An Introduction to IDL

David Alex Lamb

November 1988
External Technical Report
ISSN-0836-0227-
1988-237

Department of Computing and Information Science
Queen's University
Kingston, Ontario K7L 3N6

Version 1.2
Document prepared June 12, 1997

Abstract

This report contains some of my recent writings about IDL, including a tutorial introduction, a description of possible *target models* for IDL, some possible extensions to IDL, and a list of suggested standard *implementation notes*.

Contents

List of Figures

Chapter 1

Introduction

IDL (Interface Description Language) is a notation for describing programs and the data structures by which they communicate. Since 1980, IDL has formed a significant fraction of my research work. This report contains several chapters that represent recent work I have done describing various portions of IDL; as such they may seem somewhat disconnected. I expect to reuse portions of this report in later IDL-related work, especially the PCG project [Lamb87b].

Chapter 2 gives a tutorial introduction to portions of IDL. For full details of the language, you should refer to the version 2 language definition [Nest82]. After version 2, IDL has split into several dialects, but the early sections of the tutorial apply to them all. Chapter 4 describes my current thinking about the Queen's dialect of IDL.

The thing about IDL that I have had the hardest time convincing people of is the wide variety of implementation strategies it can support. Chapter 3 describes many possible approaches to defining *target models* for implementing IDL; I have written earlier papers on the same theme [Lamb85,Lamb87d, Lamb87c]. Appendix B summarizes some standard implementation strategies IDL implementations might consider supporting.

Margaret Lamb, John Nestor, Joe Newcomer, and Don Stone make helpful comments on earlier drafts of this material.

Chapter 2

Using IDL: A Tutorial

This chapter introduces you to the basic concepts of IDL, including how to interact with it in conventional programming languages.

2.1 Why Use IDL?

A significant fraction of IDL consists of notation for describing data structures. A reasonable question is, why should you use IDL rather than the type declaration mechanisms of your favorite programming language? Three of the important reasons are:

1. IDL can describe complex data structures in a simple but formal way. It gives you a way to talk about the information content of your data separately from its representation, much more clearly than most programming languages do.

2. IDL tools can automatically generate stereotypical modules of a program, such as input/output procedures.

3. The automatically-generated modules support passing data between programs that might be written in different programming languages, and might run on different machines.

The third point was our main original motivation for designing IDL. At this level, the only communication we care to support is exchanging data structures. We want each program to be able to work on an internal representation of its data that is appropriate for its processing needs. Moreover, we might want to write each program in a different implementation language, and may want to run each of them on a different type of machine. On the other hand, we might want to design two programs separately for convenience, then combine them into a single program for efficiency.

These requirements dictate some of the design decisions in IDL. We must have some way to talk about individual programs, and group basic programs into composite ones. We must be able to describe the abstract information-content properties of data structures (the ones all programs can depend on) separately from the concrete representational properties (the ones that change from program to program).

2.2 Basic Concepts

A *structure* is an IDL declaration of a data structure. You can view it as a schema that describes a collection of object types; the objects form a graph (technically, a typed attributed directed graph). This section discusses IDL's facilities for describing and manipulating graphs.

2.2.1 Names

IDL identifiers start with a letter and consist of a sequence of letters, digits, and underscores. IDL ignores case distinctions between letters; the identifier `NAME` means the same thing as `NaMe` or `name`. Underscores *do* make identifiers different; `name_1` is not the same as `name1`. For readability, you should avoid using consecutive underscores in an identifier; it is hard to tell `some_name` from `some__name`.

IDL reserves several identifiers for its own use, typically to define language constructs.

2.2.2 Node Types

The basic units from which you build graphs are *nodes*, which are analogous to records in typical programming languages. Associated with each node you may have several *attributes*, which you can view as information about the node; they are analogous to record fields. In IDL, what you declare is the node type; the IDL notation does not include the notion of variables. Thus one might implement the IDL declaration

```
item => name : string,
        value : integer;
```

with the Algol-like type declaration

```
type item_rep = pointer to record
    name : string_rep; value : integer_rep
    end
```

We use the "`_rep`" suffix to distinguish the implementation language's representation type from the IDL type. The type is a pointer to a record rather than simply a record because we will eventually be building trees and graphs from such objects.

In a program that uses IDL data, you would declare variables of this type and manipulate their attributes. For example,

```
var a, b : item_rep;
...
a.value := b.value+5;
```

shows how you might declare two objects and manipulate their `value` attributes. The exact syntax for accessing attributes would vary from language to language. In C you might write

```
a->value = b->value+5;
```

whereas in Pascal you might write

```
a ^.value := b ^.value+5;
```

These issues are part of what we call the *target model*: the description of how you map IDL concepts into different programming languages. We will see in Section 3.2 that some target models would require you to write

```
Set_value(a, Fetch_value(b)+5);
```

The point is that there is a simple concept, such as attribute access, and several different ways of representing that concept in different implementation languages.

If you want to build interesting data structures, you need several different types of nodes. For example, to the previous example you could add

```
date => month      : integer,
        day        : integer,
        year       : integer;
item => start_date : date,
        end_date   : date;
```

This says that `item` nodes have two `date`-valued fields, and that `date` nodes have three `integer`-valued fields. You might access this information by saying

```
if a.start_date.year = b.end_date.year then ...
```

This example also shows that IDL lets you declare the attributes of a node in separate declarations if you choose to do so. Thus

```
item => start_date : date,
        end_date : date;
```

means the same thing as

```
item => start_date : date;
item => end_date : date;
```

Furthermore, you do not need to declare something before you use it, so you could declare the `date`-valued attributes before you defined type `date`. This is a general characteristic of most IDL declarations; their order does not matter, and you can define several things in one declaration or split them into several declarations.

Remember that we defined the `item_rep` type as a pointer; similarly, `date_rep` would be a pointer. Thus, `a.start_date` points to a `date` object that is separate from `a`. Thus, perhaps somewhat surprisingly, we have created a simple graph, since `item` nodes point to `date` nodes; we will see in Section 3.2.2 how to avoid this by embedding the dates in the items.

