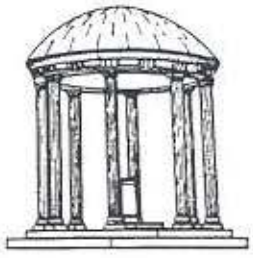
# An Efficient Runtime System for IDL

*TR87-029*

*October 1987*

*Vikram Biyani*



The University of North Carolina at Chapel Hill
Department of Computer Science
Sitterson Hall, 083A
Chapel Hill, NC 27514

# AN EFFICIENT RUNTIME SYSTEM FOR IDL

by

Vikram Biyani

A thesis submitted to the faculty of the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science.

Chapel Hill

October 1987

Approved by

Advisor _____

Reader _____

Reader _____

©1987
Vikram Biyani
ALL RIGHTS RESERVED

VIKRAM BIYANI. An Efficient Runtime System for IDL
(Under the direction of RICHARD SNODGRASS)

The Interface Description Language is a language to specify data structures communicated between processes. This research deals with the design of an efficient runtime system for IDL. This thesis discusses the functions such a runtime system should support, and describes how they are implemented in our system. The runtime system includes support for input and output of IDL instances both in ASCII and in relocatable binary format, support for sets and sequences of objects, an object management system with garbage collection and a memory display facility for debugging purposes. An example IDL specification is presented. Limitations of the system and directions for future work in the area are discussed.

**Acknowledgement**

My special thanks to Rick Snodgrass for his help and encouragement through the course of this research;
to Dean Brock and Bharat Jayaraman for taking the time to serve on the thesis committee and for their comments on this text;
and to Ralph Cook, Ed McKenzie, Anil Nair, and Sundar Varadarajan who helped at various stages of coding, debugging and text preparation.

# Contents

# List of Figures

# 1 Introduction

The *Interface Description Language* (IDL) is a formal way to specify data structures communicated between processes [5, 9, 11]. A process typically reads some input data, manipulates it, and produces some output data which is written out. IDL provides high level utilities for creation and manipulation of these data structures and their input and output.

An IDL specification consists of two parts, the data specification and the algorithm specification. The IDL data specification facility is especially suited for describing graph–structured data. The data structures are specified in a declarative fashion that resembles attributed grammar notation.

The algorithm specification is a program written in one of the IDL target languages, using the high–level utilities provided by IDL. IDL supports abstract data structures such as sets and sequences and provides utilities for their manipulation, so that the user may express the algorithm more naturally in terms of operations on abstract entities without worrying about implementation details. The input and output of data is done through unidirectional *ports*. Each algorithm specification may have several input ports and output ports. Associated with each port is a specification of the data structure that is to be input or output through this port.

The IDL specification is run through the IDL translator, which compiles it into a program in the appropriate target language. After this program has been compiled and linked with the IDL runtime library, it is ready to run.

This thesis describes a runtime system for IDL which is more efficient than the current one, the *IDL Runtime Library Version A*. First, the existing runtime system is briefly touched upon, along with the problems associated with it. Then, other relevant work in this area is reviewed, followed by a description of the design and implementation details of the new system. Finally, recommendations are made for future work in this area. The IDL runtime interface description appears an an appendix. Another appendix presents an example IDL specification, along with files generated from it by the IDL translator.

It is assumed that the reader is fairly familiar with IDL [11]. The next chapter presents and briefly explains a simple example of an IDL specification.

# 2 An Introduction to IDL

This chapter contains a brief introduction to the Interface Description Language with the aid of an example. Most of the material here is excerpted or paraphrased from *A Tutorial Introduction to Using IDL*[11] by Jerry Kickenson. This chapter may be skipped over by the reader familiar with IDL.

To use the IDL system, the user writes a specification in IDL of the cooperating processes that compose the system being developed and of the data structures these processes share.

## 2.1 The Data Declaration

The data structures are specified using named collections called *structures*. A structure is built from units called *nodes* and *classes*.

A node is a named collection of zero or more named values called *attributes* that the user wishes to treat as a unit. Attributes actually hold the data values; nodes are a grouping device.

The declaration for a node consists of the name of a node, followed by the node production operator, `=>`, and a list of zero or more comma separated attribute–type pairs, terminated by a semicolon. All attributes in the same node must have distinct names. The order in which the attributes appear in a declaration is not significant.

The following is an example node declaration:

```
B  =>  X : Boolean,
       R : Rational;
```

This declares `B` to be a node containg two attributes, `X` and `R`, of types `Boolean` and `Rational` respectively. The standard attribute types supported by IDL are `Integer`, `Boolean`, `Rational`, and `String`. A node may have another node as its attribute. IDL also supports structured attribute types like `Sets` and `Sequences` of objects of other types.

In addition to the above mentioned attribute types, IDL allows the user to augment the standard IDL attribute types by defining and using what are known as `Private Types`. The user is responsible for providing

data declarations and procedure definitions for his private types in the Target Language. These declarations are made known to the IDL translator through the IDL specification and are linked with the user program at compile time.

A class is a name for a variable that may hold a reference to one of a set of nodes or other classes. A Class is analogous to a C union. The nodes and classes that a class can refer to are called its members.

A class declaration consists of the name of the class, followed by the class production operator, ::=, and a list of one or more node or other class names separated by alternation signs, |, and terminated by a semicolon. The ordering of node and class names on the right hand side of a class declaration is not not significant.

All node and class declarations occur within structure declarations. A declaration of a structure starts with the IDL keyword `Structure` followed by the structure's name, then by the designation of a node or class as its root introduced by the IDL keyword `Root` and then by a list of the one or more node and class declarations that comprise the structure that begins with the IDL keyword `Is` and ends with the IDL keyword `End`.

Below is an example structure declaration.

```
Structure Structin Root Anode Is
  Anode  =>  List: Seq Of B,
             Name: CharSet;

  For CharSet Use Package CharSet;
  For CharSet Use Type CharSet;
  For CharSet Use External Representation String;

  B  =>  X: Boolean,
         R: Rational;
End
```

A structure called `Structin` is declared, consisting of two nodes, `Anode` and `B`. The root of this structure is the node `Anode`. One of the attributes of `Anode`, called `Name`, is of a private type called `CharSet`. By saying,

```
For CharSet Use Package CharSet;
```

the user informs the IDL translator that the declarations for this private type are to be found in the file `CharSet.h` and its procedure definitions are in the file `CharSet.c`. By saying,

```
For CharSet Use type CharSet;
```

the user informs the IDL translator that the IDL private type `CharSet` is implemented as the `C` type `CharSet`. The user also informs the IDL translator that the external representation for `Charset` is the IDL type `String`.

IDL allows the user to declare a new data structure in terms of one or more previously declared structures. This is known as *derivation* of one structure from other structures. Derivation allows the user to copy the node and class declarations from several structures and then add or delete attributes to node types, members to class types, whole node types, or whole class types.

For instance,

```
Structure Structout Root Anode From Structin Is
  Without B => X;

  Anode => Number: Integer;
End
```

The above declares a structure named Structout which is derived from the previously declared structure Structin. The first node production is preceded by the IDL keyword `Without`. It indicates the deletion of the attribute `X` from the node type `B`. The second node production introduces a new attribute `Number` into the nodetype `Anode`.

## 2.2 The Process Declaration

A process is the IDL model for a computation. An instance of a process reads and writes instances od IDL–specified data structures to and from external storage through a collection of *ports*.

A port is an association between an IDL–specified data structure and a name for the IDL–supplied implementation of the routines for reading and

writing that structure. A Pre port is for input and a Post port is for output. The routines that do the input and output of IDL data structures have to translate between the internal (in main memory) representation and the external ( outside main memory) representation of thee data structures. IDL supports two different formats for external representation, *ASCII External Representation Language* (ASCII ERL), and *Relocatable Binary*.

A Mark port is for neither input nor output. It associates with a structure a routine that sets bits indicating whether an object is reachable from a given structure instance.

Below is an example process declaration:

```
Process InOut Is
  Target Language C;
  Target Runtime Version B;

  Pre P: Structin;
  For P Use Ascii External Representation;

  Mark M: Structin;

  Post Q: Structout;
  For Q Use Binary;

  Pre R: Structout;
  For R Use Binary;

  Post S: StructIn;
  For S Use Ascii External Representation;
End
```

The above declares a process named InOut with five ports, namely P, M, Q, R and S. Associated with each port is a previously declared structure. For each input or output port, the external representation is specified as Binary or Ascii External Representation. There are also statements that specify the target language and the runtime system used for this process.

## 2.3 ASCII External Representation

Of the two external representation formats used by IDL, the ASCII ERL is the one that is human readable.

The syntax of the ASCII ERL is free form. A structure instance is represented in terms of the objects that comprise it. The representation of an object can be nested within the representation of the node that references it or placed at the highest level to form a flat list of objects. A representation that uses any intermediate level of nesting is also acceptable.

The following is an example ASCII ERL representation of an instance of the structure `StructIn` described earlier:

```
Anode [List < FirstB^ SecondB^ ThirdB^ SecondB^> ;
       Name "astz"
      ]
FirstB: B[X TRUE; R  -11/30]
SecondB:B[R 11.3]
ThirdB: B[X FALSE]
#
```

The above is a flat representation of the four nodes that comprise the instance. The first node in the file is the root node. A node is represented as an optionally labeled node name followed by an enumeration of its attribute–value pairs, separated by semicolons and bounded by square brackets. The ASCII ERL representation of an instance is terminated by a `#`.

## 2.4 A Complete IDL Specification

Below an IDL specification is presented. It uses a private type named `CharSet`. The algorithm specification and the user supplied private type files for this example can be found in *Appendix B*. The files generated by the IDL translator for this example can also be found in *Appendix B*.

```
Structure Structin Root Anode Is
  Anode  =>  List: Seq Of B,
             Name: CharSet;
```

```
  For CharSet Use Package CharSet;
  For CharSet Use Type CharSet;
  For CharSet Use External Representation String;

  B  =>  X: Boolean,
         R: Rational;
End

Structure Structout Root Anode From Structin Is
  Without B => X;

  Anode => Number: Integer;
End

Process InOut Is
  Target Language C;
  Target Runtime Version B;

  Pre P: Structin;
  For P Use Ascii External Representation;

  Mark M: Structin;

  Post Q: Structout;
  For Q Use Binary;

  Pre R: Structout;
  For R Use Binary;

  Post S: StructIn;
  For S Use Ascii External Representation;
End
```

# 3 The Problem

In this chapter, we describe the functions that need to be supported by the IDL runtime library. Then we describe the pre–existing runtime system, with some of the problems associated with it. Finally we describe our approach to supporting these functions.

IDL data structures, termed *instances*, are directed, possibly cyclic, graphs. The fundamental building block of a structure instance is a self–identifying, or *tagged*, chunk of memory termed a *data object*. It has a 4–byte header word that contains its size, type, and similar information. A data object may also be referred to simply as an *object*.

There are two types of data objects, namely, *nodes* and *system objects*. A node is a user–defined data object implemented as a constant amount of storage depending upon its *nodetype*, with slots for its *attributes*. An IDL specification of a node is much like a *C* `struct` declaration and the attributes of a node are a lot like the fields of a *C* `struct`. A *nodetype* is analogous to a *C* type identifier.

A node may have an attribute either by *value* or by *reference*. The distinction between the two is that of sharability. A *value attribute* is not sharable among nodes. It is physically located within the node to which it belongs. A *reference attribute*, as the name suggests, is a pointer to a data object in main memory. More than one node can share a reference attribute by possessing a reference to it. Value attributes are those whose sizes can be determined at compile time, typically scalar data types such as booleans, integers and rational numbers. Reference attributes are either other nodes or things like character strings and sets and sequences of other objects, whose size cannot be determined at compile time.

System objects are system–defined and are invisible to the user. They are used to implement sets, sequences and strings. Some system objects are used by the runtime system to implement data structures used internally. They do not form part of any structure instance.

A *structure instance* is a directed graph with a distinguished vertex termed its *root*, from which all of its other vertices are reachable. The vertices of this graph are nodes and system objects. Its arcs are pointers linking nodes to their reference attributes, and pointers linking objects to form sets and sequences.

The IDL runtime library supports four classes of functions:

1. Operations on structured data such as sequences and sets;
2. Creation of new instances of IDL data structures, and their initialization, if necessary;
3. Reclamation of storage from instances that have outlived their use;
4. Movement of IDL structure instances between main memory and disk.

The current IDL runtime system uses the ASCII External Representation Language (ASCII ERL) to represent IDL structure instances on disk. The ASCII ERL representation of an IDL structure instance is a linear enumeration of its sub–graphs. Each node, set or sequence, may have a (unique) label through which it can be referred to.

Within a node, each attribute is listed as a *(name, value)* pair. A reference attribute can be listed alternatively as a *(name, label)* pair. The drawbacks of ASCII ERL are that it is bulky, and that a considerable translation effort is required to convert between it and the IDL internal representation. Its advantages are that it is machine and language independent and that it is in ASCII, which make it, at least in theory, human readable. In practice, though, an IDL structure may be complicated, and if it is large enough, any linear representation of it, as in the ASCII ERL, may be hard to read.

A new object is created by allocating storage for it and inserting into it the information that makes it self–identifying. Each time space needs to be allocated, a UNIX system call is made. The user may specify an initialization procedure for objects of a particular type. If no initialization procedure is specified, the newly created object undergoes a default initialization before being returned to the user.

