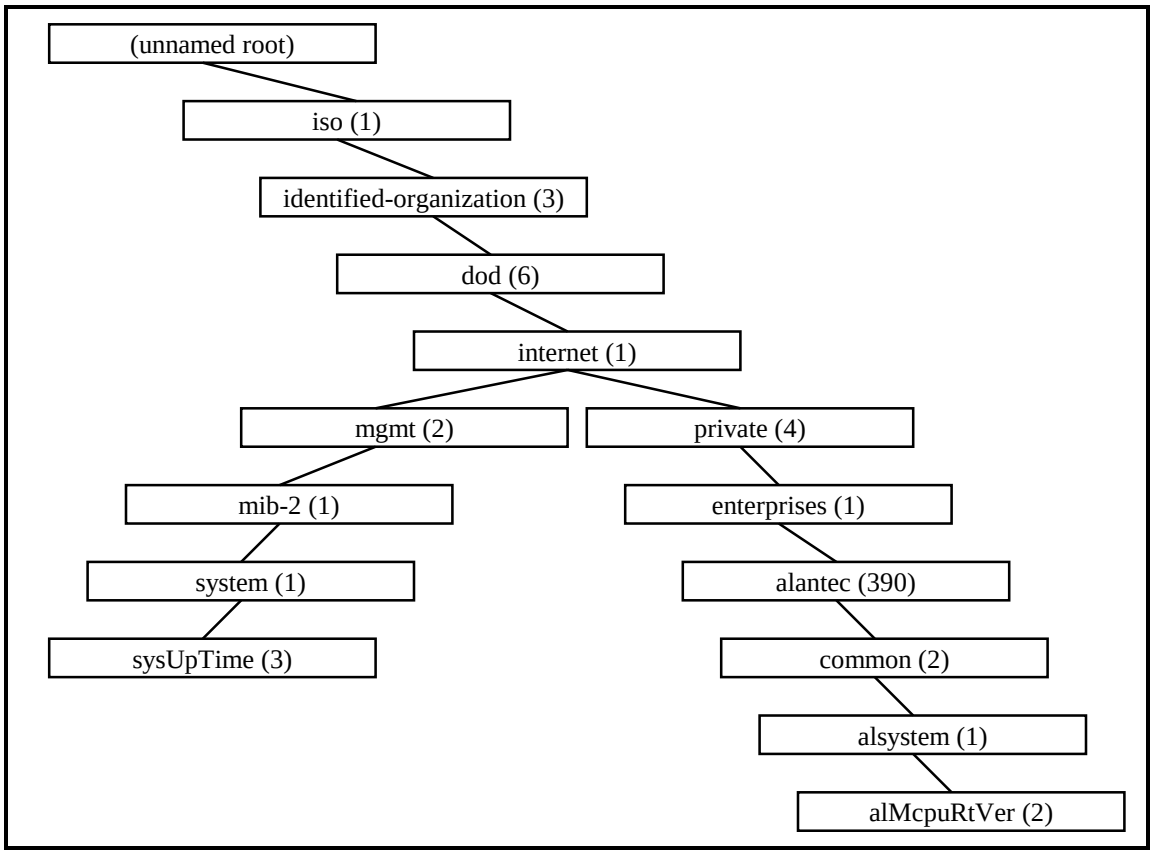

Figure 3: Part of the ASN.1 Object Naming Tree

Notice that the naming tree is used to derive OIDs for object *types*, not instances. Since the managed objects that SNMP deals with are *instances* of object types, the SNMP protocol must define the scheme for assigning OIDs to instances. There are two cases to consider, referred to as "scalar" and "columnar" [Rose91b]. A scalar object type is one having a single instance per device; a columnar object type can have multiple instances per device. Since only a single instance of a scalar object type can exist at a device, there is a one-to-one correspondence between scalar object types and their instances. SNMP identifies an instance of a scalar object type by simply appending ".0" to the OID of the scalar object type. Thus, since the OID for object type "sysUpTime" is 1.3.6.1.2.1.1.3, the OID for the (one and only) instance of sysUpTime at a particular managed device is 1.3.6.1.2.1.1.3.0.

Columnar object types can have multiple instances per device. Consider "tcpConnState", which represents the state of a TCP connection at a device. There is one instance of tcpConnState per TCP connection currently in use at the device. Therefore, simply adding a ".0" to the OID of the object type is not sufficient for distinguishing between the instances. The instance identification convention for columnar object types varies; each columnar object type must specify its convention in the MIB where it is defined. In the case of tcpConnState, the socket identifiers of the two endpoints of the TCP connection are used for this purpose. For example, suppose device "A" has IP address 141.199.4.1, device "B" has address 211.11.126.2, and port 23 of device "A" forms a TCP connection with port 4000 of device "B". Since the OID for object type tcpConnState is "1.3.6.1.2.1.6.13.1.1", the OID for the instance of tcpConnState for that TCP connection at device "A" is "1.3.6.1.2.1.6.13.1.1.144.199.5.1.23.211.11.126.2.4000". (Device "B" would have a corresponding instance, but in its OID, B's socket identifier would appear before A's).

2.3 Describing the Attributes of Managed Objects

In addition to the identification scheme just described, a method is needed for associating some semantic information with the OID of each managed object. The method specified by SMI for this purpose is to collect the semantic information in modules called MIBs.