2.2.3 Classes

If we want to construct interesting graphs, we need a way to say that an attribute can reference more than one type of node. The IDL way to do this is to define a *strict class* type. Thus

```
expression ::= binary | literal;
binary => left : expression, right : expression;
literal => symbol : string;
```

The first line says `expression` is a strict class that contains classes `binary` and `literal`. The `left` and `right` attributes of `binary` nodes form a graph by pointing to other `expression` (that is, `binary` or `literal`) nodes.

At first glance, strict classes seem similar to union types in languages such as C or Algol-68. However, IDL classes give you more expressive power than ordinary unions; they provide an *inheritance mechanism* similar to Simula classes or Smalltalk objects. You can declare

```
expression => value : integer;
```

This says that all `expression` nodes have an integer-valued `value` attribute. Thus every `literal` node or `binary` node will have such an attribute; they inherit it from their parent, `expression`.

The term *class* includes both node types and strict class types. Thus `expression`, `binary`, and `literal` are all classes, but only `expression` is a strict class. We use this terminology to avoid having to say “strict class or node type” in several places where we need to refer to both; we can say “class” instead. Thus we can describe the two forms of definition we have seen so far as “a class name, followed by `=>`, followed by a sequence of attribute definitions separated by commas,” and “a strict class name, followed by `::=`, followed by a sequence of class names separated by vertical bars.”

Ideally, a target model will provide types and operations that match the IDL graph’s meaning as closely as possible. Thus you should be able to declare objects of implementation-language types corresponding to `expression`, `binary`, and `literal`; you should also be able to fetch and store appropriate attributes of each type. Thus after declaring

```
var t : expression_rep;
    i : binary_rep;
    l : literal_rep;
```

in your program you should be able to say

```
t.value := i.left.value + string_length(l.symbol);
```

This particular computation may not make much sense itself; we use it only to show the attribute accesses

2.2.4 Graphs

Conceptually, attributes are pointers to objects; forming graphs is easy. For example, if we wanted to add a cross-reference from each literal node to the last node with the same `symbol` field, we could do this by saying

```
literal => previous : literal_void;
literal_void => literal | void;
```

We will discuss `void` in Section 2.4.3; for now, interpret this declaration as saying that `previous` attributes point to `literal` nodes or to nothing.

The idea of attributes as pointers may surprise you somewhat. When we say

```
binary => left : expression,    right : expression;
```

there is no particular reason the `left` attribute of an `binary` node could not point at the node itself, forming a cyclic graph instead of a tree. Thus the notation encourages general graph structures; to restrict the graph to a tree, you would need to use the assertion language, which we introduce in Section 2.4.4.

2.2.5 Attribute Types

We have seen examples of integer-valued attributes and class-valued attributes. Attributes can also be booleans, strings, or rationals. Rationals include all the fixed point, floating point, and "real" numbers computers support. We call these four types, plus classes, the *basic types*.

Attributes can also be composite types: sets or sequences of some basic type. Thus you can say

```
dictionary => symbols : set of string,
call =>       actuals : seq of expression;
```

Sets are unordered collections; sequences are ordered collections. As with all other IDL types, there are many possible representations. For example, we might implement a sequence as an array, a singly linked list, or a doubly-linked list.

Sometimes you will want to use a type that IDL knows nothing about. To do this, you can declare a *private type*, and tell the IDL translator (using *implementation notes*; see Section 2.4.1) where to find an implementation for the type. Thus, for example, you could declare a stack type, `MyStack`, by saying

```
    type MyStack;
    for MyStack use Package(myPackage,repType);
```

IDL does not have a special mechanism for creating enumeration types; instead, the strict class mechanism serves that purpose. If a class consists entirely of node types without attributes, it behaves much like an enumeration type. For example,

```
    binary => op : operation;
    operation ::= plus | minus | times | divide;
      plus =>; minus =>; times =>; divide =>;
```

defines class `operation`, which serves as the type of the "operation code" field of binary expressions.

2.2.6 Structures and Instances

A *structure* is an IDL declaration of a data structure. You can view it as a schema that describes a collection of object types. An *instance* of an IDL structure is a collection of objects whose types and relationships match those of the corresponding structure declaration. In general, the objects form a directed graph, which might contain cycles.

The general form of a simple structure declaration (in the extended Backus-Naur form of Appendix A) is

```
structure <name>1 {root <name>2 }? is
    { <structure item> # ; }*
end
```

This is a simplification; the IDL reference manual [Nest82] gives the full syntax. `<name>`$_1$ is the name of the structure; depending on the implementation model you choose, it may also become the name of the module that implements the structure. The nonterminal `<structure item>` includes the declarations we have already seen, as well as items we will see in later sections.

`<name>`$_2$ is the name of the *root type* for your structure. It must name some type you define within the structure, and will usually be a class type or a sequence type. IDL defines the "contents" of an instance of your data structure as all the objects you can reach by following attributes starting at the root. This is not a restrictive requirement; most data structures such as parse trees and flow graphs already have a well-defined root. If not, the designer can add a dictionary node that points to the roots of all the disjoint subgraphs.

Figure 2.1 shows an example of an IDL structure. Previous sections explained the details of this example. Figure 2.2 shows one possible instance of

```
structure TreeStruc root expression is
   expression ::= call | binary | literal;
   binary => left: expression,
             right:expression;
   binary ::= plus | minus | times | divide;
     plus =>; minus =>; times =>; divide =>;
   call => name :    string,
           actuals : seq of expression;
   literal => symbol:string;
   expression => value:integer;
end
```

Figure 2.1: Expression Structure Example

the structure from Figure 2.1. There are two cases of sharing in this diagram:

- `root.value` is the same integer-valued object as `root.actuals[1].value`,
- `root.actuals[1].right.value` contains the value `23`, and is the same integer-valued object as `root.actuals[3].actuals[1].value`.

This illustrates that IDL permits sharing of any type of object, even scalars.[1]

2.2.7 Membership, Narrowing, and Widening

If you write a program that manipulates trees, you will sometimes need to inspect a tree node and determine whether it is a `literal` or a `binary` or a `call` node. Depending on the target model, there may be several ways of doing this.

A node object's *kind* is a tag corresponding to its node type. Thus corresponding to node types `call` and `literal` would be tags `call` and `literal`; most implementation languages would require that you give the tags different names, such as `call_kind` and `literal_kind`. An implementation would give you some way of inspecting a class object's kind; for example, a `NodeKind` operation might returns its parameter's kind. Thus

[1]You can avoid sharing by embedding an unsharable object in the object that references it. Many target models do this with scalars by default.