The IDL runtime library provides a facility to create sets and sequences of objects, and associated operations like insertion and deletion of elements, sorting of sequences, and membership tests. There can be several ways to represent sets and sequences. A sequence can be represented as a linked list, or as an array of pointers to its members. A set of abstract entities can be represented as a bit map. This representation leads to efficient set operations, but is feasible only if the size of the average set is not too small

compared to the size of the universal set in the domain. A set of objects can also be implemented as a special sequence, with no member occurring more than once.

The current IDL runtime system only supports explicit deletion of objects. The user has to keep track of objects that are not useful any more and reclaim the space used by them.

Some of the problems with the existing system are as follows. The ASCII ERL is extremely inefficient both in time and in space. Object management routines are slowed down because they use UNIX system calls for allocation and deletion of objects. The requirement of explicitly deletion to reclaim space puts unnecessary burden on the user. In addition, it may often be impossible to determine at programming time whether an object will become useless and its space should be reclaimed. If all references to a data object are destroyed, and it becomes inaccessible, it becomes *garbage* and the space allocated to it can never be recovered. Another problem is that of an object being deleted before all references to it are destroyed. Such a reference is called a *dangling reference*. If a program attempts to modify through a dangling reference, contents of a totally unrelated data object, or even the yet unallocated memory are likely to be modified, leading to errors in execution that may be very hard to detect.

We see that though all functions listed above are supported in some form by the existing IDL runtime system, there is need for improvement in efficiency, functionality, and usability.

For reasons given below, the representation of a data object on main memory must be different from its representation when it is transferred to disk. We call the main memory representation of data objects their *internal representation* and the representation on disk their *external representation*. Within an IDL process, there may be more than one structure declaration, each associated with one or more ports. IDL provides the user with a data derivation facility which makes it possible to define one data structure in terms of other previously defined data structures. Derivation allows a user to copy a previous node definition from one structure to another, retaining its nodetype, and then to edit it by adding new attributes and deleting old attributes. As a result of such derivations, a process may have the same nodetype defined with different attributes within different structures.

Within a process, all node objects with the same *nodetype* have the

same representation in main memory. This representation is defined by the *process invariant structure* which is derived by taking the union of all structures defined in the process. Though this representation may be wasteful in terms to space, it simplifies the routines and tables that have to be maintained by the runtime system to aid in the creation and manipulation of these data structures. Before a structure instance associated with an output port is to be written out, the invariant representation for each node object in that instance must be shrunk to just the node declaration in that particular structure. Before writing to disk, each reference to an object in main memory (generally a pointer) must be translated to a reference to the same object on disk. Similarly, when an instance of a structure associated with an input port is to be read in, each node object read in is mapped to its invariant representation in main memory. Each relocatable reference must be translated to an absolute memory reference.

An alternative to the requirement of having to free space by explicitly deleting instances is to provide for garbage collection. As soon as an allocation request fails owing to the lack of available space, the garbage collector is activated.

In the runtime system described in this thesis, data objects are written to disk in a *relocatable binary* format. A relocatable binary representation is a dump of all useful memory with all pointers made relocatable. Unlike the ASCII ERL, this representation is both machine and language dependent. Also unlike the ASCII ERL, this representation is compact, the translation between this and the IDL internal representation is quick, and the code required for this translation is small.

This runtime system also provides an improved facility for input and output in ASCII ERL. Both ASCII and relocatable binary I/O are table driven. The tables are generated from the user's IDL data specification by the IDL translator. Even though ASCII ERL continues to be supported for input and output, the preferred format for input and output is relocatable binary.

In contrast to the existing runtime system, which uses UNIX system calls for space allocation and de–allocation, this system does its own memory management. It maintains a pool of free blocks of memory from which allocation is done. Explicit de–allocation of IDL instances is not required. Whenever the memory management system runs out of space, a garbage

collector is activated. This garbage collector de–allocates all instances that are not reachable through currently active user variables. This prevents the creation of dangling references. Adjacent free blocks of memory are coalesced into one block. This relieves the user of the burden of space management. The overhead of explicit deletion of instances is eliminated, since the garbage collector goes into operation only when the process has run out of space.

# 4 Previous Work

This section reviews work by other researchers in fields relevant to the the system under discussion.

## 4.1 Input and Output of IDL instances

The spectrum of IDL external representations is described by Newcomer [6]:

1. ASCII
2. Tokenized ASCII
3. Absolute Binary
4. Relocatable Binary

We have already described the ASCII ERL and the problems associated with it.

The use of tokenized ASCII eliminates the need for lexical analysis from the reading phase. This makes it faster and smaller than ASCII ERL, though it is not human readable.

The *absolute binary* representation is essentially a memory dump. A structure transferred from one process to another, using this representation, must occupy the same memory locations in the virtual space of the receiving process that it occupied in the source process. This requirement is much too restrictive for this representation to be generally useful.

The *relocatable binary* representation is an improvement on the absolute binary representation in that all references are relocatable. It is not human readable, but it is compact. Newcomer reports that the input and output of IDL structure instances is improved in speed by as much as two orders of magnitude when the ASCII ERL is replaced by the relocatable binary format.

As noted in the previous chapter, the existing IDL runtime system supports only ASCII ERL format for input and output of IDL structure instances. The IDL translator analyzes the data specification and generates routines for reading and writing of IDL structure instances. It creates a

different routine to read or write node objects of each individual nodetype, and these routines are called during the input or output of an IDL structure instance. The amount of code required for input and output is proportional to the complexity of the IDL specification for that structure.

In contrast, the runtime system described here uses a general purpose driver for reading and writing IDL structure instances. The IDL translator generates a tables that contain the information specific to the IDL structure definitions in a particular process. The general purpose drivers use information in these tables to read in and write out particular types of nodes and their attributes.

When an IDL structure is being traversed so that it can be written out, all the pointers in each node have to be located and followed. Newcomer uses a bit–map called a *pointer dictionary* for each type of node. Each bit in the bit–map indicates whether the corresponding word in the node is a pointer.

In addition to information about location of pointers within nodes, our implementation also needs information about mapping of attributes in each node in the process invariant to the corresponding attributes in the corresponding nodes in the port structure. The latter is best represented in tables. Rather than keeping a table and a pointer dictionary, we find it more efficient to incorporate the information about pointers in the tables and do away with the pointer dictionary.

## 4.2 Recovery Of Instances Not In Use

Data elements that are no longer in use should be recovered, and space allocated to them should be freed so that it can be used elsewhere. There are three principal approaches to recovery [1, 7]:

1. Explicit Return,
2. Reference Counting, and
3. Mark–Scan Garbage Collection.

The explicit return approach is the simplest to implement. The runtime system provides a procedure which, when called with a data object as its

argument, destroys the object and frees the space associated with it. This is the approach taken by the existing IDL runtime system. The problems associated with it have been described in the previous chapter.

Reference counting is an approach to space recovery that prevents the creation of dangling references [2]. Associated with each data object is a number called its *reference count*. When an object is created, its reference count is set to *1*. Each time a new reference to an object is created, its reference count is incremented. Conversely, each time a reference is destroyed, its reference count is decremented. When the reference count of an object becomes zero, it indicates that no references to that object exist any more. At this time, the object is deleted and the space occupied by it is reclaimed.

There are several problems with this approach. This technique may fail to prevent the creation of garbage if there are self referential data structures. The requirement that each object have a reference count may lead to a severe space overhead if the average size of objects is small. The maintenance of reference counts causes a substantial overhead in execution time as well. For every assignment to a pointer variable, one reference is destroyed and another created, i.e., one reference count is decremented and another incremented.

Mark–scan garbage collection is an approach that makes no explicit deletions, thus preventing the creation of dangling pointers. It lets garbage be created until all space is exhausted, at which time, a special procedure goes into operation that identifies all garbage and returns the storage to the free pool. This approach requires each data object to have a 'mark' bit, which is initially off.

There are two stages in this procedure. In the first stage, all objects that are accessible by the process running are marked. In the second stage, the entire memory space is scanned and all objects that are not marked are recovered. All objects that were marked in the first phase have their marks turned off.

There are other more specialized and more sophisticated approaches to garbage collection. For instance, Deutsch and Bobrow describe an incremental garbage collector whose performance is based on the observation that most allocated storage is either referenced only by one unchanging pointer throughout its lifetime or is used for *temporary results*, i.e., is abandoned quickly after creation [3]. While this observation may be true for

execution of LISP programs, it probably does not hold for IDL instances in memory, therefore the efficiency of this garbage collection strategy in the context of IDL processes is doubtful.

Wegbreit describes a compactifying garbage collector whose main virtue is that it works even when pointers point in the middle of objects, and the algorithm goes to great lengths to ensure correctness in such situations [12]. Each pointer in an IDL instance refers to the starting location of an object, never to the middle of it, therefore the problem that this garbage collector is attempting to solve does not exist in this context.

# 5 Overview of the Runtime System Design

Chapter 2 briefly described the functions that should be supported by an IDL runtime system. This chapter contains a high-level description of the approach taken in this runtime system to support each of these functions. The next chapter provides the next level of implementation detail.

## 5.1 IDL Data Objects

There are three types of IDL data objects:

1. Node,
2. System Object, and
3. Free Page.

Each data object is a contiguous piece of memory. The first word of the data object, also known as its *header*, identifies the object. The interpretation of the rest of the data object depends on its header. The node is the only user defined object. Other objects are system-defined.

The header of a node contains a field called its **nodetype**. All node objects with the same **nodetype** have the same size and the same attributes. The IDL translator generates a table called the *process invariant table* which has entries describing the memory layout of instances of each nodetype. For each IDL port, it also generates a table called *port table*, describing the nodes in the corresponding port structure.

System objects are further classified into eight types. Below, the three that may appear in IDL structure instances are listed.

1. String,
2. List Cell, and
3. List Header.

The other five types of system objects, used internally by the runtime system, are discussed in the next chapter.

A *string* is a variable sized object that stores a character string.

*List cells* are used to construct linked lists, representing sets and sequences of other objects.

A sequence (or a set) can be shared between many nodes, through a pointer to the first list cell in the sequence. As changes are made to a sequence during the lifetime of a process, the first cell in the sequence must remain the same, so that dangling pointers are not created. This requirement cannot always be met. For instance, a null sequence has no first list cell, therefore, when a shared non-null sequence is made null, it will always create dangling references. A way to get around this problem is to always have a dummy list cell as the first element of a sequence. This is a special type of system object known as a *list header*. A list header is merely a place–holder for a sequence. It provides a stable address for a sequence whose composition might keep changing.

The IDL object management system maintains a pool of free memory from which it periodically allocates storage for the creation of nodes and system objects. This pool consists of variable sized chunks of memory known as free pages.

## 5.2 Support for Sets and Sequences

Sets and sequences are two structured data types supported by IDL. A set or a sequence is a collection of instances of one of the predefined types, namely, `Boolean`, `Integer`, `String`, and `Rational`, a nodetype, or a user defined type known as a *private type*. Figure 1 lists operations possible on sets and sequences.

The representation of sets and sequences uses list cells. A sequence of instances of a scalar type like Boolean, Integer, Rational, or a private type, is represented by linked list of list cell objects, the data field in each list cell containing a member of the sequence. A sequence of node objects or of string objects, is also represented by a similar linked list, the data field in each list cell containing a pointer to a member of the sequence.

A set is implemented as a special case of a sequence, with no member occurring more than once.

| operation | *sets* | *sequences* |
|---|---|---|
| 1. Addition of an element | √ | To the front |
| | | To the rear |
| | | In order |
| 2. Making a copy | √ | √ |
| 3. Test for emptiness | √ | √ |
| 4. Iteration over elements | √ | √ |
| 5. Initialization to empty | √ | √ |
| 6. Membership Test | √ | √ |
| 7. Return the size | √ | √ |
| 8. Removal of an element | √ | Of the first occurrence of a given element |
| | | Of the first member |
| | | Of the last member |
| | | Of the $i^{th}$ member |
| 9. Retrieval | × | Of the first member |
| | | Of the last member |
| | | Of the $i^{th}$ member |
| | | Of the tail |
| 10. Sorting | × | √ |

Figure 1: Operations Supported on Sets and Sequences

## 5.3 Creation of IDL Data Objects

As mentioned earlier, this runtime system does its own memory management. Throughout execution, a pool of free memory is maintained by the memory management system. This pool consists of data objects known as *free pages*. A free page is identified by its header. The data contained in a free page includes its size and pointers linking it into the pool of free pages.

At the start of execution, the free pool consists of a rather large free page which constitutes the entire IDL data memory available to the user process. As execution proceeds, this free page keeps getting smaller with each allocation. At some point there may be an allocation request that

cannot be met by this free page. This failure causes the garbage collection procedure to go into operation. When garbage collection is over, the free pool is likely to contain several free pages, one of which will likely be of adequate size. If this is not the case, the allocation request fails and the user is notified.

### 5.3.1 Creation of an IDL Node Object

The runtime system receives a request for the creation of a new node instance with a specified nodetype. The size of memory to be allocated is obtained from the corresponding entry in the *process invariant table*. A chunk of memory of this specified size is carved out of one of the free pages in the free pool. The first word of this piece of memory is set up as the node header, with information identifying it as an instance of a particular `nodetype`.

The life of a node object may extend beyond the lifetime of the process by which it is created. A process may create a node and write it out to disk as a part of a structure instance. At some time in the future, this node object may be read in by another process, possibly running on a different machine. It is desirable to associate with each node object a process identifier unique over space and time, which identifies the process by which this node was originally created. This process ID is derived from the time of process creation, the address of the machine on which the process is running, and its UNIX `process_id`. The runtime system maintains a *process ID table* and each node object in main memory has a `processID` field in its header that contains an offset into this table, identifying the process by which it was created.