2.3.1 ASN.1: The Language of MIBs

MIBs are coded in a language called "Abstract Syntax Notation One", or simply ASN.1 [OSI87a]. (According to Larmouth [Lar94], the original abbreviation was "ASN1", but this proved confusing as it was sometimes misread as "ANSI", the abbreviation for the American National Standards Institute—thus, the period was added!). ASN.1 allows data structures to be described abstractly, that is, without reference to the way the structures are to be represented when they are implemented in some actual programming language. Thus, it is possible in ASN.1 to declare a particular data item as being an INTEGER, and even to specify the value(s) over which the item may range, but it is not possible to specify whether negative values are represented in two's complement, nor in what order the octets comprising the item are to be transmitted. Similarly, the representation of character data (e.g., ASCII, EBCDIC) cannot be specified via ASN.1. (Specification of representation details is the responsibility of a *transfer syntax*. The transfer syntax used by SNMP is known as "Basic Encoding Rules", or BER [OSI87b]).

ASN.1 is a fairly rich language, and if there were no restrictions placed on its use, MIBs written by different parties might look quite different from each other. To solve this problem, SMI restricts MIB authors to a subset of ASN.1, and defines an ASN.1 macro called "OBJECT-TYPE" to be used for defining object types. A companion document entitled "Concise MIB Definitions" (RFC 1212 [Rose91b]) further standardizes the coding of MIBs. Together, SMI and the Concise MIB Definitions enforce a consistent style on MIB authors, thereby simplifying the task of writing a MIB compiler.

2.3.2 MIB Layout and Constructs

The format of MIB modules can now be described. Figure 4 shows an excerpt from MIB-II. A MIB module starts with the MIB name, followed by the body of the MIB enclosed in a BEGIN/END pair. The body of the MIB consists of an (optional) IMPORTS section, followed by the MIB definitions. The IMPORTS section allows definitions from other MIBs to be used in the current MIB. The body of the MIB can contain several types of constructs, two of which are especially important for MIB compilers: OBJECT IDENTIFIER and OBJECT-TYPE.

OBJECT IDENTIFIER is a built-in ASN.1 type. It allows a symbolic name to be associated with an OID. For instance, in Figure 4, an OBJECT IDENTIFIER declaration is used to equate the name "mib-2" with the OID "mgmt.1". The definition of "mgmt" is imported from MIB RFC1155-SMI, in which an OBJECT IDENTIFIER declaration is used to associate "mgmt" with OID "internet.2". Tracing backwards like this makes it possible to arrive at a completely numeric OID for any name. The numeric OID for "mib-2" winds up being "1.3.6.1.2.1" (as can be seen from Figure 3). OBJECT IDENTIFIER declarations can be used to associate names with nodes in the ASN.1 object naming tree but cannot be used to define object types for managed objects; the OBJECT-TYPE macro is used for this purpose.

OBJECT-TYPE is an ASN.1 macro that was defined by SMI and subsequently refined by the Concise MIB Definitions document. It not only associates a name with an OID (as OBJECT IDENTIFIER does), but also defines the attributes of a managed object. Figure 4 shows the OBJECT-TYPE definition for object "sysUpTime". The meaning of the macro's clauses are as follows:

- SYNTAX: the data type of the object type.
- ACCESS: the read/write status of the object type.
- STATUS: an indication of the support level of this object type.
- DESCRIPTION: A textual description of the object type.

The OBJECT-TYPE macro concludes with the assignment of an OID to the object type. "sysUpTime" is assigned OID "system.3", which becomes "1.3.6.1.2.1.1.3" when the "system" label is iteratively resolved.

A MIB module must be contained in a single ASCII file, although multiple MIB modules can exist in a single file.

```
    RFC1213-MIB DEFINITIONS ::= BEGIN

        IMPORTS
            mgmt, NetworkAddress, IpAddress, Counter, Gauge,
            TimeTicks
                FROM RFC1155-SMI
            OBJECT-TYPE
                FROM RFC-1212;


        mib-2      OBJECT IDENTIFIER ::= { mgmt 1 }
        system     OBJECT IDENTIFIER ::= { mib-2 1}

sysUpTime OBJECT-TYPE
              SYNTAX  TimeTicks
              ACCESS  read-only
              STATUS  mandatory
              DESCRIPTION
                 "The time (in hundredths of a second) since the
                 network management portion of the system was last
                 re-initialized."
              ::= { system 3 }

    END
```

Figure 4: Excerpt from MIB-II

2.4 Lexicographical Ordering of Object Instances

The SNMP framework provides a means by which an NMS can retrieve all object instances from a device without knowing the OIDs of those instances a priori. To accomplish this, a lexicographical ordering is imposed on OIDs, so that the OIDs of all object instances supported by a device can be ordered from least to greatest.

The SNMP protocol includes a command called "GetNextRequest", which instructs the device to return the object instance whose OID is lexicographically next with respect to the OID specified on the request. The NMS can use this command to "walk" the object instances at a device in order: it first issues a GetNextRequest with an OID known to be lexically less than any object instance at the device; each time the device responds with the OID of an object instance, the NMS uses that OID on its subsequent request. This continues until the device responds with an error code indicating no "next" instance exists.

This concludes the SMI background section. Next we turn to MIB compilation.

3.0 Design of MICA

3.1 Design Alternatives

Several important design decisions were made early on in the design of MICA. The first was that the project would be implemented entirely in Java. This was based on the fact that no freely available MIB compilers in Java currently existed, and because Java seemed to be well-suited to the task.