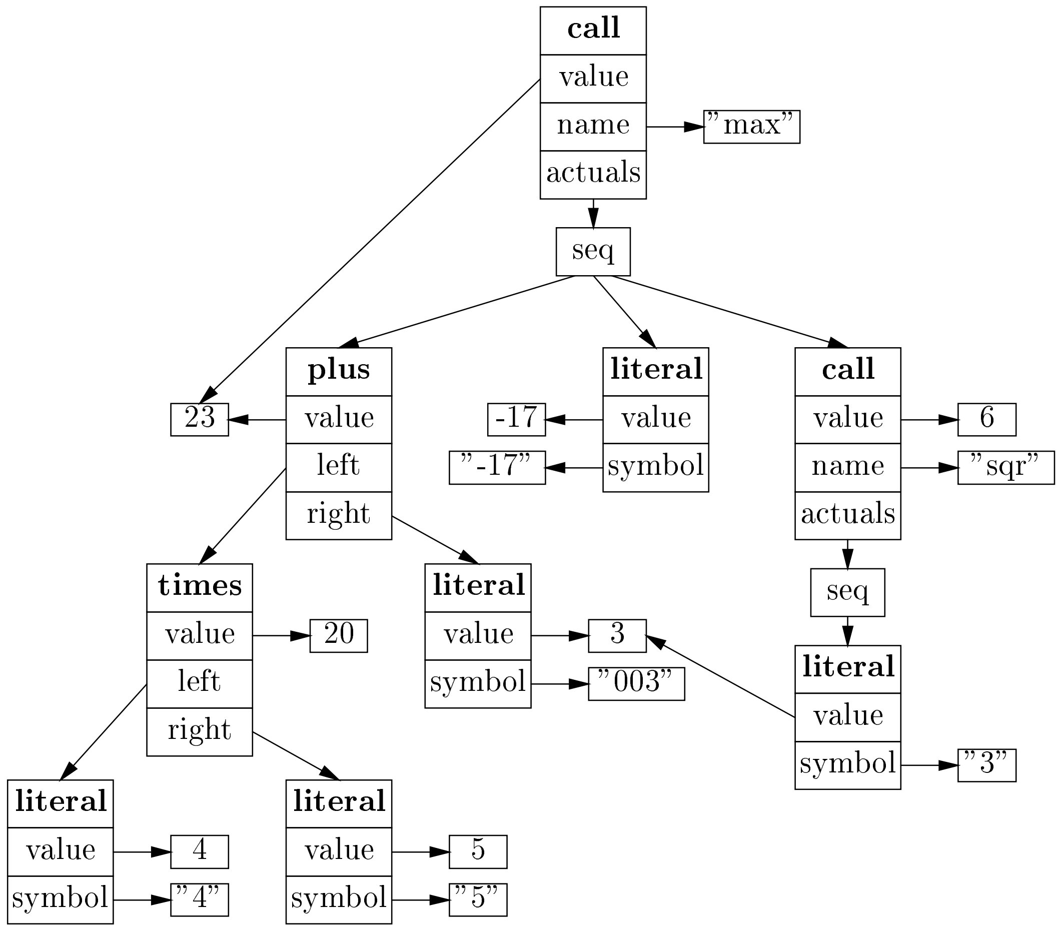


Figure 2.2: Picture of an Instance of the Structure of Figure 2.1

```
var t : expression_rep;
...
if NodeKind(t) = literal_kind then ...
```

might test whether a tree node is a literal.

An alternative way of packaging this test is for the target model to provide boolean operations that test class membership. Thus another way of doing the same test is

```
if Is_literal(t) then ...
```

A third alternative is for the target model to provide a generic operation that takes an object and a class and reports whether the object is a member of the class.

```
if IsMember(t,literal) then ...
if IsMember(t,binary) then ...
```

If we know that an object is a member of a particular class, we may wish to treat it as an object of a containing class. For example, given a `literal`, we may wish to pass it to a procedure that takes an `expression` as a parameter. Since many implementation languages are strongly type checked, this may require a *widening* operation to convince the language translator to treat the object as the more general type. For example, in

```
procedure Grind(t:expression_rep) is ... end Grind;
...
var i : binary_rep;
...
GrindTree(binary_to_expression(i));
```

the function `binary_to_expression` is a widening operation. Ideally, widening is purely a compile-time operation.

Similarly, we may wish to treat a variable of one class as if it were a member of a contained class; this may require a *narrowing* operation. Unlike widening, narrowing typically requires a run-time check that the object has a kind corresponding to the narrower type. Thus in

```
var i:binary_rep;
...
print(expression_to_literal(i.left).value)
```

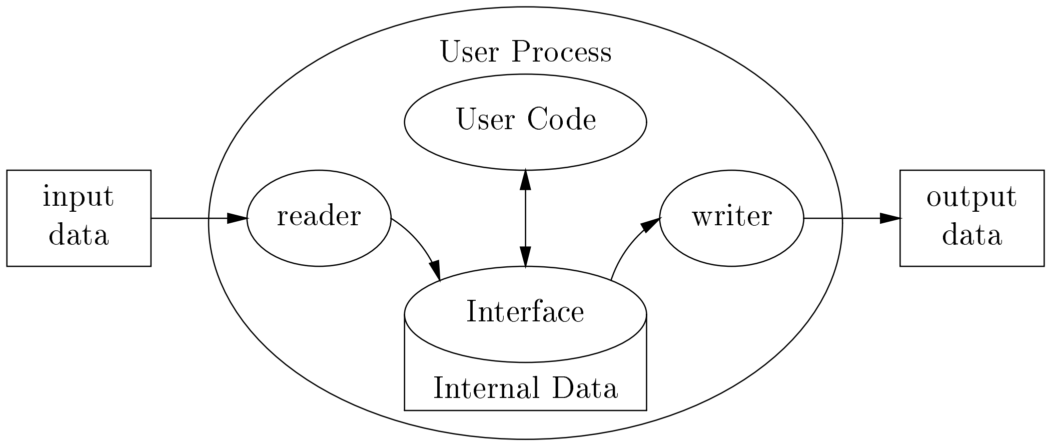
the function `expression_to_literal` is a narrowing operation. It checks at run-time that `i.left` is of kind `literal`, and reports an error otherwise.