When a node object is created, its `processID` field is given the appropriate value.

All attributes of this node are initialized using a user provided initialization routine for instances of that particular nodetype. If there is no initialization routine, the attributes are given system defined default values. Finally a pointer to the newly created node instance is returned.

### 5.3.2 Creation of an IDL String Object

A request for the creation of a new string object is accompanied by a pointer to a C string.

The runtime system maintains a hash table to store addresses of all string objects in use. When it receives a request for the creation of a new string object, it tries to find an already existing string object that matches the given C string. If such an object exists, a pointer to it is returned to the caller. Otherwise, a piece of memory is allocated from the free pool, its header is set up to reflect that it is an IDL string object of a particular size, and the characters in the C String are copied to it. A pointer to the newly created string object is returned to the caller, and also stored in the hash table. This procedure ensures that a process never has more than one string object with the same contents. This results in the test for equality of strings being reduced to the much simpler test for equality of pointers. It also requires that once a string object has been created, its contents cannot be modified. Of course, a reference to a string object can be replaced by a reference to another string object, containing different characters.

### 5.3.3 Creation of Other Data Objects

Creation of any other object is much like the creation of a node object. It involves allocation, setting up the header, initializing the data, and returning to the caller a pointer to the newly created object.

## 5.4 Reclamation of Storage

As soon as an allocation request is made which cannot be met because of a lack of free space in the IDL data memory, a special procedure known as *garbage collection* goes into operation.

The *marking* phase of the garbage collector scans each location that could possibly be a pointer in the runtime stack and the global variable area of the process in execution [10]. It treats each such word as a potential pointer to an IDL data object. If such a word is found to contain the address of the header of an IDL data object, that object is classified as *active*. Occasionally, a word which is not really a pointer is treated as one,

and as a result, a piece of garbage may be marked *active*. This does not cause any error; it merely results in some garbage not being collected.

All IDL data objects reachable from an active object are also classified as active. During the *marking* phase, the garbage collection flag in each active object is turned on.

After the marking phase is over, the *scan* phase begins. During this phase, a linear scan of the IDL data memory is made. Each data object encountered with its garbage collection flag off is reclaimed and added to the pool of free pages. Physically adjacent free pages are coalesced into one page. The garbage collection flags of active objects are turned off during this phase. At the end of this phase, all active objects are as before, all garbage has been collected and returned to the free pool, and garbage collection flags are off on all objects.

In contrast with the method of reference counts described in chapter 3, this scheme can garbage collect circular structures.

## 5.5 Reading and Writing of IDL Instances

This section first describes the relocatable binary writer and reader algorithms. Then it describes the minor changes and enhancements needed to support ASCII reading and writing.

### 5.5.1 Binary Writer

The IDL writer is called to write IDL structure instances existing in main memory to secondary storage. Its tasks include the following:

1. Identify all data objects reachable from the root of the structure instance to be written to disk. All these are parts of the structure to be written out.

2. Locate all pointers within each of these data objects and convert them to relocatable references.

3. Ensure that data objects shared within the structure instance are written out only once.

4. Ensure that each node object is written out in conformity with the structure specification associated with that particular output port. This means that in particular, the invariant representation of a node in main memory should be translated to its representation in the port structure before being written out.

5. Ensure that all private type objects and attributes are written out in their external representations.

The IDL writer requires the following data structures to be supplied. Some of them are previously initialized global variables, others are passed to it as procedure parameters.

1. A pointer to the root of the structure instance to be written out. This is supplied as a parameter by the user.

2. An open file to write to, also supplied as a parameter.

3. The *process invariant table*, a table describing the *invariant representation* of each type of node in main memory. This table is generated by the IDL translator and is available to the runtime system.

4. The *port table*, a table describing the port structure. There is one such table for each IDL port, generated by the IDL translator and available to the runtime system.

5. The *private type table*, a table containing specifications for private types, pointers to initialization routines, and to routines for conversion between their external and internal representations. This table is also generated by the IDL translator and is available to the runtime system.

Starting with the root, with information from the port table and the process invariant table, all pointers in the given structure instance are followed. These include the pointers to reference attributes in nodes and the set and sequence links. Each data object encountered is *marked*. The marking operation consists of associating with each data object a unique positive even integer known as its *relocation index* and setting up a flag in the data object that says that it has been visited during the marking phase of the

writer. The relocation index of the object is stored in place of the object header. The header of this object and a pointer to it are saved in an entry of a table called the *relocation table.*

If there are $n$ data objects in the structure to be written out, their relocation indices range from *2* to *2n*. Index 2 is assigned always to the *root* data object. Index 0 represents a null pointer. Odd integers are used to represent nodes without attributes and are therefore not usable as relocation indices. Since a data object is uniquely identified by its relocation index, each absolute pointer to that object can be made relocatable by replacing it with the relocation index of that object.

Once a data object has been marked, it is ready to be written to disk. Below, the procedure of writing out a node is described. List cells and string objects can be written out in a similar manner.

First the header of the node is written out. Then each of its attributes have to be written out in some form. The port table contains entries indexed by `nodetype` that describe an instance of that nodetype and its attributes.

As noted earlier, a node may have two kinds of attributes. A reference attribute is represented within a node as a pointer to an IDL data object. Before this attribute can be written out, the data object it points to must be marked. The relocation index of this data object is written out in place of the pointer.

A value attribute can be either of a scalar or of a private type. The external representation of a scalar is the same as its internal representation. It can be written out as is. An attribute of a private type has to be translated to its external representation. A pointer to the conversion routine can be obtained from the *private type table.* The external representation of a private type may turn out to be yet another private type, in which case another conversion step is required. The chain of conversions stops as soon as it gets to an external representation that is not a private type. Now it can be dealt with like a reference attribute or a scalar value attribute.

### 5.5.2 Binary Reader

The actions of a the binary reader are an exact reverse of those of the binary writer. The binary reader is called by a process to read in from disk to main memory, an IDL structure instance which was written out by

a binary writer called possibly by some other process. The tasks of the reader include the following:

1. Allocate space in main memory and read in the data objects comprising the IDL structure being read.

2. Translate the external representation of each node object, as specified by the structure declaration associated with the input port, into the invariant representation for that node.

3. Convert all relocatable references to absolute memory references.

4. Convert external representations of private type objects and attributes to their internal representations.

The data structures required by a binary reader are similar to those needed by the binary writer. It needs an open file to read from, the *process invariant table*, the *port table* for the particular input port, and the *private type table*.

The file contains a list of data objects that make up the structure being read in. For each data object, first its header is read in. Since the header of a data object identifies its size and type, space can be allocated for it in the data memory. Once space has been allocated, the rest of the data object can be read in. The size of a node in the input structure may be different from the size of the corresponding node in the process invariant structure. In addition, the number of attributes and their physical locations within the node may vary between the two. The reader must read in nodes of the first kind and create in memory matching nodes of the second kind. In order to do this, it needs the information contained in the *process invariant table* and the *port table*. All string objects read in are entered in the *string hash table*. All private type attributes and objects are converted from their external to their internal representations with the help of the private type table. After all the data objects have been read in, the relocatable pointers are translated to absolute memory addresses and the reading operation is over.

### 5.5.3 ASCII Writer

The ASCII writer outputs an IDL structure instance in human readable form. Each structure instance is written out in fully nested format, with a label on each object so it can be referred to from other objects in case it is shared. The ASCII writer needs an additional data structure to carry out its task. The *Nodetype Table* maps from integers to the names of nodes they represent, and also from indices of attribute within nodes to the names of the attributes they represent. This table is generated by the IDL translator for ports that have ASCII writers.

### 5.5.4 ASCII Reader

The ASCII Reader parses the structure instance represented in ASCII External Representation Language, and reads it into main memory. It needs an additional data structure, the *NodeName Table* generated by the IDL translator. This table maps node names into integer nodetypes and from attribute names to attribute indices.

## 5.6 Marker Ports

In addition to the ports that are used for input and output of structure instances, there are ports that are used to do a reachability test on given structure instance. These ports are known as *marker ports*; in spite of their name, they have nothing to do with input or output. However, like in the case of an input or output port, a structure specification is associated with each marker port, describing the structure instances it operates on.

An IDL structure declaration is associated with each marker port. When given a pointer to a structure instance, the marker port traverses the entire instance according to its structure declaration and *marks* all data objects that comprise this instance. The marking algorithm is very similar to the writing algorithm except that nothing gets written. There are two bits in the header of each data object, namely `touched` and `shared`, that are used by marker ports. Initially, both these bits are reset. After marking is over, each object reachable from the root has either its `touched` or its `shared` bit, but not both, set to *1*. The `shared` bit indicates that the object is

reachable from the root through more than one path. The touched bit indicates that it is reachable through only one path.

The marking algorithm, staring with the root of the instance, is as follows:

1. If both bits are reset, set the touched bit, and mark all objects reachable from this object, following the pointers and private type values.
2. If the touched bit is set, turn it off and set the shared bit.
3. If the shared bit is on, do nothing.
4. To mark a private type value, call the user provided marking routine for this private type with the marker port routine as its argument.

## 5.7 Private Types

The standard set of attribute types provided by IDL is limited to the *scalar* types, `Boolean`, `Integer`, `Rational` and `String`, the *structured* types, `Set Of` and `Seq Of`, and the user defined *Node* type. This set can be augmented by the user through the use of the *private type* facility.

To use this facility, the user must first declare a named private type. Then he must declare the internal and external representations for this private type. The data definition for the internal representation of this private type is specified in the target language by the user. The size of the internal representation can be determined from this data definition. Its external representation must be an IDL standard type or a previously defined IDL node or class.

As an example, the IDL specification in *Chapter 2* uses a private type named `CharSet`. Its external type is `String`.

The user must provide certain procedures to the runtime system for manipulation of this private type. The names of these procedures supplied by the user must follow the scheme described below.

⟨*TypeName*⟩`To`⟨*External*⟩ should be the name of the routine that translates from the internal representation of a private type value to its external representation. The corresponding routine in the example is `CharSetToString`.

Similarly, the routine that translates from the external representation of a private type value to its internal representation should be named ⟨*External*⟩`To`⟨*TypeName*⟩. The corresponding routine in the example is `StringToCharSet`.

Either routine takes two parameters, both of type `* char`. The first parameter, `source`, points to the the value to be translated. The second points to a buffer where the translated value is to be returned.

The user is responsible for providing an *initialization* routine for each private type. This routine should be named ⟨*TypeName*⟩`Initialize`. The corresponding routine in the example is `CharSetInitialize`. This routine takes one parameter, of type, `* char`, which points at the starting location of the private type value to be initialized.

A *marking* routine must also be provided with each private type. This routine should be named ⟨*TypeName*⟩`Mark`. The corresponding routine in the example is `CharSetMark`. This routine takes three parameters, the first of which is of type `* char`, a pointer to the private type value to be marked, the second is a pointer to a routine, and the third is a pointer to the port table for that marker port. the task of the marking routine is to call the passed routine for each IDL data object that is reachable from the private type value which is being marked. The passed routine has two arguments, the first is of type `* int` and is the address of the data object to be marked, and the second is the above mentioned pointer to the port table. This routine returns `void`. If it is a non-null pointer, the This marking routine is used during the garbage collection and during marking by a *Marker Port*.

The size of a private type value is known at compile time, therefore, in a node, a private type attribute is a value attribute.

The support for sets and sequences of private types is rather scant. Only creation of new list cells to store private type values is supported. The other set and sequence operations are not supported.

# 6 Runtime System Implementation

After the high level overview of the runtime system in chapter 4, this chapter describes the lower level details of the system.

This runtime system makes several assumptions about the target architecture. It assumes that the target machine has a byte addressable memory, that each address is 32 bits long, `floats` are 32 bits long, `doubles` are 64 bits long, and that standard types that are 32 bits or longer are allocated at (4 byte) word boundaries..

## 6.1 Private Types

IDL assigns a unique number `PvtTypeCode` to each private type and constructs a table called `PvtTypeTable`, indexed by `PvtTypeCode`. Each entry in this table pertains to a particular private type and contains six elements:

1. `ExtToInt`: a pointer to the routine that converts the external representation to the internal representation.
2. `IntToExt`: a pointer to the routine that does the inverse conversion.
3. `Initialize`: a pointer to the initialization routine for a value of this type.
4. `Mark`: a pointer to the marking routine for this private type.
5. `ExtType`: a one byte field specifying the type of the external representation.
6. `IntSize`: an integer specifying the size, in bytes, of the internal representation.

Another data structure used in the input and output of private types is a table known as the `ExtRepTable`. References to external representations of private type attributes are stored in this table.

## 6.2 Self Identifying Data

As mentioned in the previous chapter, all IDL data is self identifying. The header of each data object identifies it completely. An object is identified as belonging to a particular object type by the pattern of the two least significant bits in its header. Since there are only three data object types and four possible two bit patterns, the remaining pattern can be used to identify a special type of header known as the *indirect header* (Figure 2).

During the *mark* phase of the binary writer, the header of each object to be written out is replaced by an indirect header. This is done only for nodes and the system objects that occur in structure instances. In an indirect header, the remaining 30 bits form the relocation index of the object to be written out.