A choice then had to be made regarding the form of the MIB compiler's output. A traditional approach would be to store the compiled output as a set of formatted records, which an SNMP application parses and internalizes. However, writing the compiler in Java offers an intriguing alternative: implementing a MIB database as an abstract data type (ADT). This allows the SNMP application to extract information from MIBs by invoking Java methods, rather than from private data structures built for this purpose. MICA's ADT approach fits well with Java's object-oriented nature.

Another consideration was the form the program itself should take. Java applets offer ease of loading (via a browser) and excellent visual presentation possibilities. Instead, the program is an application. The main reason for this is the default security restrictions on applets, which prevent them from accessing any files on the local system, and from establishing network communications with any host other than the one from which they were downloaded (the default security settings can be overridden on some browsers). These restrictions do not apply to applications.

The next consideration was whether to write the MIB compiler's scanner/parser from scratch or to take advantage of a scanner/parser generator for this purpose. The latter approach was adopted, based on Holmes' recommendations [Holmes95]. The generator chosen is "JavaCC", which is described in the next section.

Finally, a choice had to be made regarding the version of SMI to be supported. Project resource constraints did not permit the development of support for both Versions 1 and 2. MICA supports Version 1. Support of Version 2 is left as future work.

3.2 JavaCC: a Java-based Scanner/Parser Generator

The scanner/parser generator used for this project is "Java Compiler Compiler", or JavaCC [Sun97]. The input to JavaCC consists of a file containing:

- scanning rules, which define the syntax of the target language,
- parsing rules, which define the semantics of the target language, and
- the Java code to be executed when a given parsing rule is used.

The scanning and parsing rules are specified using an extended BNF notation.

For each non-terminal in the grammar of the target language, the user supplies a Java method to be invoked when that non-terminal is encountered.

The parser generated by JavaCC is recursive descent (top-down) in nature, and by default can recognize LL(1) languages. JavaCC warns the user of ambiguities in the grammar, and the user can increase the lookahead value at such points to compensate. As an example, see the OBJECT IDENTIFIER and OBJECT-TYPE constructs in Figure 4. In each case, the construct begins with a name. Using the default lookahead value results in an ambiguity, since the parser is unable to determine the construct to which the name belongs. Increasing the lookahead value to 2 allows the parser to determine whether the token following the name is "OBJECT" or "OBJECT-TYPE", thus resolving the ambiguity.

3.3 Lexical and Parsing Rules for MIBs

Appendix A lists the lexical and parsing rules used by MICA to compile MIBs. MICA makes the information from a MIB's OID value assignments and OBJECT-TYPE definitions available through its API; the other MIB constructs (TRAP-TYPE, SEQUENCE, and textual conventions) are parsed but not further processed.

3.4 MICA's Logic

When MICA is started, it initializes the MIB database with several OID value assignments from SMI. These assignments, which define the structure of the uppermost levels of the ASN.1 object naming tree, are then available for use by MIB modules.

When a compilation is initiated by the user, MICA scans and parses the input MIB file, which consists of one or more MIB modules. Any scanning or parsing errors cause a message to be issued and compilation to cease. The compilation logic is as follows:

1. If end-of-file has been reached in the input stream, exit successfully. Otherwise, processing begins on the next MIB module appearing in the input.
2. The name of the MIB module is saved.
3. If an IMPORTS section is present, it is processed as follows. For each MIB from which names have been imported, a Vector is created to hold the names. These Vectors are saved for use later in resolving the names appearing in the body of the MIB.
4. The OBJECT IDENTIFIER and OBJECT-TYPE declarations in the body (which may appear in any order) are processed as follows. Recall that both of these constructs associate a name with an OID. Most commonly, the OID is specified via the symbolic name of the object's parent in the naming tree, followed by the child number. For example, in Figure 4, "mib-2" is associated with the OID "mgmt.1". MICA saves the OID as specified, and does not attempt to resolve the parent name until later. In the case of OBJECT-TYPE macro, additional information from the clauses of the macro is also saved.
5. When parsing of the body of the MIB module is complete, a check is made to see if a MIB module with the same name has already been compiled. If so, an error message is issued, and control is returned to Step 1.
6. An attempt is then made to resolve any symbolic references encountered in the OIDs from Step 4. If a symbol from the IMPORTS section is referenced, a check is made to verify that the MIB from which the symbol is imported has previously been compiled, and that the symbol was indeed defined in that MIB. Otherwise, the symbol must be resolvable using a definition from the current MIB. Since forward references are possible within a MIB, resolution is performed iteratively. Passes are repeatedly made through the list of unresolved names until either all names are resolved, or no progress is made on a given pass. At the end of this procedure, if any names remain unresolved, an error message is issued listing all unresolved names, and control returns to Step 1.
7. Otherwise, compilation was successful. The MIB's definitions are added to the database of compiled MIBs, a message is issued, and control returns to Step 1.

If any scanning or parsing errors are encountered, the error message issued by MICA cites the line and column number of the input file at which the problem occurred, and lists the token(s) that MICA was expecting at that point. This feature, made possible by facilities provided by JavaCC, allows errors in MIB modules to be found rapidly and corrected.

4.0 MICA User's Guide

4.1 API for MICA

MICA's MIB compiler "engine" is contained in a Java package called "mica". The MICA distribution also includes a driver and a GUI for the compiler, as well as an SNMP application. These are mainly provided as examples of how the MIB compiler can be integrated into a network management application. This section describes the Application Programming Interface (API) for the MIB compiler engine.