Target models can ensure that both narrowing and widening are perfectly type-safe; they are *coercions*, which convert a value of one type to another compatible type, rather than *type casts*, which tell the compiler to pretend that a value of one type is of another type.

2.3 Generated Modules

Earlier, we said a benefit of using IDL was automatic generation of stereotypical modules of your program, particularly graph readers and writers. This section discusses some of the modules the IDL system can generate for you.

It is easy to show why automating the coding of readers and writers is useful. Internally, most programs represent references as pointers to locations in memory. In an external file, the pointers must have some different representation, such as offsets from the beginning of the file, or textual labels and label references. The code to walk over a graph, converting pointers and emitting a representation of each node, is tedious and error-prone to write by hand.



Arrows indicate direction of data flow.

Figure 2.3: Typical IDL-Based Program

2.3.1 Program Organization

Figure 2.3 shows the organization of a simple IDL-based program. It reads input data in some external representation, maniupulates it in some internal representation, and writes the altered data in another external representation. We can view the job of a *reader* module as converting an external instance into an equivalent internal instance; a *writer* module does the reverse coversion. We normally talk of readers and writers that convert an entire structure at once, although some implementations may also provide incremental conversions. A third module, the *interface*, provides the means for user code to access internal data. In effect, the interface defines an abstract data type; the internal form is a representation.

2.3.2 IDL Translator Organization

The primary new tool you would use in an IDL system is an *IDL translator*, which can automatically generate readers, writers, and interface from an IDL description. Figure 2.4 shows the general behavior of such a translator. A user writes an IDL description and the code for a program (dotted boxes). The translator analyses the description to produce

- a listing with summary information and cross-references, to assist in maintaining the IDL description;

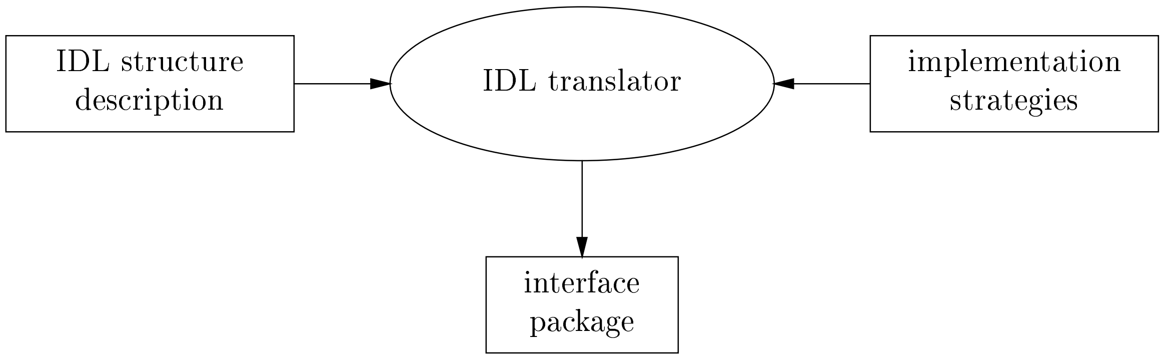


Figure 2.4: IDL Translator Operation

- the specification of an interface module, for use in compiling the user's code;
- the implementation of the interface module, for use in the running system;
- tables and code fragments that, when combined with information from a library, give the readers and writers that assist in exchanging data with other tools.

In emitting a module, the translator may make use of a library of information about implementation strategies. Some parts of this "library" may be part of the code of the translator, other parts might reside in tables, while still others might be part of a separate file.

Chapter 3 discusses the operation of IDL translators in more detail, and discusses issues one must consider in designing interface, reader, and writer modules.

2.3.3 Processes

A *process* is IDL's notion of an executable program or fragment of a program. It places few constraints on the program; it merely deals with IDL-related aspects of the program, such as what data structures it manipulates. The IDL V2.0 processes were not particularly useful; users at Tartan complained about having to include an otherwise useless process declaration just to get any output. Nestor has designed one replacement, and I am contemplating an alternative replacement.

The key things that a process declaration must specify are:

- A name for the process.
- What structures the process reads and writes.
- What structures the process uses internally.
- If the process contains subprocesses, how the processes communicate with each other.

2.3.4 Exchange Representations

In order for two processes to exchange data, they must agree on the representation of the data. If the two run on machines with different word sizes and data formats, they must agree on some representation that both of them can interpret. We call such a data representation an *exchange representation*. It is the task of a writer to translate its process' internal representation into an exchange representation; it is the task of a reader to do the inverse transformation.

The amount of work a reader or writer must do depends on how similarly the two processes represent data. If the two programs have identical representations and run in the same executable image (say, as overlays), no work is necessary. If they have nearly identical representations but are in separate programs, they might use a binary representation. If they run on different types of machine, they may need the flexibility of an ASCII representation. Ideally, the exchange representation schema should be readily derivable from the IDL description.

In other parts of the report we sometimes use an ASCII representation to show particular data structures; although there are many possible external representations, we have defined this one as part of IDL to ensure there is one representation all implementations can depend on. The representation of a structure consists of a reference to the root node followed by a sequence of labeled nodes. The representation of a node consists of the name of the node type, followed by list of attribute-value pairs. Any value may have a label; all labels must be unique.

Figure 2.5 shows how to represent formally the structure instance that Figure 2.2 on page 10 represented informally. Figure 2.5(a) shows each object in a "flat" list. The external ASCII form permits some variability; Figure 2.5(b) represents the same instance using a "nested" form. In either form, the writer can add extra white space between tokens to improve readability. Whether to use a flat or nested form depends on the way in which you will use the external representation, and how large the structure is. The nested form seems more natural for small tree-like structures, but for deep graphs the white space used to line up attributes of the same node rapidly gets out of hand. The flat form uses less white space, and some systems can choose label names to make it

```
L0:  call [value L1: 23; name "max"; parameters <L3 ^ L4 ^ L5 ^>]
L3:  plus [value L1 ^; left L6 ^; right L9 ^]
L4:  literal[value -17; symbol "-17"]
L5:  call [value 6; name "sqr"; parameters <L10 ^>]
L6:  times [value 20; left L7 ^; right L8 ^]
L7:  literal[value 4; symbol "4"]
L8:  literal[value 5; symbol "5"]
L9:  literal[value L2: 3; symbol "003"] ]
L10: literal[value L2 ^; symbol "3"]
```

(a) Flat Form

```
call [value L1: 23; name "max"; parameters <
      plus [value L1 ^;
            left times [value 20;
                        left literal[value 4; symbol "4"];
                        right literal[value 5; symbol "5"] ];
            right literal[value L2: 3; symbol "003"] ]
     literal[value -17; symbol "-17"]
     call [value 6; name "sqr"; parameters <
           literal[value L2 ^; symbol "3"] >]
     > ]
```

(b) Nested Form

Figure 2.5: Sample External Forms

easier for a human being to find individual nodes, but processing the labels can slow down a reader considerably.