### 6.2.1 The Node Object

There can be two types of nodes. A node with no attributes is represented by an odd integer. A node with one or more attributes is represented by a pointer to a node object allocated in the IDL data memory. Since a node object is always allocated at (4–byte) word boundaries, a pointer to it is an integer which is always a multiple of four, and can be distinguished from a node with no attributes. The header of a node object consists of the following fields, as in Figure 2:

1. Process ID: this is an index into a process table maintained by the runtime system. The Process ID identifies the process by which the node was originally created.

2. Node Type: this is a non–negative integer, uniquely identifying the entries in the *invariant table* and the respective *port tables* that characterize this node. Node type $i$ is associated with the $2i^{th}$ entry in the port tables and the $i^th$ entry in the invariant table. In the port tables, the odd numbered entries are for nodes without attributes. In the invariant table, there are no entries for nodes without attributes.

3. Garbage Collection bit: This bit is used to mark objects in the scan phase of garbage collection.

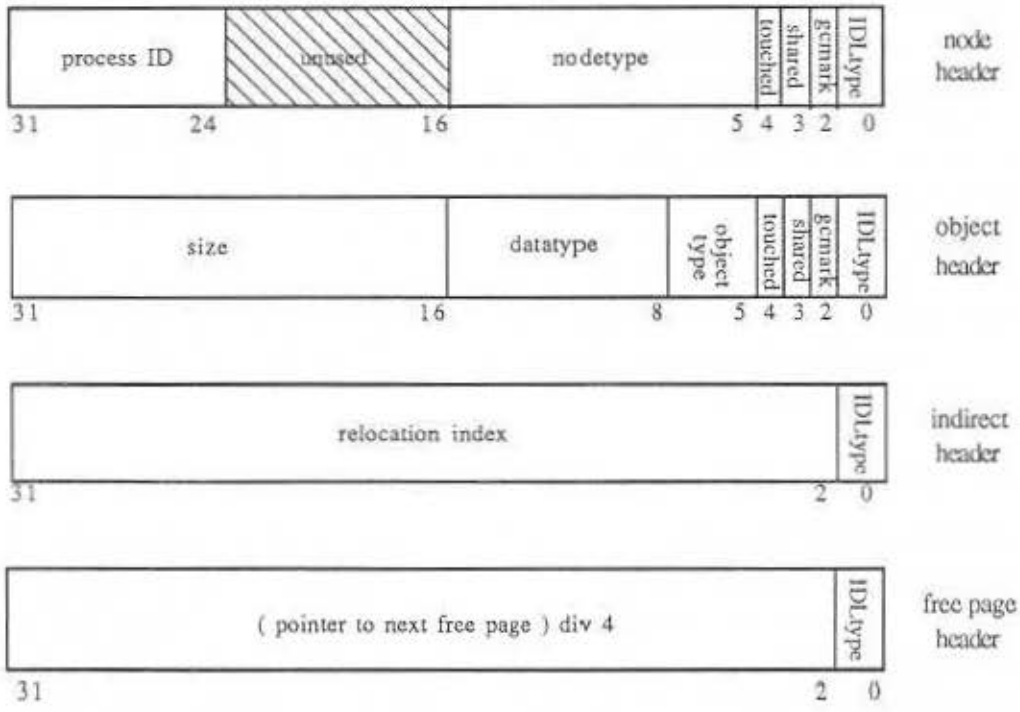



Figure 2: Object Headers

4. Touched bit: This is used by marker ports to mark all objects which are reachable from a certain given object. It is also used by the ASCII writer to mark objects that have already been written out.

5. Shared bit: This is also used by marker ports to mark objects that are shared within a structure.

The body of the node, as that of any other object, consists of an integral number of words containing the values of its attributes.

### 6.2.2 The System Object

The system object header has the following fields, as in Figure 2:

1. Size: size of the object in words.

2. System Object Type: type of the system object.

3. Data Type: this field identifies the type of data contained. It is relevant only in the header of a list cell object.

4. Garbage Collection bit.

5. Touched bit.

6. Shared bit.

There are eight types of system objects:

1. String,

2. List Cell,

3. List Header,

4. String Hash Table,

5. Label Table,

6. Relocation Table,

7. External Representation Table, and

8. Marking Buffer.

A string object consists of an object header, a pointer field, and a variable length data field, as in Figure 3. The data field contains a null terminated string of characters padded on the right so it occupies an integral number of (4 byte) memory words. All strings are stored in a hash table to be described in the next chapter. The pointer field is used to link all string objects that hash to the same location in the *string hash table.*

A *list cell* consists of a header, a next field containing a pointer to the next list cell in sequence, and a variable length data field that contains a value. The datatype field in the header of a list cell describes the type of data it contains.

A *list header* object consists of a header and a next field containing a pointer to the first list cell in the set or sequencer. This object is three words

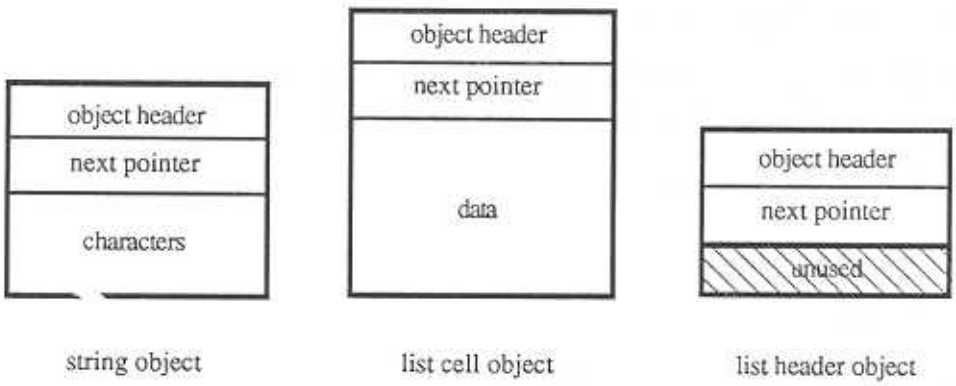



Figure 3: The String, List Cell, and List Header

long, even though it has only two words that serve any useful purpose. This is because the object management routines do not allow objects smaller than three words.

The *string hash table* is used to store string objects and provide quick access to them. It contains an array of buckets, each of which can store a pointer to a string object (Figure 4). When a string object is created, it hashes to one of these buckets. It is then added to the linked list of strings associated with this bucket. Searching for a particular string involves searching through the linked list of strings associated with the bucket it hashes to.

The *label table* object (Figure 5) is used in ASCII Reader to store and later resolve references to objects. Each entry in this table has three fields, namely, `label`, `resolved`, and `address`. The pointer field `label` is 31 bits long, the low order bit being used for the `resolved` field. No information is lost because the least significant two bits of `label` are zeroes. The first contains the symbolic label and the second indicates whether or not this reference has been resolved yet (Figure 6). If this reference is yet unresolved, it contains a pointer to the linked list of the *fixup* locations which should

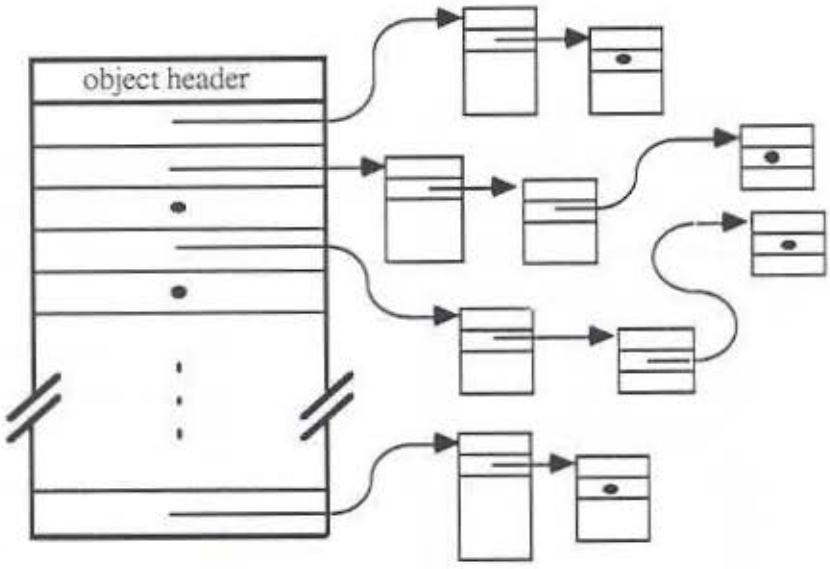



Figure 4: The String Hash Table

eventually contain the resolved reference. The `address` field contains the address of the first fixup location, which, in turn contains the address of the second fixup location, and so on. The last fixup location in the linked list contains the null pointer. In Figure 6, three references to an object labeled "L1234" have been read in before the object itself. Two of these references are within nodes, and one is in a list cell. In the label table entry for this label, the *0* in the `resolved` field indicates that the references have not been resolved yet, the `address` field contains the first link in the chain of fixup locations, and the `label` field contains a reference to the string object representing the label "L1234". While reading is in progress, if a reference to an unresolved label is encountered, its location is added to this linked list. After the object to which an unresolved label refers is read, the address of the object is stored in the `address` field of the corresponding label entry and its `resolved` field is set to *1*. Then the linked list of locations is traversed and the object address is stored in each location encountered (Figure 7). In this figure, the references have been resolved to point to the newly read in node object.

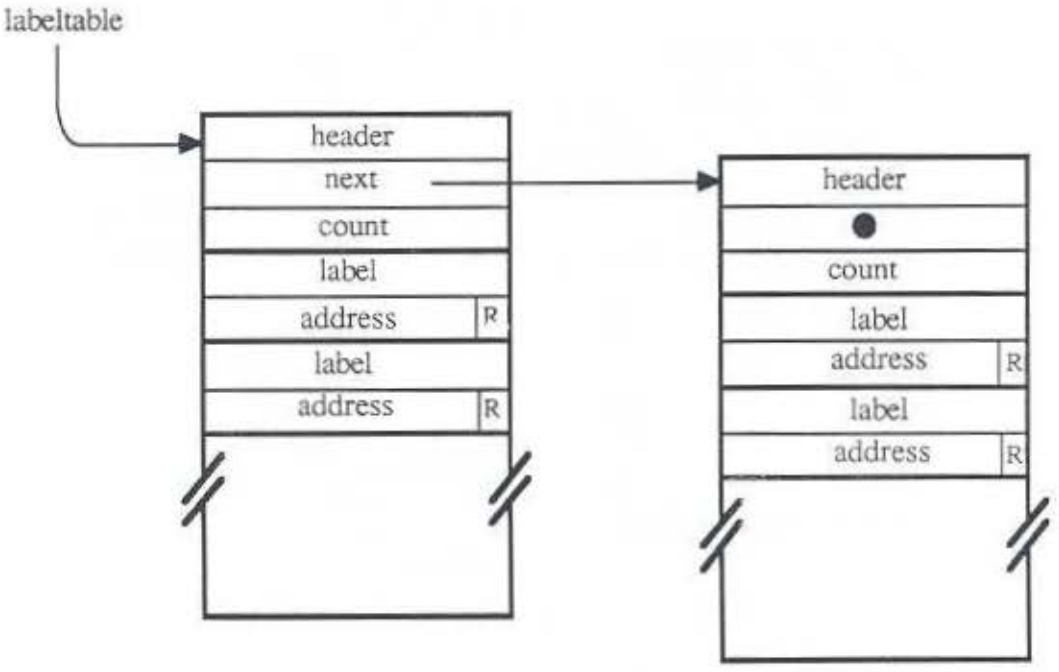



Figure 5: The Label table

The *label table* object is also used by the binary reader. While the ASCII reader uses it to resolve references to symbolic labels, the binary reader uses it to resolve references to relocation indices. Each entry now has only two fields, `resolved`, and `address`. The counterpart to the symbolic label field is the relocation index, which is an implied field, equal to the offset of the entry in the label table. The storage of unresolved references and the resolution of references is identical to the corresponding functions in the ASCII reader.

The *relocation table* object is used by the binary writer. Stored in each entry are the header of the object to be written out, and a pointer to the actual object. Before an object is written out, its header is stored away in this entry and a relocation index is put in its place. This index uniquely identifies the entry in the relocation table which contains the header. When the entire structure has been written out, the relocation indices are discarded and the headers restored.

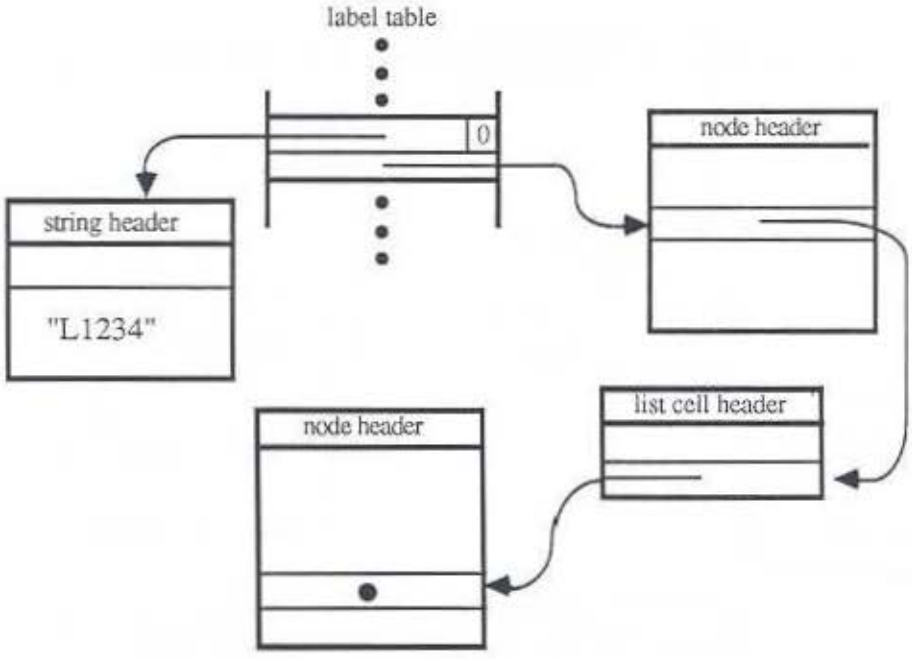



Figure 6: An Unresolved Label Entry

The ASCII Writer uses the relocation table to store addresses of objects that have been marked and written out. After the writing operation is over, this table is used to locate the objects that were written out so their 'marks' can be removed.