Both the driver and the GUI used with the MIB compiler engine import the "mica" package. Further, the driver declares and allocates a static object of type "MibCollection", which acts as the MIB database. SNMP applications reference this object when they need to extract information from the MIBs that have been compiled.

To perform a MIB compilation, the GUI first declares objects of the following types:

- TextArea, to receive compiler messages
- URL, to represent the URL of the MIB source
- InputStream, to represent the data stream associated with the URL
- MibParseSub, to represent an instance of the MIB compiler

The GUI then performs the following actions:

- Allocate an instance of the URL object, passing the name of the desired URL to the constructor
- Invoke the "openStream()" method on the URL object, saving the result in the InputStream object.
- Allocate an instance of the MibParseSub object, passing the InputStream object as a parameter. Note: if the MibParseSub object has been used in a previous compilation, two options are available. Either a new instance of the object may be allocated, or the "ReInit()" method may be invoked on it. In either case, the InputStream object must be passed as a parameter.
- Finally, invoke the "ParseIt()" method on the MibParseSub object, passing the MibCollection and TextArea objects as parameters.

When control returns from the "ParseIt()" method, MIB compilation is complete. The newly compiled MIB objects, if any, have been added to the MibCollection object, and any compilation messages have been written to the TextArea object. The GUI notifies the SNMP application that the MibCollection object has been updated, and an updated list of MIB objects is displayed.

MICA provides several classes as part of its API, which when instantiated provide access to the data contained in MIB modules. The public methods of these classes are described below.

4.1.1 Class MibParseSub

By default, the scanner/parser generated by JavaCC (which is of class "MibParse") writes all compiler messages to the standard output stream object ("System.out"), normally the Java console. To allow these messages to be presented by the GUI instead, class MibParseSub is used instead of MibParse. MibParseSub effectively redirects the messages to the TextArea object that was passed as a parameter to its constructor. The public methods of MibParseSub are shown in Table 1.

Method	Input Object(s)	Comments
MibParseSub	InputStream	Constructor.
ReInit	InputStream	Reinitialize the compiler. This method should be invoked prior to each compilation except the first.
ParseIt	MibCollection, TextArea	Invoke the MIB compiler on the "this" object. The data from the compiled MIB module(s) is stored in the MibCollection object. Parser messages are written to the TextArea object.

Table 1: Public Methods of Class MibParseSub

4.1.2 Class MibCollection

The MibCollection object represents the MIB database. It contains all the information available from the MIBs compiled so far. Its public methods are shown in Table 2. Note that the "getMibOIDItems" method can be used to "walk" the list of all compiled OBJECT-TYPE definitions and OID value assignments.

Method	Input Object(s)	Comments
MibCollection	none	Constructor
hasName	String	Return "true" if the MIB name contained in the String object has been compiled, and "false" otherwise.
getMibNames	none	Return a Vector of Strings, where each string represents the name of a compiled MIB. The strings are stored in alphabetical order.
getMibOIDItems	none	Return an object of type MibOIDItems, which represents all the OID value assignments and OBJECT-TYPE definitions compiled thus far. See the description of class MibOIDItems below for information on how to manipulate this object.
lookupObj	String	Return the object of type MibObject associated with the OID represented by "String". The given OID, which must correspond to a leaf object, may contain instance information (which will be ignored). If no match is found, "null" is returned.

Table 2: Public Methods of Class MibCollection

4.1.3 Class MibOIDItems

The MibOIDItems object represents the information obtained from all OBJECT-TYPE definitions and OID value assignments contained in the MIBs compiled thus far. Its public methods are shown in Table 3.

Method	Input Object(s)	Comments
getItems	none	Return a Vector containing one element per OBJECT-TYPE definition and OID value assignment compiled thus far. The elements in the Vector are sorted by OID, in increasing lexicographic order. Each element is an object of type MibOIDItem; see the description of that class below.

Table 3: Public Methods of Class MibOIDItems

4.1.4 Classes MibOIDItem, OIDVA, and MibObject

Class MibOIDItem is the superclass for all objects that associate an OID with a name. There are two subclasses of MibOIDItem: OIDVA and MibObject. OIDVA represents an OID value assignment; MibObject represents an OBJECT-TYPE definition. Since the two subclasses have most of their methods in common, a MibOIDItem object can be used to represent either one. In fact, since OIDVA has all of its methods in common with MibObject, there is never a need to declare an OIDVA object when using the API; a MibOIDItem object may be declared instead.

The MibOIDItem methods are shown in Table 4. The additional methods available to MibObject objects are shown in Table 5.

Method	Input Object(s)	Comments
getSummary	none	Return a String containing the attributes of the MibOIDItem object (e.g., name, OID, MIB in which object was defined).
getName	none	Return the name of the MibOIDItem object as a String.
getOID	none	Return the OID object associated with this MibOIDItem object. See the description of class OID below for information on how to manipulate this object.
getLevel	none	Return an int indicating the nesting level of this MibOIDItem object, that is, its depth in the ASN.1 object naming tree. The root of the tree is considered level 1.

Table 4: Public Methods of Class MibOIDItem

Method	Input Object(s)	Comments
instanceToString	String	Return a String of the form "name.instance". The input String is a series of dotted octets that identify an instance of this object. For example, if the input String is “1.3.6.1.2.1.1.1.0”, the returned String is “sysDescr.0”.