2.4 Advanced Topics

This section discusses several advanced topics that you should study if you want a more complete understanding of IDL and the software engineering problems it addresses.

2.4.1 Implementation Notes

We designed IDL to support abstraction, with a single type implementable in many different possible ways. For example, the type integer might require a machine word, or or a variable size block capable of representing arbitrarily large integers. Similarly, sequences might be fixed size arrays or variable size arrays or linked lists. In the absence of any user action, each IDL translator applies its own set of default rules to select a specific representation. Users can also gain more explicit control by using *implementation notes.*

The syntax of an implementation note is

```
<structure item> ::= <implementation statement>;
<implementation statement> ::=
        { for <implementation reference> }?
        use <implementation note>;

<implementation note> ::= <name> { ( { <note parameter> #
, }+ ) }?;
<note parameter> ::= <implementation note> | <literal> ;
```

Whether you omit the `<implementation reference>` clause depends on the particular implementation note.

- If you omit the implementation reference, it usually means you are saying something about the entire structure. For example, in the Queen's implementation,

  ```
  use Filename("string")
  ```

  means that you should use the name *string* for any files the translator emits for this structure.

- A note with a name reference usually applies to a named type and all attributes of the type; thus

  ```
  operation ::= plus | minus | times | divide;
  for operation use Enumeration;
  ```

  says that type `operation` (a strict class type) should have an enumeration type as its representation.

- A note with an attribute reference applies to a particular attribute of a particular node; thus

  ```
  for expression.value use Range(0,255)
  ```

  says that the `value` attribute of `expression` nodes (an integer attribute) should hold numbers in the range 0 to 255.

There are three forms of representation reference.

```
<implementation reference> ::= { 'root' | <name> { '.'
<name> } }
        { '(' '*' ')' }?;
```

The `<name>` refers to a named type; this was the form we used in the preceding example. The form `<name>`$_1$.`<name>`$_2$ refers to an attribute; `<name>`$_1$ is a class type, and `<name>`$_2$ is one of its attributes. The **root** form refers to the root type of the structure. The optional asterisk for these forms gives a way to specify a representation for the elements of a composite type (a set or sequence type). Thus

```
call => actuals : seq of expression;
for call.actuals use Array;
for call.actuals(*) use Size(16);
```

says that the `actuals` attribute of objects of type `call` should be an array of 16-bit values (presumably short pointers).

Some properties of a data structure that you think of as abstract properties are really implementation details. For example, we know of one project that commonly had declarations like

```
call => head : expression_void, tail : expression_void;
expression => next : expression_void;
expression_void ::= expression | void;
```

What the writers intended was to have a sequence of `expression` elements in each `call`. The `head` and `tail` attributes of `call` nodes reference the first and last elements of the sequence; the `next` attribute in each `expression` node links it with the next element in the list. A better way to reflect what is really going on is to declare a sequence, and add an implementation note. This would result in

```
for call.actuals use Threaded(next);
```

2.4.2 Derivation

Derivation is the process of saying that one structure is similar to another. For example, from the structure defining expressions, we could derive two others, one of which represented the sequence of actual parameters in a `call` node as an array, and one of which represented it as a linked list. From a complex structure we could derive several others that give narrower interfaces to the same data, eliminating certain attributes or types of nodes. From distinct structures we could derive a single structure that combined them all.

Ideally, the derivation mechanism should be able to support different structures giving different views of the same data. For example, there are two ways to define binary expressions. Earlier, we showed one way:

```
    binary => left : expression, right : expression;binary => op : operation;
    operation ::= plus | minus | times | divide;
      plus =>; minus =>; times =>; divide =>;
```

Alternatively, we could make `binary` a strict class:

```
    binary => left : expression, right : expression;binary ::= operation;
    operation ::= plus | minus | times | divide;
      plus =>; minus =>; times =>; divide =>;
```

It is possible to think of these as two different representations of the same information.

IDL V2 had two mechanisms: derivation and refinement. Derivation was more general than refinement; anything you could do with refinement you could also do with derivation. The only reason for having refinement was if some uses of derivation were easier if you could guarantee that you only the refinement subset. I plan to implement only derivation in the Queen's subset, until I find some circumstance that demands refinement instead.

2.4.3 Void

It is common to have class-valued attributes that reference either a member of a particular class or nothing. You need to be able to say that the attribute could reference nothing. However, IDL does not define the concept of "pointing to nothing"; every attribute must reference an object of some appropriate type.

The way we deal with "pointing to nothing" is to define a class called `void` that has no attributes. The declarations

```
     expression => parent : expression_void;
     expression_void ::= expression | void;
```

say that `expression.parent` might reference nothing (for example, the root node might not have a parent). This is cleaner than the alternative, which is to say that `expression.parent` is a set or sequence of `expression` nodes with either 0 or 1 elements. `void` is part of a "standard prelude" from which all structures inherit predefined types like `integer` and `boolean`. Other than this, there is nothing special about the name `void`; if we replaced all occurrences of `void` with `fred`, the meaning would be the same.

2.4.4 Assertions

The declarations we have discussed so far do not capture all the abstract properties we might want to specify for structures and processes. IDL provides a powerful *assertion* sublanguage for declaring such properties.

An assertion consists of the reserved word **assert** followed by a boolean expression. If an assertion occurs inside a structure, then the boolean expression must be true for any instance of the structure. Assertions are not necessarily true of a structure instance at all times. They must certainly be true whenever someone claims to have a complete instance, such as just after reading one in or just before writing one out. However, while user code is manipulating a structure, it need not meet its assertions.