The *external representation table* is used by the reader to store external representations of private type objects, before their conversion to internal representations.

The *label table*, *relocation table*, and *external representation table* object are implemented as arrays of entries. Each table has its own object header, a `count` field indicating how many valid entries it contains, and a `next` field through which another table object of the same kind can be linked in case of overflow(Figure 5). These tables are allocated from the IDL data memory as and when they are needed. This is done while input or output is going on. No tables are allocated during garbage collection. Pointers to them are stored in variables global to the entire runtime system. They are garbage collected when they are not in use.

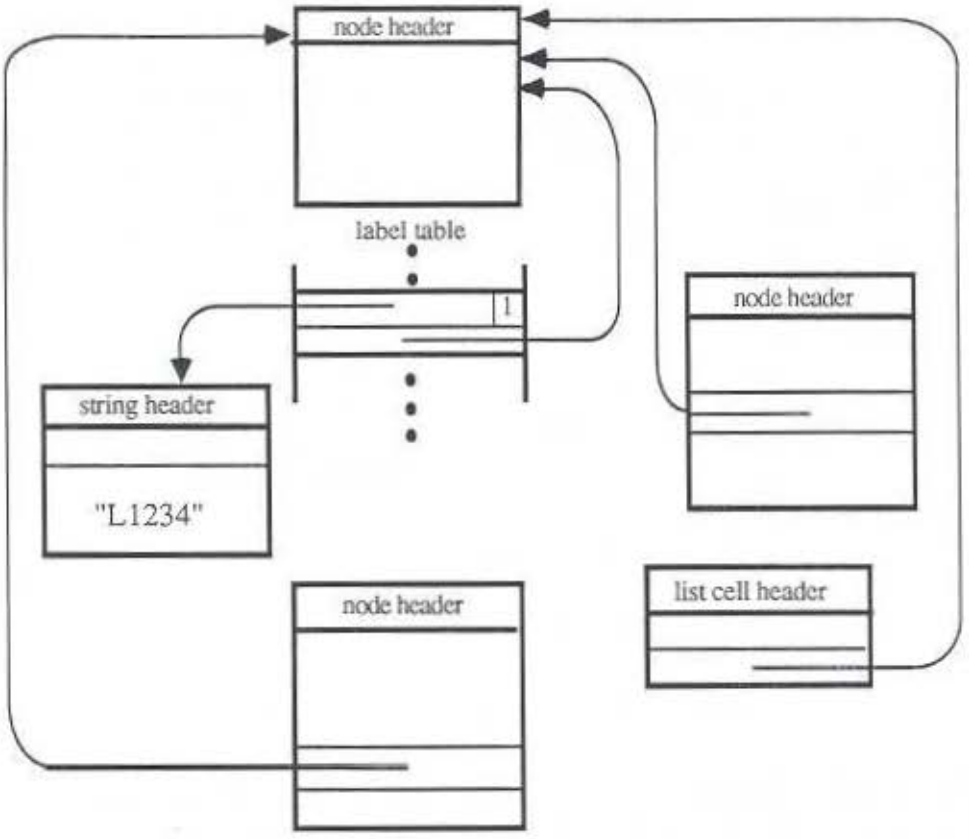



Figure 7: A Resolved Label Entry

The *marking buffer* is used during the *mark* phase of the garbage collector. All potential pointers to data objects found on the runtime stack are stored in this buffer and sorted. Then they are compared with the addresses of actual data object headers. The ones that match are recursively marked and others are discarded. If there is not enough room in the buffer to take all possible pointers from the runtime stack, this process is repeated until the stack is exhausted.

### 6.2.3 The Free Page Object

The pool of free pages is a circular doubly linked list of free page objects. The 30 more significant bits of the free page header form the size of the

free page in (4 byte) words.

The two words following the header contain the forward and backwards links into the free pool. Since each free page needs three words to store its two pointers and its size, it has to be at least three words long.

### 6.2.4 IDL Memory Layout

Figure 8 illustrates the memory layout of an IDL process runing under UNIX on a Sun workstation or a VAX. The low end of the memory is

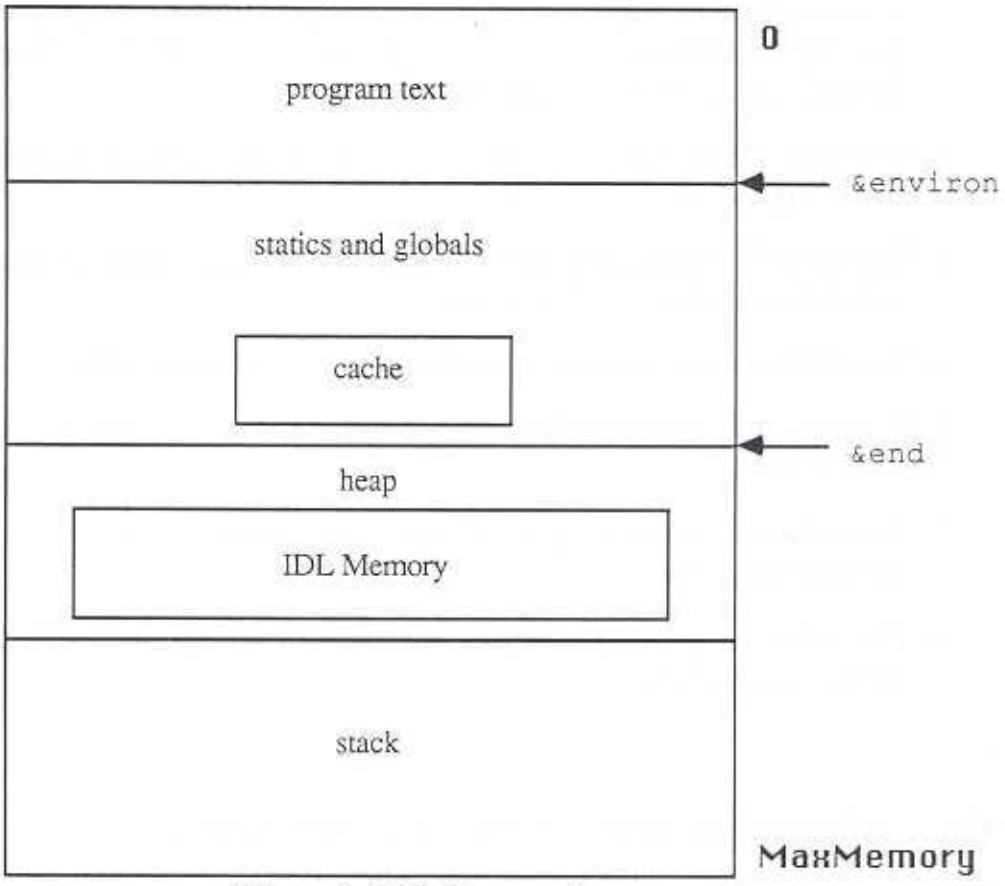



Figure 8: IDL Memory Layout

occupied by the program text. Above that is the initialized *static* and *global* data. Following this area is the storage for uninitialized static and

global data. The remaining storage is used by the heap and the runtime stack. The heap grows starting at the low end of the remaining space and the stack starts at the high end. The IDL data memory is part of the heap. The UNIX runtime system supplies two pseudo–variables that mark the boundaries around the data area. `environ` and `end` are allocated at the start and end of the data area, respectively. The uninitialized global variable area contains a cache of global variables with the help of which all IDL data objects can be located.

1. The variable `IDLMemory` of type `* int` points to the first location of the IDL data memory. The size of this memory in words is given by the variable `IDLMemorySize` of type `*int`.

2. The pool of free pages is pointed to by the global variable `IDLFreeList` of type `* int`.

3. The variable `InvTable` points to the *invariant table*, with the help of which all nodes can be described.

4. The *label table* is pointed to by the variable `IDLLabelTable`.

5. The *external representation table* is pointed to by the variable `IDLExtRepTable`.

6. The *relocation table* used by the reader is pointed to by the variable `IDLReadRelTable`.

7. The *relocation table* used by the writer is pointed to by the variable `IDLWriteRelTable`.

## 6.3 Allocation and Initialization of Objects

Initially, the free pool contains only one free page object of a certain constant size, but after garbage collection, it is likely to contain several free pages of different sizes (Figure 9). The first-fit algorithm is used for space allocation. All data objects allocated are at least three (4–byte) words long. The smallest allocated object cannot be smaller than the smallest possible

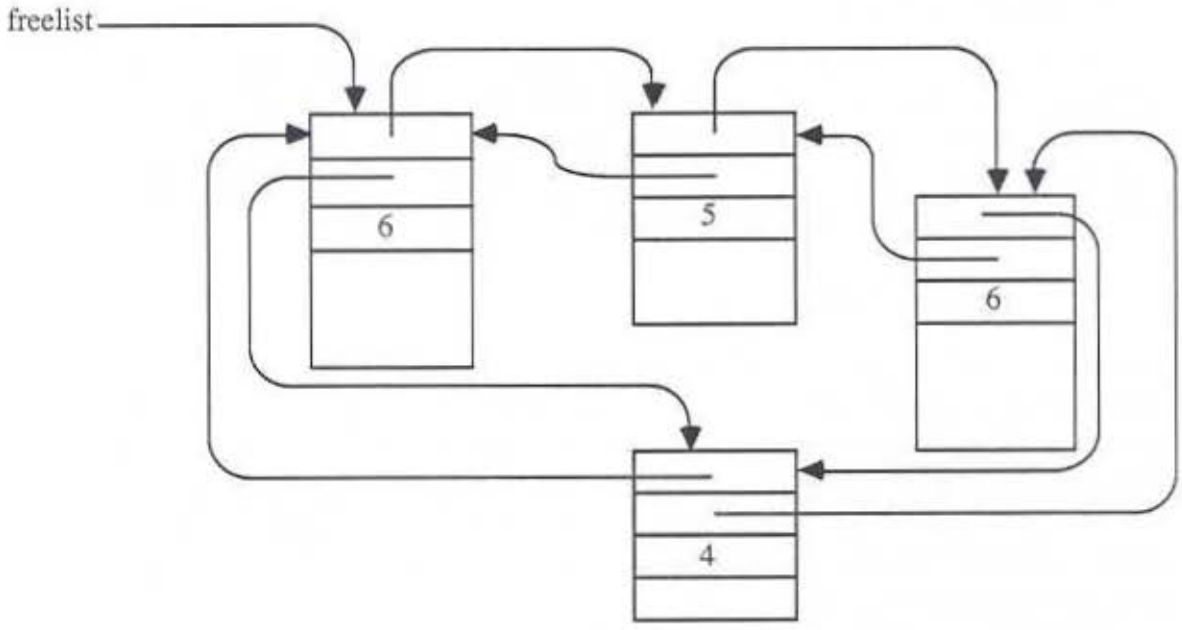



Figure 9: The Free Page Pool

free page, because if it is garbage–collected, it may have to be transformed into a free page by itself.

For each nodetype defined, the user may provide a node initialization routine. At the time of creation, each node is a chunk of memory with an appropriate header. Before it is returned to the user, it is initialized, using the initialization routine provided by the user. In case such a routine is not provided, all attributes of the node are initialized to system defined defaults which are as follows:

1. Boolean: FALSE,
2. String: NULL,
3. Integer: Zero,
4. Rational: Zero,
5. Set: NULL,

6. Sequence: NULL.

The user must provide an initialization routine for each private type. Whenever a list cell is created, it is also initialized according to its data type.

An `Integer` attribute may be one, two or four bytes long. A `Boolean` is a stored in a byte. The bit pattern consisting of all zeroes represents the value *FALSE* and anything else represents the value *TRUE*.

## 6.4 Garbage Collection

In order to collect garbage, all global variables in the program, as well as local variables of the currently active procedures, have to be accessed. For purposes of garbage collection, all memory locations in the runtime stack and the global and static variable area of the currently executing process are assumed to contain pointers to IDL data objects. As explained in the previous chapter, this assumption, though incorrect, leads to a conservative, but valid method of garbage collection.

Whether or not any input or output is in progress is indicated by the global variables `IDLbinwriting`, `IDLbinreading`, `IDLasciiwriting`, and `IDLasciireading`. Clearly, at any given time, at most one of these variables can be true. When the garbage collector is invoked, it tests these variables to find out if at all any input or output is in progress, and if so, what type. The garbage collector is aware of the kind of tables mentioned above that each type of input or output uses.

If the garbage collector is called during input or output, it starts with marking objects accessible from the relocation tables currently in use. It also marks the tables themselves. If the external representation table is also in use, it marks object accessible from that, too.