Table 5: Public Methods of Class MibObject

4.1.5 Class OID

An OID object represents the information associated with an OID. The public methods of this class are shown in Table 6.

Method	Input Object(s)	Comments
toString	none	Return the OID as a String.

Table 6: Public Methods of Class OID

4.2 Modifying MICA's Scanning and Parsing Rules

To modify MICA's scanning and/or parsing rules, follow these steps:

1. Install JavaCC [Sun97].
2. Make the desired changes to file "MibParse.jj" in the "mica" directory.
3. If the changes to the scanning/parsing rules affect the API, update the appropriate Java source files in the "mica" directory.
4. Issue "javacc MibParse.jj". This produces the Java source files for the MIB compiler.
5. Issue "javac *.java" to compile all the source files in the "mica" directory.

5.0 Conclusion

5.1 Lessons Learned

The biggest "surprise" in implementing this project was the ease with which applications can be written in Java. Java provides a rich set of abstractions for handling URLs and socket communications, as well as having "built in" support for many types of data structures. Further, some of the data structures are expandable on the fly. For example, the "Vector" data type, which is similar in function to a one-dimensional array in C, grows automatically if an attempt is made to add an element when the Vector is full. Another powerful feature of Java is automatic garbage collection, which returns program-allocated storage to the free list after the last reference to the storage has been deleted. Features such as these save the programmer a surprising amount of effort.

Using a scanner/parser generator (JavaCC) rather than writing a scanner/parser from scratch was critical to the success of the project. This approach saved a great deal of time and resulted in a more modular design. Another benefit of this approach is that it required the format of MIBs to be specified as a grammar, giving MICA an advantage over MIB compilers that use ad hoc means for scanning and parsing. For instance, some MIB compilers allow only one MIB module to appear in a file, even though SMI allows multiple MIB modules to appear in a file. During MICA development, once the grammar to handle a single MIB module had been developed, it was a simple matter to extend the grammar to support multiple MIB modules, by adding a "+" wildcard to one of the productions.

One disadvantage of JavaCC is that the Java code to be invoked when a particular non-terminal is encountered is coded in-line with the productions themselves, somewhat obscuring the productions. Careful commenting and formatting of the JavaCC input file is needed to make the file comprehensible to others.

Finally, it must be observed that the Simple Network Management Protocol is not very "simple" after all. In spite of attempts by its designers to avoid introducing complexity, there are many subtleties in SNMP. Further, to master SNMP, one must also understand ASN.1, which Tanenbaum describes as "large, complex, and not especially efficient" [Tan96]. Writing an SNMP application or agent also requires an understanding of BER, the encoding rules used when messages are transmitted. Perhaps SNMP should have been called the Sophisticated Network Management Protocol!

5.2 Future Work

There are several possibilities for further development of MICA, such as:

- adding support for SMI Version 2
- increasing the amount of error/consistency checking of MIBs
- enhancing the API to include support for the following MIB constructs: TRAP-TYPE, SEQUENCE, and textual conventions

There are also many opportunities for enhancing the GUI and SNMP application used with MICA. For instance, managed objects could be represented as icons that change color when the object's value changes. The SNMP application could be enhanced to support a "set" capability, to handle more than one object at a time (so that entire tables could be displayed at once), and to support SNMP Versions 2 and 3.

Appendix A: Lexical and Parsing Rules for MIBs

This appendix lists the lexical and parsing rules used by MICA to compile MIBs. These rules were derived primarily from syntax specifications in Perkins [Perkins97]. Three "wildcard" characters are used in these rules: "?", meaning zero or one occurrences; "+", meaning one or more occurrences; and "*", meaning zero or more occurrences. Literals are enclosed in double quotes, and the backslash is used as the escape character. Thus, to specify that a double quote is to be scanned, the following notation is used: `"\""`.

The meanings of the other meta-characters should be obvious. Consult JavaCC documentation [Sun97] for details.

A.1 Lexical Rules

A scanner uses lexical rules to break the input stream into tokens, which are then passed to the parser. The lexical rules used by MICA are shown in Table A-1 below. If one rule is a prefix of another (e.g., "OBJECT" and "OBJECT-TYPE"), the longer of the two must appear first; otherwise the longer rule will never be used.

Token	Rule	Comments
<COMMENT>	"--" (("-")? (~["-", "\n"])+)* ("\n" | "--" | "-\n")	ASN.1 comments begin with "--" and end with the next occurrence of "--" or a newline, whichever comes first. Comments are not passed to the parser.
<CHRSTR>	"\"" (~["\""])* "\""	Character strings begin and end with a double quote.
<WHITESPACE>	" " | "\t" | "\n" | "\r"	As with comments, these whitespace tokens are not passed to the parser.
<ACCESS>	"ACCESS"	Keyword of OBJECT-TYPE macro.
<ACCESSV1>	"not-accessible" | "read-only" | "read-write" | "write-only"	Allowable values of the "ACCESS" clause in SMIv1.
<DESCRIPTION>	"DESCRIPTION"	Keyword of OBJECT-TYPE macro.
<BEGIN>	"BEGIN"	Keyword used to mark the start of a MIB module's definitions.
<DEFINITIONS>	"DEFINITIONS"	Keyword used in MIB header.
<DEFVAL>	"DEFVAL"	Keyword of OBJECT-TYPE macro.
<END>	"END"	Keyword used to mark the end of a MIB module's definitions
<ENTERPRISE>	"ENTERPRISE"	Keyword used in TRAP-TYPE macro.
<FROM>	"FROM"	Keyword used in IMPORTS clause.
<IDENTIFIER>	"IDENTIFIER"	Keyword used as part of "OBJECT IDENTIFIER" construct
<IMPLIED>	"IMPLIED"	Keyword of OBJECT-TYPE macro
<IMPORTS>	"IMPORTS"	Keyword used to mark the start of the "IMPORTS" clause
<INDEX>	"INDEX"	Keyword of OBJECT-TYPE macro.
<MAX>	"MAX"	Keyword used to signify the maximum possible value of a field