The IDL V2 assertion language had several problems, not the least of which is that almost no one (besides Snodgrass' group at the University North Carolina) ever used it. Nestor has defined a revision to the assertion language; however, until we have more use of the language, it is hard to say whether we understand it well enough to stabilize it.

I expect to expand on the assertion language tutorial as I decide what subset of the assertion language to implement at Queen's.

Chapter 3

Generating Interface Packages

Chapter 2 introduced the basic ideas of IDL, and mentioned a few ways of implementing some of them. This chapter gives much more detail on modules that give access to the data structures you can define with IDL, and exchange data among programs. It touches on some general characteristics of IDL translators, concentrates on what issues you must address in designing modules, and shows how to use libraries of predefined implementation strategies for efficient data structure representations.

3.1 Target Models

A *target model* is the set of functions, notations, and conventions used to construct the user code interface for all structures. The IDL translator constructs a particular user code interface by applying the target model to a particular structure specification. Normally, there is a different target model for each different programming language. Some target models might support several languages, if a single program has portions in each of them that need to access the same data. Different implementations often use different target models for the same language.

In designing a target model, you must make a set of interacting (and sometimes conflicting) engineering decisions. You must ask yourself several questions.

- **Representation independence**. To what degree do you wish to protect users of an interface module from changes in the representation of the package? A high degree of protection means that you need not edit clients of a module when the representation changes. If you want little protection, your interface module will typically make visible the underlying representations. If you want much protection, your interface module will require users to use access procedures to manipulate the representations. Your implementation language's degree of support for information hiding will influence this decision.

- **Type model**. To what extent do you want to mimic the type structure of the IDL description? Close mimicry means your language's type checking mechanisms can help check whether your program uses the IDL types correctly. If you want to mimic it closely, your package must export implementation language types for each IDL node type, class type, private type, and attribute type in the IDL description. If you do not want to mimic it closely, you might have one implementation language type for the entire IDL structure, or perhaps one for each of a few classes within the structure. The closeness of your implementation language's type model to that of IDL, and the extent of its abstraction facilities, will influence this decision.

- **Attribute access**. Given a programming language representation of a node, how efficiently can you fetch or store an attribute of the node? Your implementation language's facilities for specifying low-level representational details will influence this decision.

- **Development support**. To what extent must you recompile clients of a module when its specification changes, even if the parts of the interface the clients depend on do not change? This is a different issue from representation independence; it involves adding, changing, or deleting attributes, nodes, or classes. Your implementation language's facilities for separate compilation, its compiler's facilities for recompilation analysis, and the effort you are willing to put into recompilation support in an IDL translator, will influence this decision.

- **Generic tool support**. To what extent must the modules support generic or table-driven tools defined outside the package? A generic tool is one that embodies an algorithm that applies to any structure; it relies on structure-specific information (typically in tables) in the interface module. Often a generic tool is slower than one "hard-coded" for a particular structure. Your implementation language's facilities for generic operations (or type-cheating mechanisms for implementing such operations) might influence this decision.

Most of the following sections show how to use an IDL specification, and that from an implementation strategy library, to generate interface modules (specification and implementation). Section 3.7 discusses the design of reader and writer modules.

3.2 Reference Representations

Attributes of nodes are like fields of records, provided you generalize the standard mental model of records as contiguous blocks of memory and fields as offsets into such blocks. In the IDL formal model, attributes are always pointers to objects. Once you break away from both of these mind sets there are two issues with any attribute.

1. How do you represent the object the attribute references? We refer to these as *object representations.*
2. How do you represent the attribute itself? We refer to these as *reference representations.*

This section addresses the second question; later sections address the first.

3.2.1 Access Procedures

If you accept that you want to hide representational decisions, you must design access procedures that let clients fetch and store attributes without revealing either the representation of the attribute or of the object the attribute references. An IDL attribute declaration of the form

```
ClassName => AttrName : AttrType;
```

corresponds to a pair of fetch/store procedures; in Pascal they might take the form

```
function Get_ClassName_AttrName(x:ClassName) : AttrType;
procedure Set_ClassName_AttrName(var x:ClassName; a:AttrType);
```

The first fetches the current value of the attribute; the second stores a new value. Figure 3.1 shows the corresponding Pascal procedure bodies if one adopts the obvious record representation for the class. Since the bodies of these procedures are so simple, a designer would probably wish to declare that the compiler should expand calls on the subprograms inline. Standard Pascal gives no way to do this, but in Ada you could say

```
pragma inline(Get_ClassName_AttrName, Set_ClassName_AttrName);
```

In Ada you could rely on overloading to disambiguate procedures when different classes have attributes of the same name. This would give shorter procedure names, of the form

```
function Get_AttrName(x:ClassName) return AttrType;
procedure Set_AttrName(in out x:ClassName; a:AttrType);
```

With such a simple representation, many people wonder why one must go through all the complexity of having a pair of procedures to represent an attribute. Where before users might have said

```
x.AttrName := y.AttrName
```

```
type Classname = ^  { typically a pointer type }
    record ... AttrName : AttrType ... end;

function Get_ClassName_AttrName(x:ClassName) : AttrType;
    begin
        Get_ClassName_AttrName := x.AttrName;
    end { Get_ClassName_AttrName };

procedure Set_ClassName_AttrName(var x:ClassName; a:AttrType);
    begin
        x.AttrName := a;
    end { Set_ClassName_AttrName };
```

Figure 3.1: Standard Attribute Representation in Pascal

they must now say

```
Set_ClassName_AttrName(x, Get_ClassName_AttrName(y))
```

Figure 3.2 gives an alternative representation that shows why we need the procedures. Here, the attribute almost always has a particular value. To save space, we use the address of the record representing the IDL node as a key for a hash table. If the key is in the table, the table also holds the value of the attribute. If the key is not in the table, the value of the attribute is the default. This may be a suitable representation for the DIANA `lx_comments` attribute, which is almost always null [Goos83].

If a designer initially uses the representation of Figure 3.1 and makes available the fields of the record type, programmers will directly assign to components of the record. If space performance measurements later show that the representation of Figure 3.2 is appropriate, converting the code will require massive re-editing. However, if client modules only use the Set and Get procedures, maintainers need only recompile them after changing the single module representing the `ClassName` type.