The global and static variable area is between the addresses of the variables `environ` and `end` provided by the C runtime system on UNIX. All local variables are on the runtime stack. The address of the first automatic variable declared in the `main` procedure marks the bottom of the runtime stack and one of the local variables in the garbage collection routine can be considered to be at the top of the stack. The user is required to make a call to the procedure `IDLInit` provided by the runtime system. This call must be made only once and must precede any call to any other procedure provided by the runtime system. `IDLInit` takes one parameter, which should

be the address of the first automatic variable declared in the `main` procedure. `IDLInit` initializes the runtime system and establishes the bottom of the runtime stack.

The following conditions are tested for on each (4–byte) word in the variable and stack areas.

1. It points into the data memory area.
2. It points at a word boundary.
3. It points to a word that looks like a valid *unmarked* node header or a valid *unmarked* header of a string, list cell or list header object.

If it tests true on all of the above three conditions, it is considered to be a potential pointer to a data object. All potential pointers are stored in the *marking buffer* object and sorted. Then the data memory is scanned from top to bottom along with the marking buffer to determine which of these pointers actually point to objects. All pointers in the buffer that point to an unmarked node or object are recursively marked.

The recursive marking process is carried out as follows.

1. A node is marked by setting the *garbage collection bit* in its header, and marking all of its non–null reference attributes and private type attributes.
2. A list header object is marked by setting the *garbage collection bit* in its header and marking the list cell pointed to by its `next` field.
3. A list cell is marked by setting the *garbage collection bit*, marking the list cell pointed to by its `next` field, and marking its data if its data is a node or a string or a private type.
4. A string object is marked by setting the *garbage collection bit*.
5. A private type object is marked by calling the `Mark` routine provided for that private type by the user. This routine takes as its argument a pointer to the garbage collection marking routine.

After the marking process is over, the headers of the marking buffer and the string hash table are marked. These objects are not to be garbage collected. Then all the string objects are scanned. Those that are found to be unmarked are de–linked from the string hash table so they can be garbage collected. This step is necessary to eliminate dangling pointers to garbage–collected string objects.

At this time all garbage is unmarked and ready to be collected. The pool of free pages is now initialized to null. A top to bottom scan of the data memory is carried out. All free pages and unmarked nodes and objects are garbage. They are linked into the free pool. Since this is a linear scan, it is easy to detect adjoining objects to be garbage collected. They are coalesced into one large free page and linked into the free list. During this scan, all marked objects are made unmarked again. At the end of this scan, garbage has been collected, and all useful data objects are as before. Hopefully, more space is available for allocation and the process can proceed with its task.

## 6.5 Performance

In this section we analyze the performance of the runtime system, in terms of both space and time.

### 6.5.1 Space Overhead

The following is the overhead on data objects:

1. Each node object has an overhead of *4 bytes* that comprise its header.
2. Each string has an overhead of *4 bytes* on its header, *4 bytes* on its next pointer, and on average *1.5 bytes* due to the requirement that the size of each object be a multiple of (4 byte) words.
3. Each set or sequence has an overhead of *12 bytes* used by the list header.
4. Each element of a set or a sequence has an overhead of *8 bytes* used by its header and its next pointer.

5. There is a constant overhead due to the *Marking Buffer*.

The following is the overhead during input or output:

1. There is an overhead of one relocation entry per object read during a binary read. The relocation entry size is *4 bytes*.

2. There is an overhead of one relocation entry per object written out during a binary write. The relocation entry size is *8 bytes*.

3. There is an overhead of one label entry per label read during an ASCII read. The size of a label entry is *8 bytes*.

4. There is an overhead of one external representation entry for each private type value read in, during the ASCII or the binary read operation. The size of the external representation entry is *12 bytes*.

There is no overhead during garbage collection.

### 6.5.2 Time Performance

The following is the analysis of the performance of input and output operations, assuming that no garbage collection occurs during the operation.

1. **Writing**: This involves a pass over the entire structure to be written out, and the traversal of all pointers within the structure instance. Since the size of the instance is larger than the number of pointers in the instance, The ASCII write operation is $\Theta(size(instance))$, where $size(instance)$ is the total size of all objects that comprise this structure instance.

2. **Reading**: This involves reading in the entire structure and doing fix–ups on the forward references as they are read in. Once the entire structure instance is read in, all private type values are translated from their external types to their internal types. This operation is also $\Theta(size(instance))$.

The following is an analysis of the performance of operations relating to object management:

1. **Allocation and Initialization of objects:** Object allocation involves traversing the list of free pages, until one at least as large as the object to be allocated is found. This takes, on average, time proportional to the length of the list of free pages. Then this object is initialized, which takes time proportional to the size of the object.

2. **Garbage Collection:** The 'mark' phase involves scanning the global variable and the stack areas, picking out potential pointers to data objects, and following them. We compute the probability that one of the words scanned is a potential pointer, assuming that all bit patterns are equally likely. The probability that a given word is a valid unmarked node header is:

$$p(nh) = \frac{1}{4} \times \frac{1}{2} \times \frac{2^7}{2^{11}}$$

In the above formula, the first factor denotes the probability that the word has the two bit pattern that classifies it as a node header. The second factor is the probability that its garbage collection bit is off. The third factor is the probability that it is a valid nodetype. The assumption here is that of the 11 bits available to represent a nodetype, not more than 7 will actually be used in a typical program.

Similarly, the probability that a given word is a valid unmarked header of a string, list header, or a list cell is:

$$p(oh) = \frac{1}{4} \times \frac{1}{2} \times \frac{3}{8}$$

The first factor denotes the probability that the word has the two bit pattern that classifies it as an object header. The second factor is the probability that its garbage collection bit is off. The third factor is the probability that it is a header of either a string, list header, or a list cell, of the possible 8 object types.

The probability that a word is a pointer to a word in the IDL data memory is:

$$p(p) = \frac{1}{4} \times \frac{2^{20}}{2^{32}}$$

The first factor is the probability that it points at a word boundary. The second factor is the probability that it points into the IDL data

memory, which is assumed to be one megabyte in size for a typical program.

Therefore, the probability that a pointer is a potential pointer to an object that should be marked for garbage collection is:

$$p(pp) = p(p) \times (p(nh) + p(oh))$$

$$p(pp) \approx 2^{-18}$$

From the above analysis, if we provide a marking buffer of size 1000 words, it is highly unlikely that it will overflow. This marking buffer is to be sorted, which is $\Theta(n^2)$, where $n$ is the number of potential pointers. After this, all accessible objects are marked, which is $\Theta(d)$ where $d$ is the number of objects in the data memory. After marking is over, a linear scan of the data memory is made and garbage is collected, which is again $\Theta(d)$. Since the number of objects in the data memory is likely to be much larger than the number of potential pointers, the entire garbage collection operation is $\Theta(d)$.

# 7 Conclusion and Future Work

As currently implemented, the runtime system consists of

1. Reader and Writer routines, both ASCII and relocatable binary.
2. Object Management including a mark–and–scan garbage collector and a memory display facility.
3. Support for sets and sequences of objects.

This runtime system does not support fully general IDL. The restrictions are as follows:

1. The ASCII ERL representation of a structure instance is, in general, a list of subgraphs with references from one subgraph to another through labels. Each subgraph has a *root*, i.e., the object enumerated first. This runtime system supports only node objects as roots. This restriction occurs only in the ASCII reader.
2. In the ASCII ERL representation of a structure instance, only nodes, sets and sequences may be labeled.
3. Only the linked–list implementation of sets and sequences is supported. Other possible implementations include arrays and bit vectors.
4. Sets and sequences cannot be elements of other sets and sequences.
5. A node object cannot be of size less than 3 words, or 12 bytes.
6. The `String`, `Rational`, `Set`, and `Sequence` attributes are allocated at word boundaries.
7. An `Integer` attribute is allocated at a word boundary if its representation is four bytes long.
8. Sets and sequences of private types are not adequately supported.

## 7.1 Future Work

The ASCII reader could be modified to support fully general ASCII ERL.

The object management system allocates space for the IDL data memory just once. If there is no available space even after garbage collection, the user is notified. The system could be modified so that in such a situation it attempts to enhance the IDL data memory so that the allocation request can be met.

The garbage collection mechanism, in its mark phase, uses a recursive routine that does a depth–first search. This operation, being recursive, can potentially use up a lot of stack space, which may cause problems, because the reason garbage collection is initiated is that there is not very much space available, anyway. The marking phase can be modified to run in accordance with the Schorr-Waite marking algorithm which, though relatively complicated, uses very little stack space in comparison [8].

In this system, the reference to a data object is through its header, i.e., the header is at word offset 0 and all useful space in the object starts at word offset 1. Usually user code assumes that useful space in a newly allocated object begins at offset 0, as returned by `malloc`. Thus, for the sake of uniformity, it is advisable to refer to an object through the word that occurs after the header [4]. In other words, the header should be at word offset -1. In the *string* and *list cell* objects, however, the header is followed by a pointer, after which the useful space begins. To ensure that in all objects the user space begin immediately after the header, this pointer would have to be moved so it occupies the last word of the object.

It is desirable to provide private `touched` and `shared` bits for each *marker port*. If objects with only one `touched` and one `shared` bit have been marked by more than one marker port, it is impossible to determine which port has marked which object.

# 8 Bibliography


1. Cohen, J. Garbage Collection of Linked Data Structures. *ACM Computing Surveys 13*, 3 (1981), 341-368.

2. Collins, G. A method for overlapping and erasure of lists. *Communications of the ACM 3*, 12 (December 1960), 655-657.

3. Deutsch, L. P., and Bobrow, D. G. An Efficient, Incremental, Automatic Garbage Collector. *Communications of the ACM*, (July 1976), 522-526.

4. Lamb, D. A. Private Communication.

5. Nestor, J. R., Wulf, W. A., and Lamb, D. A. **IDL Formal Description, Draft Revision 2.0.** 3, Computer Science Department, Carnegie-Mellon University, (June 1982).

6. Newcomer, J. M. *IDL: The Language and Its Implementation.* unpublished, (1986).

7. Pratt, T. W. *Programming Languages, Design and Implementation.* Prentice-Hall, Inc., Englewood Cliff, NJ 07632, (1984).

8. Schorr, H., and Waite, W. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM 10*, 8 (August 1967), 501-506.

9. Snodgrass, R. *The Interface Description Language: Definition and Use (forthcoming).* Computer Science Press, Rockville, MD, (1988).

10. Teitelman, W. **The Cedar Programming Environment: A Midterm Report and Examination.** CSL-83-11, Xerox Corporation, Palo Alto Research Center, (June 1984).

11. Warren, W. B., Kickenson, J., and Snodgrass, R. T. **A Tutorial Introduction to Using IDL.** 1, (Softlab Document), Computer Science Department, University of North Carolina at Chapel Hill, (November 1985).

12. Wegbreit, B. A Generalized Compactifying Garbage Collector. *The Computer Journal*, (1972), 204-208.

# A The Runtime Interface

This appendix describes the interface between an IDL process and the IDL runtime system. The IDL translator, IDLC, processes the IDL specification and produces a `.c` and a `.h` file. In the next appendix, a complete example IDL specification is presented, along with the `.c` and the `.h` files generated by the IDL translator. The `.c` file must include the `.h` file. The following include statement should go at the top of the `.h` file:

```
#include "idlruntime.h"
```

If the process uses any private types, the user-supplied `.h` files for the private types must also be included here.

The file `idlruntime.h` contains `extern` declarations of all the functions available from the runtime system. It also contains definitions of all tables generated by IDLC.

The `.h` file contains `struct` definitions for all `nodenames` in the IDL specification, so that the attributes of node objects may be accessible to the user as fields of a structure. It contains `#define` statements that associate with each *nodename* its integer `nodetype`.

The remainder of the `.c` file contains the initialization of the invariant table for the IDL process, the binary port tables for each port in the process, the ASCII port tables for each ASCII port, the Private Type Table, and the code for the port routines.

## A.1 Aspects

The attribute type of each attribute in each node is represented by an 8-bit number known as the *aspect* (Figure 10) in the invariant table, the port table, and the private type table. The aspect contains information about the representation of the attribute, whether it is a private type, and whether it is a set or a sequence of some type. The least significant 5 bits represent the type, which ranges from *0* to *31*. The next two bits indicate whether this is a 'simple' type, a set, or a sequence. If it is not a simple type, the most significant bit indicates whether it is represented as a linked list or an array.

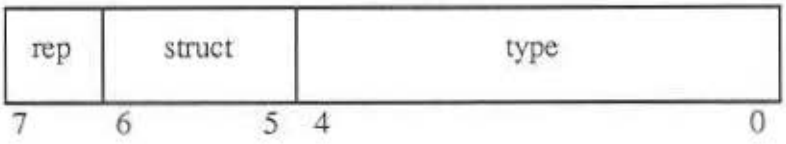



Figure 10: Aspect

```
/* aspect fields */

#define Node            0
#define String          1
#define Boolean         2
#define OneInt          3
#define TwoInt          4
#define FourInt         5
#define Float           6
#define Double          7
#define ATTTYPE         0x1F
#define BASICTYPECOUNT  8
/* 8 and above are private types */

#define Simple          0
#define SetOf           0x20
#define SeqOf           0x40
#define ATTSTRUCTURED   0x60

#define Linked          0
#define Array           0x80
#define ATTREP          0x80
```

## A.2 The Invariant Table

The following is the definition of the `invariant_table_entry` data type.

```
typedef struct invariant_table_entry
{
```

```
  void (*initialize)();
  short size;
  short att_count;
  {
    char aspect;
    short offset;
   } *attributes
};
```

The invariant table is an array of type `invariant_table_entry`. It has an entry for each `nodetype`. Each entry contains the size of the node in (4 byte) words, the number of attributes it has, and an array of its attribute characteristics. The invariant table is indexed by `nodetype`. An attribute is identified by its starting location relative to the node header. The entries for attributes of nodes do not have to be in any particular order. Once an order has been established, it is used by each port to identify attributes within a node.