Table A-1: MICA's Lexical Rules (Part 1 of 2)

Token	Rule	Comments
<MIN>	"MIN"	Keyword used to signify the minimum possible value of a field
<OBJECTTYPE>	"OBJECT-TYPE"	Keyword used to invoke the OBJECT-TYPE macro
<OBJECT>	"OBJECT"	Keyword used as part of OBJECT IDENTIFIER construct
<OCTET>	"OCTET"	Keyword used as part of OCTET STRING construct
<OF>	"OF"	Keyword used as part of SEQUENCE OF construct
<REFERENCE>	"REFERENCE"	Keyword of OBJECT-TYPE and TRAP-TYPE macros
<SEQUENCE>	"SEQUENCE"	Keyword of OBJECT-TYPE macro
<SIZE>	"SIZE"	Keyword used in subtyping an OCTET STRING
<STATUS>	"STATUS"	Keyword of OBJECT-TYPE macro
<TRAPTYPE>	"TRAP-TYPE"	Keyword used to invoke the TRAP-TYPE macro
<VARIABLES>	"VARIABLES"	Keyword of TRAP-TYPE macro
<WELLKNOWNNAME>	"ccitt" | "iso" | "joint-iso-ccitt"	The names of the child nodes of the (unnamed) root of the ASN.1 object naming tree. All SNMP MIB objects are descendants of the "iso" node.
<ASSIGNOP>	"::="	ASN.1 assignment operator
<VALUE>	<BINSTRING> | <HEXSTRING>	
<OIDVALREF>	<UCNAME> "." <LCNAME>	An OID value reference
<UCNAME>	<UCLETTER> ((<LETTER> | <DIGIT> | "-")* (<LETTER> | <DIGIT>))?>	Identifier that begins with an uppercase letter, followed by zero or more occurrences of letters, digits, and hyphens. The identifier may not end with a hyphen.
<LCNAME>	<LCLETTER> ((<LETTER> | <DIGIT> | "-")* (<LETTER> | <DIGIT>))?>	Identifier that begins with an lowercase letter, followed by zero or more occurrences of letters, digits, and hypens. The identifier may not end with a hyphen.
<NONNEGDEC>	(<DIGIT>)+	Non-negative decimal number
<HEXSTRING>	"\'" (["0"-"9", "A"-"F", "a"-"f"])* "\'" ["H", "h"]	Literal hexadecimal string, for example: '1AC'h. Oddly, no digits need appear between the single quotes; thus, the following is legal: ''h.
<BINSTRING>	"\'" (["0", "1"])* "\'" ["B", "b"]	Literal binary string. See comments for <HEXSTRING", above.
<UCLETTER>	["A"-"Z"]	An uppercase letter
<LCLETTER>	["a"-"z"]	A lowercase letter
<LETTER>	<UCLETTER> | <LCLETTER>	An uppercase or lowercase letter
<DIGIT>	["0"-"9"]	A decimal digit

Table A-1: MICA's Lexical Rules (Part 2 of 2)

A.2 Parsing Rules

Parsing rules, or productions, are used by the parser to determine whether the stream of tokens generated by the scanner is semantically valid. The productions used by MICA appear in Table A-2 below. The start symbol is "<Input>".

Production	Comments
<Input> := <MibMod>+ <EOF>	Parsing of a MIB file begins with this non-terminal. The MIB file contains one or more MIB modules, followed by an end-of-file indicator.
<MibMod> := <UCNAME> <DEFINITIONS> <ASSIGNOP> <BEGIN> <MibBody> <END>	Defines the high-level structure of a MIB module.
<MibBody> := <ImportsClause>? <BodyDefns>*	The IMPORTS clause, if present, must be specified at the start of the MIB body, followed by zero or more definitions in the body of the MIB.
<ImportsClause> := <IMPORTS> <ImportsFrom>+ ";"	The IMPORTS clause contains a list of one or more imported items. It is terminated with a semicolon.
<ImportsFrom> := <ImportsName> ("," <ImportsName>)* <FROM> <UCNAME>	Defines a list of names imported from MIB "<UCNAME>".
<ImportsName> := <OBJECTTYPE> | <TRAPTYPE> | <LCNAME> | <UCNAME>	The name of an imported item
<BodyDefns> := <OTdefinitionV1> | <OVAdefinition> | <TTdefinition> | <TCAdefinitionV1> | <SeqConstruct>	The constructs allowed in the body of a MIB module.
<OVAdefinition> := <LCNAME> <OBJECT> <IDENTIFIER> <ASSIGNOP> <OIDValue>	An OID value assignment
<OIDValue> := "{" <WELLKNOWNNAME> (<NONNEGDEC> | <Enum>)+ "}" | "{" (<OIDVALREF | <LCNAME>) (<NONNEGDEC> | <Enum>)+ "}" | "{" (<NONNEGDEC> | <Enum>)+ "}" | (<OIDVALREF> | <LCNAME>)	The four possible formats of an OID value assignment. Note that in the first three forms, the value assignment is enclosed in curly braces.
<TCAdefinitionV1> := <UCNAME> <ASSIGNOP> <SimpleOrEnum>	A textual convention assignment.
<OTDefinitionV1> := <LCNAME> <OBJECTTYPE> <SYNTAX> <SyntaxV1> <ACCESS> <ACCESSV1> <STATUS> <LCNAME> (<DESCRIPTION> <CHRSTR>)? (<REFERENCE> <CHRSTR>)? (<DEFVAL> "{" <DefValV1> "}")? (<INDEX> "{" <IndexV1> "}")? <ASSIGNOP> <OIDValue>	OBJECT-TYPE macro definition. Note that four of the clauses are optional.