Even languages that allow user-defined assignment might not give enough support to allow

```
x.AttrName1 := y.AttrName2;
```

The hashing representation requires that you combine selecting a destination attribute (the dot "operator") with assigning to the attribute (the assignment operator); some proposals for user-defined assignment separate the two in such

```
{ references package Hash, which provides operations
  Lookup, Delete, Enter }
...
type ClassName = ... ;
function DefaultVal : AttrType; { Pascal idiom for a run-time constant }
    begin DefaultVal := ... end;

function Get_AttrName(x:ClassName) : AttrType;
    var val : AttrType;  IsPresent:boolean;
    begin
        Lookup(HashTable, x, val, IsPresent);
        if IsPresent
            then Get_AttrName := val
            else Get_AttrName := DefaultVal
    end { Get_AttrName };

procedure Set_AttrName(var x:ClassName; a:AttrType);
    begin
        if a=DefaultVal
            then Delete(HashTable, x)
            else Enter(HashTable, x, a)
    end { Set_AttrName };
```

Figure 3.2: Hash Table Attribute Representation in Pascal

a way that you never have enough information in one place to implement the hashing scheme. An alternative is to use a preprocessor that translates

```
x.AttrName1 := y.AttrName2
```

into

```
Set_AttrName1(x,Get_AttrName2(y))
```

where appropriate; **?CS?** propose this approach with their C implementation of IDL.

There are other possible ways of packaging the fetch/store procedure pairs. For example, for structures where classes never (or rarely) have multiple attributes of the same type, the name of the function might reflect the attribute type rather than the attribute name.

```
function Get_AttrType(x:ClassName) : AttrType;
```

If one also represents all IDL node types by a single implementation language type (see Section 3.3), this scheme requires fewer interface procedures than the previous one (the size of some styles of interface module strains some compilers). Alternatively, one might have a single generic fetch function

```
function GetAttribute(x:Universal, class:ClassName,
                      attr:AttrName) : Universal
```

where `Universal` is some type capable of representing any IDL object, and `ClassName` and `AttrName` are strings or enumerated types naming the desired class and attribute. This method requires run-time consistency checks to ensure the class really has such an attribute. In a language like Ada you could get a bit more type checking by overloading on the return type, and specializing the first argument to represent just the node types of the structure:

```
function GetAttribute(x:Structure_Rep, class:ClassName,
                      attr:AttrName) return AttrType;
```

The GRAPHITE project at the University of Massachusetts uses this form of attribute access [Clar86].

3.2.2 Implementation Strategies

There are at least five ways of implementing an attribute reference (that is, of answering question 2 on page 23). In the following paragraphs, "the node" means the node containing the attribute, and "the object" means the object referenced by the attribute.

1. The attribute may be a pointer. This representation is closest to the general IDL model, and is the obvious way to implement class-valued attributes. One might say
   ```
   for node.attr use pointer;
   ```

2. One might embed the object in the node. This is the obvious way of implementing unshared data, such as most integer-valued or boolean-valued attributes. One might say
   ```
   for node.attr use embed;
   ```

3. One might keep the object in a hash table, where the node's address is a hash key. Thus the attribute requires no space in the node. This might be suitable for attributes that almost always have some default value. A hash implementation note might need several parameters, specifying open versus closed hashing, the size of the hash table, the method of rehashing on collisions, the default value, and so on.

4. One might keep the object in a table, where the attribute is the index into the table. This might be suitable when there are few enough objects

that size of an index is significantly smaller than the size of a pointer. An index specification would need a parameter for the length of the table, and the size of the index field.

```
for node.attr use index(47); -- 47 elements
for node.attr use reference_size(byte);
```

5. One might combine several attributes into a sequence, as Figure 4.1 on page 54 shows, using any of the above representations for elements of the sequence would.

   ```
   expr => children: seq of expr;
   for bin_exp.{left right} use sequence(children);
   for expr.children(*) use index(23)
   ```

   This example shows another reason we call these "reference representations" rather than "attribute representations": most of them apply to any reference to an object, including elements of sequences.

It might be undesirable to fix some of the parameters for hashed and indexed specifications in the IDL description. If you do not permit some parameters, you must provide some mechanism for postponing their binding. For example, suppose the generic hash representation has five parameters $\texttt{parm}_1$ through $\texttt{parm}_5$. Instead of providing these parameters in the implementation note via

```
for xxx use hash(parm₁, ..., parm₅)
```

you might require that the user define a package that exports these five parameters, and have the IDL description name this package. In Ada the package might take the form

```
package HashParms
    parm₁ : integer;
    ...
    procedure parm₅(...);
end HashParms
```

In the IDL declaration, you would then say

```
for xxx use hash(HashParms)
```

The package IDL generates would contain appropriate declarations to use `HashParms`.

If you specify one of these representations for a type, rather than an attribute, the representation would apply to any attribute of the type. Thus

```
for Integer use embed
```

says that you want to embed all integer-valued attributes in their nodes. If a user later says

```
x => a : Integer;
for x.a use pointer
```

```
structure example root E1 is
    A => a1: String;
    B => b1: Integer,  b2: Rational;
    C => c1: Rational;
    D => d1: Integer;

    E1 ::= A | B;    E2 ::= E1 | C ;

    E1 => x : Integer;   -- all members of E1 have x
    E2 => y : Rational;  -- all members of E2 have y
end
```

Figure 3.3: Simple Structure Declaration

a translator should issue an error message that this conflicts with the prior implementation note about integers.

For embedded integer-valued attributes, there are other space-saving encoding: offset and polynomial representations. If a node has three attributes `month`, `day`, and `year`, with ranges 1..12, 1..31, and 1960..2000, then representing the fields separately as unsigned integers requires $4 + 5 + 11 = 20$ bits. We can save space by representing each number as an offset within its range; the attribute fetch procedure can add the offset, while the store procedure subtracts it. Thus the smallest representation of the three fields as separate items requires $4+5+6 = 15$ bits. However, we could encode all three together as $((year - 1960) * 12 + month - 1) * 31 + day - 1$, which requires $log_2((40 * 12 + 11) * 31 + 30)$ bits, or 14 bits.