## A.3 The Port Table

The following is the definition of the `port_table_entry` data type.

```
typedef struct port_table_entry
{
  short type;
  short att_count;
  {
    char aspect;
    char index;
  } *attributes
};
```

There is one port table for each IDL port. This table consists of an entry for each nodetype. An input port table is indexed by the port table nodetype, while an output port table is indexed by the invariant nodetype. Each entry in the table contains the size of the node, the corresponding nodetype in the invariant or at the port, depending on whether this table is for reading

or writing, and an array which contains an entry for each attribute and for each stretch of unused space in the node. If this node has no attributes, the array mentioned above is null. Each non-negative index in this array points to an attribute in the corresponding node in the invariant table and the modulus of each negative index indicates so many bytes of unused space. The entries for attributes have to be in increasing order of distance of the attribute from the node header.

## A.4 ASCII Port Tables

There is one ASCII port table for each ASCII port. Below is the definition of an ascii port table.

```
typedef struct ascii_port_table
{
  short tablelength;
  {
    char *name;
    short nodetype;
    short tablelength;
    {
      char *name;
      char index;
    } *attribute_table;
  } *node_table
};
```

A port table has an array of entries, one for each nodetype in the structure. Each node entry contains the nodetype, the ASCII node name and an array of attribute entries. An attribute entry contains the ASCII name of the attribute and its index in the corresponding node entry in the invariant table.

In a table for an input port, the list of nodes is sorted in alphabetical order of node names, and the list of attributes is sorted in alphabetical order of attribute names.

In a table for an output port, the list of nodes is sorted in ascending order of nodetypes, and the list of attributes is sorted in ascending order

of indices.

## A.5 Private Type Table

This table is generated by the IDL translator. It has as many entries as there are private type definitions in the IDL data specification. The Private Type Table is an array of the following type.

```
typedef struct pvt_type_entry
{
  void (* ExtToInt)();
  void (* IntToExt)();
  void (* Initialize)();
  void (* Mark)()
  short IntSize;
  char ExtType;
};
```

ExtToInt is a procedure that converts from an external representation to an internal representation. Its first parameter is a pointer to a buffer. If the external representation is a set, sequence, node or a string, this buffer contains a pointer to it. If the external representation is a scalar, then it is contained in the buffer. The second parameter is a pointer to a buffer in which the corresponding internal representation is to be returned.

IntToExt is an exact inverse of ExtToInt.

Initialize is a procedure that initializes a private type value. Its only argument is a pointer to the first storage byte of the value.

Mark is a procedure that recursively marks all objects reachable through a private type value. It takes as its arguments a pointer to the first storage byte of the value and a pointer to the garbage collection mark routine.

IntSize is the size in bytes of the internal representation of an instance.

ExtType is the *aspect* of the external representation.

There are eight basic types of IDL attributes represented by numbers 0 through 7. Numbers 8 through 31 can be used to represent private types, i.e., the number of private type definitions in a process is limited to 24. The $i^{th}$ entry in the Private Type Table contains specifications for the private type represented by the number $i + 8$.

## A.6 IDL Runtime Routines

The routine to initialize the runtime system is

```
void IDLInit(stkbottom)
char *stkbottom;
```

In the above, `stkbottom` is the address of the first automatic variable declared in the user's `main` procedure.

The following routines invoke the binary reader and the binary writer, respectively.

```
pnodeheader IDLReadIn(table, instream)
struct port_table_entry *table;
FILE *instream;

int IDLWriteOut(rootstruct, table, outstream)
pnodeheader rootstruct;
struct port_table_entry *table;
FILE *outstream;
```

The argument `table` refers to the port table used for reading or writing. The arguments `instream` and `outstream` refer to the streams to be used for reading and writing, respectively. If successful, `IDLReadIn` returns a pointer to the root of the structure read in; otherwise it returns the `NULL` pointer. The argument `rootstruct` points to the root of the structure to be written out. If successful, `IDLWriteOut` returns 1, otherwise it returns 0.

The following routines invoke the ASCII reader and writer, respectively.

```
pnodeheader IDLAsciiReadIn(table, instream, nodenametable)
struct port_table_entry *table;
FILE *instream;
struct ascii_port_table *nodenametable;

int IDLAsciiWriteOut(rootstruct, table, outstream, nodenametable)
pnodeheader rootstruct;
struct port_table_entry *table;
```

```
FILE *instream;
struct ascii_port_table *nodenametable;
```

The ASCII reader and writer take an extra argument, nodenametable is the address of the ASCII port table. Their return values and other arguments are similar to their binary counterparts.

The marker is invoked by the following routines.

```
void IDLMark(rootstruct, table)
pnodeheader rootstruct;
struct port_table_entry *table;

void IDLUnMark(rootstruct, table)
pnodeheader rootstruct;
struct port_table_entry *table;
```

IDLMark marks the structure instance. IDLUnMark restores it to its original unmarked state.

The allocation routines for nodes and system objects are the following.

```
pnodeheader IDLNewNode(type)
short type;

pIDLstring IDLNewString(str)
char *str;
```

The size of a node is implicit in its type. The argument str is a pointer to a null terminated sequence of characters. Allocation of strings requires special treatment because each string must be entered in the global IDL-StringHashTable to facilitate comparison among strings and to avoid duplication of strings.

The following routines are provided mainly as debugging aids:

```
void IDLScanMemory(f)
void (*f)();

void IDLPrintMemory()
```

The function denoted by `* f` is applied to the header of each node, system object or free page in memory. It takes one parameter, of type `* int`, a pointer to the header of the object in question. `IDLPrint` is a special case of `IDLScanMemory` in which `*f` is a function which prints out the headers of objects on a global called `IDLPrintStream` of type `*FILE`.

Although the garbage collector is invoked by the runtime system whenever necessary, the garbage collection routine is also made available to the IDL process.

```
int IDLGarbageCollect()
```

`IDLGarbageCollect` returns the size of the largest chunk of free memory in words.

# B An Example

A sample IDL specification is presented, along with the generated `.c` and `.h` files.

The IDL specification is in the file `ex.idl`. The algorithm is in `algorithm.c`. The declarations and procedures for the private type are in the files `CharSet.h` and `CharSet.c`. The files generated by the IDL translator are `InOut.h` and `InOut.c`. The ASCII ERL input file is `ascinput` and the ASCII ERL output file is `ascoutput`. The program writes into `statusfile`.