Table A-2: MICA's Parsing Rules (Part 1 of 2)

Production	Comments
<SyntaxV1> := <SimpleOrEnum> | <SEQUENCE> <OF> <UCNAME> | <LCNAME>	Operand of the "SYNTAX" clause of the OBJECT-TYPE macro.
<SimpleOrEnum> := <UCNAME> "{" <Enum> ("," <Enum>)* "}" | ((<OBJECT> <IDENTIFIER> | <OCTET> <STRING> | <UCNAME>) <SubType>)?	A simple or enumerated type.
<Enum> := <LCNAME> "{" <NONNEGDEC> "}"	A specification of an enumeration
<SubType> := "(" ((<VALUE> | <MAX> | <MIN> | ("-")? <NONNEGDEC>) ".." (<VALUE> | <MAX> | <MIN> | ("-")? <NONNEGDEC>) | (<VALUE> | <MAX> | <MIN> | ("-")? <NONNEGDEC>)) ("|" ((<VALUE> | <MAX> | <MIN> | ("-")? <NONNEGDEC>) ".." (<VALUE> | <MAX> | <MIN> | ("-")? <NONNEGDEC>) | (<VALUE> | <MAX> | <MIN> | ("-")? <NONNEGDEC>)))* ")"	A subtype, that is, a specification of the range of values an item is allowed to take. Basically, each component of the subtype can be either a single value, or a range consisting of two values separated by two periods. The components of the subtype must be separated by a vertical bar. Note: the production shown applies to subtyping an INTEGER; a similar production for subtyping an OCTET STRING is not shown.
<DefValV1> := <CHRSTR> | <VALUE> | ("-")? <NONNEGDEC> | <OIDValue>	Operand of the DEFVAL clause of the OBJECT-TYPE macro.
<IndexV1> := <IndexItemV1> ("," <IndexItemV1>)*	Operand of the INDEX clause of the OBJECT-TYPE macro.
<IndexItemV1> := <OIDVALREF> | <LCNAME> | <OCTET> <STRING> | <OBJECT> <IDENTIFIER> | <UCNAME>	Items that can appear in the INDEX clause of the OBJECT-TYPE macro.
<SeqConstruct> := <UCNAME> <ASSIGNOP> <SEQUENCE> "{" SeqMember() ("," SeqMember())* "}"	Format of SEQUENCE construct.
<SeqMember> := <LCNAME> (<OBJECT> <IDENTIFIER> | <OCTET> <STRING> | <UCNAME>) <SubType>?	Items that can appear in a SEQUENCE construct. Note: although RFC 1155 does not permit subtype information to appear on these items, RFC 1213 uses subtyping of SEQUENCE items in some cases, so MICA allows it.
<TTdefinition> := <LCNAME> <TRAPTYPE> <ENTERPRISE> (<OIDVALREF> | <LCNAME>) (<VARIABLES> "{" (<OIDVALREF> | <LCNAME>) ("," (<OIDVALREF> | <LCNAME>))* "}")? (<DESCRIPTION> <CHRSTR>)? (<REFERENCE> <CHRSTR>)? <ASSIGNOP> <NONNEGDEC	Format of the TRAP-TYPE macro.

Table A-2: MICA's Parsing Rules (Part 2 of 2)

Appendix B: Copyright Notices

The copyright notices associated with the code distributed with MICA are listed below.

B.1 West BV's SNMP Stack in Java

The SNMP application distributed with MICA was adapted from code written by West BV Consulting [West96]. Its copyright notice is as follows:

```
Copyright (C) 1995, 1996 by West Consulting BV

Permission to use, copy, modify, and distribute this software
for any purpose and without fee is hereby granted, provided
that the above copyright notices appear in all copies and that
both the copyright notice and this permission notice appear in
supporting documentation.
This software is provided "as is" without express or implied
warranty.
author tim@west.nl (Tim Panton)
author birgit@west.nl (Birgit Arkesteijn)
```

B.2 "Java in a Nutshell" GUI Code

The GUI code distributed with MICA was adapted from examples in the book "Java in a Nutshell" by David Flanagan [Flan96]. Its copyright notice is as follows:

```
This example is from the book _Java in a Nutshell_ by David Flanagan.
Written by David Flanagan.  Copyright (c) 1996 O'Reilly & Associates.
You may study, use, modify, and distribute this example for any purpose.
This example is provided WITHOUT WARRANTY either expressed or implied.
```

B.3 MICA Code

The modifications to West BV's code and to the "Java in a Nutshell" code made as part of the MICA project, as well as all other code written for the project, bear the following copyright notice:

```
COPYRIGHT (c)  1997
THE REGENTS OF THE UNIVERSITY OF MICHIGAN
ALL RIGHTS RESERVED

PERMISSION IS GRANTED TO USE, COPY, CREATE DERIVATIVE
WORKS AND REDISTRIBUTE THIS SOFTWARE AND SUCH
DERIVATIVE WORKS FOR ANY PURPOSE, SO LONG AS THE NAME
OF THE UNIVERSITY OF MICHIGAN IS NOT USED IN ANY
ADVERTISING OR PUBLICITY PERTAINING TO THE USE OR
DISTRIBUTION OF THIS SOFTWARE WITHOUT SPECIFIC, WRITTEN
PRIOR AUTHORIZATION.  IF THE ABOVE COPYRIGHT NOTICE OR
ANY OTHER IDENTIFICATION OF THE UNIVERSITY OF MICHIGAN
IS INCLUDED IN ANY COPY OF ANY PORTION OF THIS SOFTWARE,
THEN THE DISCLAIMER BELOW MUST ALSO BE INCLUDED.

THE SOFTWARE IS PROVIDED AS IS, WITHOUT REPRESENTATION
FROM THE UNIVERSITY OF MICHIGAN AS TO ITS FITNESS FOR ANY
PURPOSE, AND WITHOUT WARRANTY BY THE UNIVERSITY OF MICHIGAN
OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING WITHOUT
LIMITATION THE IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS FOR A PARTICULAR PURPOSE.  THE REGENTS OF THE
UNIVERSITY OF MICHIGAN SHALL NOT BE LIABLE FOR ANY DAMAGES,
INCLUDING SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL
DAMAGES, WITH RESPECT TO ANY CLAIM ARISING OUT OF OR IN
CONNECTION WITH THE USE OF THE SOFTWARE, EVEN IF IT HAS
BEEN OR IS HEREAFTER ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
```

Acknowledgements and References

Acknowledgements

I thank Peter Honeyman, Director of the University of Michigan's Center for Information Technology Integration (CITI), for his guidance and advice on this project.

References

Note: The abbreviation "RFC" stands for "Request for Comments", which are specifications published by the Internet Architecture Board (IAB). RFCs are most conveniently obtained via the World Wide Web. There are many Web sites that contain repositories of RFCs; for instance, see:

http://www.cis.ohio-state.edu/hypertext/information/rfc.html

[Advent] "MIB Browser Applet Version 1.1", Advent SNMP Package. URL: http://www.adventnet.com/snmpapi/mibbrowser/browser.html

[Case90] Case, J., Fedor, M., Schoffstall, M., and Davin. J. "A Simple Network Management Protocol (SNMP)", RFC 1157, May 1990.

[Flan96] Flanagan, D. "Java in a Nutshell", 1st ed. Sebastapol, CA: O'Reilly & Associates, Inc, 1996.

[Holmes95] Holmes, Jim. "Object-Oriented Compiler Construction", 1st ed. Englewood Cliffs, NJ: Prentice-Hall, 1995.

[Lar94] Larmouth, John. "Understanding OSI", 1994. URL: http://www.salford.ac.uk/iti/books/osi/osi.html.

[Nik96] Nikander, P., Wessman, P., and Aukia, P. "Java SNMP Control Applet (JaSCA)", 1996. URL: http://termiitti.akumiitti.fi/nixu/.

[OSI87a] Information Processing Systems - Open Systems Interconnection. "Specification of Abstract Syntax Notation One (ASN.1)", International Organization for Standardization, International Standard 8824, December 1987.

[OSI87b] Information Processing Systems - Open Systems Interconnection. "Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)", International Organization for Standardization, International Standard 8825, December 1987.

[Perkins94] Perkins, D. "SNMP MIB Information Compiler next generation (SMICng)", Bay Networks, 1994. URL: ftp://ftp.net.cmu.edu/pub/smicng/.

[Perkins97] Perkins, D. and McGinnis, E. "Understanding SNMP MIBs", 1st ed. Upper Saddle River, NJ: Prentice-Hall PTR, 1997.

[Rose91a] Rose, M. and McCloghrie, K. "Structure and Identification of Management Information for TCP/IP-based Internets", RFC 1155, May 1990.

[Rose91b] Rose, M. and McCloghrie, K. "Concise MIB Definitions", RFC 1212, March 1991.

[Rose91c] Rose, M. and McCloghrie, K. "Management Information Base for Network Management of TCP/IP-based Internets: MIB-II", RFC 1213, March 1991.

[SNMX] "Simple Network Management Executive (SNMX)", Cyber Professionals, Inc. URL: http://www.cpro.com/cpro/html/cpro_snmx.html.

[Stal96] Stallings, William. "SNMP, SNMPv2, and RMON", 2nd ed. Reading, MA: Addison-Wesley, 1996.

[Sun97] SunTest, "Java Compiler Compiler", Sun Microsystems, Version 0.6 (Final), URL: http://www.suntest.com/. March 28, 1997.

[Tan96] Tanenbaum, Andrew S. "Computer Networks", 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[West96] Panton, T. and Arkesteijn, B. "West's SNMP Stack in Java", Version 1.1, West Consulting BV. URL: http://www.West.NL/archive/java/snmp, 1996.