This example, along with a `README` file and a `Makefile`, can be found in `/usr/softlab/doc/examples/biyanithesis` on the Suns.

```
-- ex.idl
Structure Structin Root Anode Is
        Anode => List: Seq Of B,
                 Name: CharSet;

For CharSet Use Package CharSet;
For CharSet Use Type CharSet;
For CharSet Use External Representation String;

        B =>     X: Boolean,
                 R: Rational;
End

Structure Structout Root Anode From Structin Is

        Without B => X;

        Anode => Number: Integer;
End

Process Inout Is
        Target Language C;
        Target Runtime Version B;

        Pre P: Structin;
        For P Use Ascii External Representation;

        Mark M: Structin;

        Post Q: Structout;
        For Q Use Binary;

        Pre R: Structout;
        For R Use Binary;

        Post S: StructIn;
        For S Use Ascii External Representation;
End
```

```
/* algorithm.c */
#include "InOut.h"

Anode thisAnode;
B newB;
SEQB seqBptr;

main()
{
        FILE *inascii, *outbinary, *inbinary, *outascii;

        /* initialize idl */
        IDLInit(&inascii);
        IDLPrintStream = fopen("statusfile", "w");

        /* input the initial structure in binary */
        inascii = fopen("ascinput", "r");
        thisAnode = P(inascii);
        (void) fclose(inascii);

        fprintf(IDLPrintStream,
                "\n\nAscii input has been read in\n\n");
        fprintf(IDLPrintStream,
                "\n\nGarbage Collected, largest free page size: %d words\n\n",
                IDLGarbageCollect());
        IDLPrintMemory();
        removefirstSEQB(thisAnode->List);
        fprintf(IDLPrintStream,
                "\n\nOne node has been removed from the sequence of B nodes\n\n"
               );
        fprintf(IDLPrintStream,
                "\n\nGarbage Collected, largest free page size: %d words\n\n",
                IDLGarbageCollect());
        IDLPrintMemory();
        /* get a new B node object */
        newB = NB();

        /* give its attributes some values */
        /* and append it to thisAnode->List */
        newB->X = TRUE;
        newB->R = 3.14;
        appendfrontSEQB(thisAnode->List, newB);

        thisAnode->Number = 10;
        /* output in binary */
        outbinary = fopen("binfile", "w");
        Q(thisAnode, outbinary);
        (void) fclose(outbinary);

        fprintf(IDLPrintStream,
                "\n\nBinary Output has been done\n\n");
        fprintf(IDLPrintStream,
```

```
        "\n\nGarbage Collected, largest free page size: %d words\n\n",
        IDLGarbageCollect());
IDLPrintMemory();

/* input in binary*/
inbinary = fopen("binfile", "r");
thisAnode = R(inbinary);
(void) fclose(inbinary);

fprintf(IDLPrintStream,
        "\n\nBinary Input has been done\n\n");
fprintf(IDLPrintStream,
        "\n\nGarbage Collected, largest free page size: %d words\n\n",
        IDLGarbageCollect());
IDLPrintMemory();

foreachinSEQB(thisAnode->List, seqBptr, newB)
        newB->X = TRUE;

/* output in Ascii */
outascii = fopen("ascoutput", "w");
S(thisAnode, outbinary);
(void) fclose(outascii);
fprintf(IDLPrintStream,
        "\n\nAscii Output has been done\n\n");

/* Mark */
M(thisAnode, TRUE);
fprintf(IDLPrintStream,
        "\n\nThe instance read in has been marked\n\n");
IDLPrintMemory();
fprintf(IDLPrintStream, "\n\n");
/* UnMark */
M(thisAnode, FALSE);
fprintf(IDLPrintStream,
        "\n\nThe instance read in has been unmarked\n\n");
IDLPrintMemory();

/* make all variables null, so everything may be garbage collected */
thisAnode = NULL;
newB = NULL;
seqBptr = NULL;
fprintf(IDLPrintStream,
        "\n\nAll variables have been made null;\n");
fprintf(IDLPrintStream,
        "everything is garbage collected\n\n");

fprintf(IDLPrintStream,
        "\n\nGarbage Collected, largest free page size: %d words\n\n",
        IDLGarbageCollect());
IDLPrintMemory();
fclose(IDLPrintStream);
```

```
}
```

```
/* CharSet.h */
typedef int CharSet;
```

```
/* CharSet.c */
#include "/usr/softlab/src/biyanilibrary/src/idlruntime.h"

        void StringToCharSet(from, to)
        char *from, *to;
        {
                char c, *str;
                int i, *p;
                p = (int *)to;
                str = ((pIDLstring)(*(int *)from))->data;
                *p = 0;
                i = 0;
                for (c = 'a';c <= 'z'; c++) {
                        if (str[i] == '\0')
                                return;
                        else if (str[i] == c) {
                                *p |= (1 << (c - 'a'));
                                i++;
                        }
                }
        }

        void CharSetToString(from, to)
        char *from, *to;
        {
                int *p, i, j;
                char c, ch[27];
                i = *(int *)from;
                j = 0;
                for (c = 'a';c <= 'z'; c++) {
                        if ( i & 1)
                                ch[j++] = c;
                        i >>= 1;
                }
                ch[j] = '\0';
                *(int *)to = (int)IDLNewString(ch);
        }

        void InitCharSet(from)
        char *from;
        {
                *(int *)from = 0;
        }

        void MarkCharSet(from, f, table)
        char *from;
        void (*f)();
        int *table;
        {}
```

```
/* InOut.h */
#include "/usr/softlab/src/biyanilibrary/src/idlruntime.h"
#include "CharSet.h"
#define KAnode 0
#define KB 1

typedef plistcell SEQB;
typedef SEQB SETB;
typedef plistcell SEQAnode;
typedef SEQAnode SETAnode;

typedef struct RB {
        nodeheader header;
        Bool X;
        char unused[3];
        float R;
} *B, RB;
#define NB() (B)IDLNewNode(KB)
typedef struct RAnode{
        nodeheader header;
        SEQB List;
        CharSet Name;
        int Number;
} *Anode, RAnode;
#define NAnode() (Anode)IDLNewNode(KAnode)

extern Anode P();
extern void M();
extern void Q();
extern Anode R();
extern void S();


/* macros for set and sequences */
#define appendfrontSEQB(nseq, nval)  \
        appendfrontseq(nseq,(char *)(&nval), Node)
#define appendrearSEQB(nseq, nval) \
        appendrearseq(nseq,(char *)(&nval), Node)
#define copySEQB(nseq) (SEQB)copyseq(nseq, Node)
#define emptySEQB(nseq) (nseq->next == NULL)
#define foreachinSEQB(nseq, nptr, nvalue) \
        for(nptr = nseq->next; \
           (nptr != NULL) && \
           ((nvalue =  *(B *)(nptr->data)) || TRUE); \
           nptr = nptr->next)
#define initializeSEQB(nseq) nseq = IDLNewListHeader()
#define inSEQB(nseq, nval) inseq(nseq, (char *)(&nval), Node)
#define ithinSEQB(nseq, index) *(B *)ithseq(nseq, index)
#define sizeSEQB(nseq) sizeseq(nseq)
#define orderedinsertSEQB(nseq, nval, ncompfn) \
        orderedinsertseq(nseq,(char *)(&nval), ncompfn, Node)
#define removeSEQB(nseq, nval) removeseq(nseq,(char *)(&nval), Node)
```

```
#define removefirstSEQB(nseq) removefirstseq(nseq)
#define removelastSEQB(nseq) removelastseq(nseq)
#define retrievefirstSEQB(nseq) *(B *)retrievefirstseq(nseq)
#define retrievelastSEQB(nseq) *(B *)retrievelastseq(nseq)
#define sortSEQB(nseq, ncompfn) sortseq(nseq, ncompfn, Node)
#define tailSEQB(nseq) tailseq(nseq)

#define addSETB(nset, nval) \
        if (!inSEQB(nset, nval)) appendfrontSEQB(nset, nval)
#define copySETB(nset) copySEQB(nset)
#define emptySETB(nset) emptySEQB(nset)
#define foreachinSETB(nset, nptr, nvalue) \
        foreachinSEQB(nset, nptr, nvalue)
#define initializeSETB(nset) initializeSEQB(nset)
#define inSETB(nset, nval) inSEQB(nset, nval)
#define removeSETB(nset, nval) removeSEQB(nset, nval)
#define sizeSETB(nset) sizeSEQB(nset)


#define appendfrontSEQAnode(nseq, nval)  \
        appendfrontseq(nseq,(char *)(&nval), Node)
#define appendrearSEQAnode(nseq, nval) \
        appendrearseq(nseq,(char *)(&nval), Node)
#define copySEQAnode(nseq) (SEQAnode)copyseq(nseq, Node)
#define emptySEQAnode(nseq) (nseq->next == NULL)
#define foreachinSEQAnode(nseq, nptr, nvalue) \
        for(nptr = nseq->next; \
           (nptr != NULL) && \
            ((nvalue =  (Anode)(*(int *)(nptr->data))) || TRUE); \
           nptr = nptr->next)
#define initializeSEQAnode(nseq) nseq = IDLNewListHeader()
#define inSEQAnode(nseq, nval) inseq(nseq, (char *)(&nval), Node)
#define ithinSEQAnode(nseq, index) *(Anode *)ithseq(nseq, index)
#define sizeSEQAnode(nseq) sizeseq(nseq)
#define orderedinsertSEQAnode(nseq, nval, ncompfn) \
        orderedinsertseq(nseq,(char *)(&nval), ncompfn, Node)
#define removeSEQAnode(nseq, nval) removeseq(nseq,(char *)(&nval), Node)
#define removefirstSEQAnode(nseq) removefirstseq(nseq)
#define removelastSEQAnode(nseq) removelastseq(nseq)
#define retrievefirstSEQAnode(nseq) *(Anode *)retrievefirstseq(nseq)
#define retrievelastSEQAnode(nseq) *(Anode *)retrievelastseq(nseq)
#define sortSEQAnode(nseq, ncompfn) sortseq(nseq, ncompfn, Node)
#define tailSEQAnode(nseq) tailseq(nseq)

#define addSETAnode(nset, nval) \
        if (!inSEQAnode(nset, nval)) appendfrontSEQAnode(nset, nval)
#define copySETAnode(nset) copySEQAnode(nset)
#define emptySETAnode(nset) emptySEQAnode(nset)
#define foreachinSETAnode(nset, nptr, nvalue) \
        foreachinSEQAnode(nset, nptr, nvalue)
#define initializeSETAnode(nset) initializeSEQAnode(nset)
#define inSETAnode(nset, nval) inSEQAnode(nset, nval)
```

```
#define removeSETAnode(nset, nval) removeSEQAnode(nset, nval)
#define sizeSETAnode(nset) sizeSEQAnode(nset)
```

```
/* InOut.c */
#include "InOut.h"

extern void StringToCharSet(), CharSetToString();
extern void InitCharSet(), MarkCharSet();
static struct pvt_type_entry SPvtTypeTable[1] =
{
        { StringToCharSet, /* External to Internal */
          CharSetToString, /* Internal to External */
          InitCharSet,     /* Initialization of Internal Rep */
          MarkCharSet,     /* Marking Routine */
          4,               /* Size of Internal Rep in bytes */
          String           /* Type of External Rep */
        }
};

struct pvt_type_entry *PvtTypeTable = SPvtTypeTable;

static struct port_attribute_descriptor StructInTable1[2] =
{
        {0x40, 0},
        {8, 1}
};

static struct port_attribute_descriptor StructInTable2[3] =
{
        {2, 0},
        {0, -3},
        {6, 1}
};

static struct port_table_entry StructInTable[3] =
{
        {0, 2, StructInTable1},
        {0, 0, 0},
        {1, 3, StructInTable2}
};

static struct port_attribute_descriptor StructOutTable1[3] =
{
        {0x40, 0},
        {8, 1},
        {5, 2}
};

static struct port_attribute_descriptor StructOutTable2[1] =
{
        {6, 1}
};

static struct port_table_entry StructOutTable[3] =
{
```

```
        {0, 3, StructOutTable1},
        {0, 0, 0},
        {1, 1, StructOutTable2},
};
static struct invariant_attribute_descriptor InvTable1[3] =
{
        {0x40, 4},
        {8, 8},
        {5, 12}
};

static struct invariant_attribute_descriptor InvTable2[2] =
{
        {2, 4},
        {6, 8}
};
static struct invariant_table_entry SInvTable[2] =
{
        {0, 4, 3, InvTable1},
        {0, 3, 2, InvTable2}
};

struct invariant_table_entry *InvTable = SInvTable;

static struct ascii_attribute_descriptor StructOutAnode[3] =
{ {"List", 0}, {"Name", 1}, {"Number", 2} };
static struct ascii_attribute_descriptor StructOutB[1] =
{ { "R", 1} };
static struct ascii_node_descriptor StructOutNode[2] =
{
        {"Anode", 0, 3, StructOutAnode},
        {"B", 1, 1, StructOutB}
};
static struct ascii_port_table AsciiStructOut = {2, StructOutNode};

static struct ascii_attribute_descriptor StructInAnode[2] =
{ {"List", 0}, {"Name", 1} };
static struct ascii_attribute_descriptor StructInB[2] =
{ { "R", 1}, {"X", 0} };
static struct ascii_node_descriptor StructInNode[2] =
{
        {"Anode", 0, 2, StructInAnode},
        {"B", 1, 2, StructInB}
};
static struct ascii_port_table AsciiStructIn = {2, StructInNode};

extern int *IDLAsciiReadIn();
extern void *IDLAsciiWriteOut();
extern int *IDLBinReadIn();
extern void *IDLBinWriteOut();
extern void IDLMark();
extern void IDLUnMark();
```

```
Anode P(F)
FILE *F;
{
return((Anode)IDLAsciiReadIn(StructInTable, F, &AsciiStructIn));
}

void M(r, ifmark)
{
if (ifmark)
        IDLMark((int *)r, StructInTable);
else
        IDLUnMark((int *)r, StructInTable);
}

void Q(r, F)
Anode r;
FILE *F;
{
IDLBinWriteOut((int *)r, StructOutTable, F);
}

Anode R(F)
FILE *F;
{
return((Anode)IDLBinReadIn(StructOutTable, F));
}

void S(r, F)
Anode r;
FILE *F;
{
IDLAsciiWriteOut((int *)r, StructInTable, F, &AsciiStructIn);
}
```

```
system object, address 0x62df4, type 4, size 3 words Touched
free, address 0x62e00 size 1003 words
node, address 0x63dac, type 0, size 4 words Touched
free, address 0x63dbc size 605 words
node, address 0x64730, type 1, size 3 words
free, address 0x6473c size 1952 words
system object, address 0x665bc, type 7, size 1002 words



The instance read in has been unmarked

free, address 0x27564 size 60832 words
system object, address 0x62be4, type 2, size 4 words
system object, address 0x62bf4, type 0, size 103 words
node, address 0x62d90, type 1, size 3 words
system object, address 0x62d9c, type 3, size 3 words
node, address 0x62da8, type 1, size 3 words
system object, address 0x62db4, type 3, size 3 words
node, address 0x62dc0, type 1, size 3 words
system object, address 0x62dcc, type 3, size 3 words
system object, address 0x62dd8, type 3, size 3 words
free, address 0x62de4 size 4 words
system object, address 0x62df4, type 4, size 3 words
free, address 0x62e00 size 1003 words
node, address 0x63dac, type 0, size 4 words
free, address 0x63dbc size 605 words
node, address 0x64730, type 1, size 3 words
free, address 0x6473c size 1952 words
system object, address 0x665bc, type 7, size 1002 words


All variables have been made null;
everything is garbage collected



Garbage Collected, largest free page size: 64534 words

free, address 0x27564 size 64534 words
system object, address 0x665bc, type 7, size 1002 words
```

```
Anode [List < FirstB^ SecondB^ ThirdB^ SecondB^> ;
        Name "astz"
]
ThirdB : B[X FALSE]
FirstB : B[X TRUE; R  -11/30]
SecondB : B[R 11.3]
#
```

```
L409004 : Anode [
List L404980 : <L404928 : B [
X TRUE;
R 3.14]
L404904 : B [
X TRUE;
R 11.3]
L404880 : B [
X TRUE;
R 0]
L404904^
>;
Name "astz"]
#
```

```
Ascii input has been read in


Garbage Collected, largest free page size: 62582 words

free, address 0x27564 size 62582 words
node, address 0x6473c, type 1, size 3 words
node, address 0x64748, type 1, size 3 words
node, address 0x64754, type 1, size 3 words
free, address 0x64760 size 1007 words
system object, address 0x6571c, type 3, size 3 words
free, address 0x65728 size 4 words
system object, address 0x65738, type 3, size 3 words
free, address 0x65744 size 4 words
system object, address 0x65754, type 3, size 3 words
free, address 0x65760 size 909 words
system object, address 0x66594, type 3, size 3 words
system object, address 0x665a0, type 4, size 3 words
node, address 0x665ac, type 0, size 4 words
system object, address 0x665bc, type 7, size 1002 words


One node has been removed from the sequence of B nodes


Garbage Collected, largest free page size: 62582 words

free, address 0x27564 size 62582 words
node, address 0x6473c, type 1, size 3 words
free, address 0x64748 size 3 words
node, address 0x64754, type 1, size 3 words
free, address 0x64760 size 1007 words
system object, address 0x6571c, type 3, size 3 words
free, address 0x65728 size 4 words
system object, address 0x65738, type 3, size 3 words
free, address 0x65744 size 4 words
system object, address 0x65754, type 3, size 3 words
free, address 0x65760 size 912 words
system object, address 0x665a0, type 4, size 3 words
node, address 0x665ac, type 0, size 4 words
system object, address 0x665bc, type 7, size 1002 words


Binary Output has been done


Garbage Collected, largest free page size: 62576 words

free, address 0x27564 size 62576 words
system object, address 0x64724, type 3, size 3 words
node, address 0x64730, type 1, size 3 words
```

```
node, address 0x6473c, type 1, size 3 words
free, address 0x64748 size 3 words
node, address 0x64754, type 1, size 3 words
free, address 0x64760 size 1007 words
system object, address 0x6571c, type 3, size 3 words
free, address 0x65728 size 4 words
system object, address 0x65738, type 3, size 3 words
free, address 0x65744 size 4 words
system object, address 0x65754, type 3, size 3 words
free, address 0x65760 size 912 words
system object, address 0x665a0, type 4, size 3 words
node, address 0x665ac, type 0, size 4 words
system object, address 0x665bc, type 7, size 1002 words


Binary Input has been done



Garbage Collected, largest free page size: 60939 words

free, address 0x27564 size 60939 words
node, address 0x62d90, type 1, size 3 words
system object, address 0x62d9c, type 3, size 3 words
node, address 0x62da8, type 1, size 3 words
system object, address 0x62db4, type 3, size 3 words
node, address 0x62dc0, type 1, size 3 words
system object, address 0x62dcc, type 3, size 3 words
system object, address 0x62dd8, type 3, size 3 words
free, address 0x62de4 size 4 words
system object, address 0x62df4, type 4, size 3 words
free, address 0x62e00 size 1003 words
node, address 0x63dac, type 0, size 4 words
free, address 0x63dbc size 605 words
node, address 0x64730, type 1, size 3 words
free, address 0x6473c size 1952 words
system object, address 0x665bc, type 7, size 1002 words


Ascii Output has been done



The instance read in has been marked

free, address 0x27564 size 60832 words
system object, address 0x62be4, type 2, size 4 words
system object, address 0x62bf4, type 0, size 103 words
node, address 0x62d90, type 1, size 3 words Touched
system object, address 0x62d9c, type 3, size 3 words Touched
node, address 0x62da8, type 1, size 3 words Shared
system object, address 0x62db4, type 3, size 3 words Touched
node, address 0x62dc0, type 1, size 3 words Touched
system object, address 0x62dcc, type 3, size 3 words Touched
system object, address 0x62dd8, type 3, size 3 words Touched
free, address 0x62de4 size 4 words
```