3.3 Strict Classes

An IDL strict class is a union of node types. Classes may contain other classes, and may overlap arbitrarily. Unlike unions in most programming languages, strict classes may have attributes. Figure 3.3 shows an example. Class `E1` contains members `A` and `B`; all members of `E1` have attribute `x`. We used to view this as an abbreviation mechanism, which prevents having to write

```
    A => x : Integer;
    B => x : Integer;
```

However, there are deeper semantic issues that require us to view classes as more than abbreviations. A simple reason to prefer the declaration in Figure 3.3 is that it clearly shows that you intend for `A` and `B` to have an attribute with the same name and same type.

Since we use classes as types of attributes, we need to design representations of classes. The following sections show several possible implementation techniques, which represent different engineering choices based on the tradeoffs we discussed at the beginning of Section 3.1.

3.3.1 Single-Type Representations

The obvious Pascal representation for an IDL structure declares one **record** type, with variants for each node type. Class variables and class-valued attributes would be pointers to such records. Similarly, in Ada one might define one **access record** type. All objects from a particular structure have the same type; an application cannot declare objects of some class narrower than the whole structure. The suggested Ada package from Chapter 4 of the DIANA reference manual implies a representation of this kind, since it defines one Ada type (`Tree`) for all DIANA nodes [Goos83].

The single-variant-record representation for a structure can be space-efficient, since each node takes only the space needed for the attributes stored in it (if the implementation language's compiler packs records efficiently). However, access to individual attributes can be slow. If a routine manipulates an object of class `E1` (from Figure 3.3), fetching the `x` attribute requires extra run-time code, as Figure 3.4 shows; the access procedure must determine the node kind at run-time before making the access. If the class containment relation forms a tree (see Section 3.3.3), a more efficient representation has sub-variants within variants. Figure 3.5 shows one possible representation. This is just as efficient in data space as Figure 3.4, requires somewhat less code space, and gives a slightly more time-efficient attribute fetch or store on some machines.

Another alternative, which does not rely on classes forming a tree, is useful if the target language provides its own representation specification facilities, and if compilers for the language do certain kinds of optimization. The IDL translator can use the record declarations of Figure 3.4, but tell the target language compiler to place particular fields at particular places within the records. The translator can give the same positional specification for the `A_x` field as the `B_x` field. When compiling function `Get_x`, the target language compiler would notice that all arms of the **case** statement contained identical code, and would simplify the procedure accordingly.

3.3.2 Distinguishing Between Classes

Section 2.2.7 on page 9 introduced the idea of to determining if a particular node is a member of a particular class. For each class `X`, the implementation

```
package Example_Struc is
    type node_kinds is (A_node, B_Node, C_Node, D_Node );
    type Example_Rep(node_kind:node_kinds) is private;
    function Get_x(n:Example_Rep) return Integer;
    ...
private
    type Example_Rep(node_kind:node_kinds) is access record
        case node_kind is
            when A_node =>
                A_a1: stringStruc;  A_x: Integer;  A_y: Rational;
            when B_node =>
                B_b1, B_x: Integer;  B_b2, B_y: Rational;
            when C_node =>
                C_c1, C_y: Rational;
            when D_node =>
                D_d1: Integer;
        end record;
end;

package body Example_Struc is
    ...
    function Get_x(n:Example_Rep) return Integer is
        begin
            case n.node_kind is
                when A_node => return n.A_x;
                when B_node => return n.B_x;
                when others => raise No_Such_Attribute;
            end case;
        end Get_x;
end;
```

Figure 3.4: Flat Node Representation in Ada

```
package Example_Struc is
    type node_kinds is (A_node, B_Node, C_Node, D_Node);
    type Example_Rep(node_kind:node_kinds) is private;
    function Get_x(n:Example_Rep) return Integer;
    ...
private
    type struc_rep(node_kind:node_kinds) is access record
        case node_kind is
            when A_node..C_node =>          -- class E2
                E2_y: Rational;
                case node_kind is
                    when A_node..B_node =>  -- class E1
                        E1_x: Integer;
                        case node_kind is
                            when A_node => A_a1: stringRep;
                            when B_node => B_b1: integer;  B_b2: Rational;
                        end case;
                    when C_node =>
                        C_c1: Rational;
                end case;
            when D_node =>
                D_d1: Integer;
        end case;
    end record;
end;

package body Example_Struc is
    ...
    function Get_x(n:Example_Rep) return Integer is
        begin
             return n.E1_x;
        end Get_x;

    pragma inline(Get_x);
end;
```

Figure 3.5: Nested Class Representation in Ada

```
function Is_E2(n:struc_rep) : boolean;
    begin
        Is_E2 := (n.node_kind = A_node) or (n.node_kind = B_node) or
                 (n.node_kind = C_node)
    end { Is_E2 };
```

Figure 3.6: Worst-Case Implementation of Is_E2 in Pascal

may provide an `Is_X` function that returns **true** if the argument is a member of the given class. Instead of one `Is_X` function per class, a target model might provide a single function `IsMember(node, class_type)`.

In the worst case, the implementation of `Is_X` must compare the node kind with all possible kinds for members of the class, as Figure 3.6 shows.

If the class containment partial order forms a tree, we can do better. The tree has pure classes as the interior elements and node types as the leaves. One can walk the tree in either NLR or LRN order[1] to sort the node kinds such that all members of each class are contiguous. More complex algorithms can find such encodings for some (but not all) non-trees [Lamb88]. Thus a node is a member of class `E2` if its kind is in the range `A_node..C_node`. The representation of Figure 3.5 also requires that all the node kind values for members of a particular class be contiguous.

A method that works for any class containment relation, regardless of overlaps, is to assign a number to each class and one to each node, and record the relationship in a boolean matrix indexed by class index and node kind. The cost of indexing into such a matrix is usually comparable to the cost of a range check, depending on the architecture of the target machine. Such matrices are usually sparse; compression techniques can reduce the size of the matrix while keeping the cost of a check low.

3.3.3 Partitions versus Overlapping Classes

If the class containment relation forms a forest (collection of trees), an implementor can partition the classes. Each class not contained in any others can be the root of a partition, and each partition may have a different style of representation. This can allow applications to define objects of types corresponding

[1]The older names for these are preorder and postorder; William A. Wulf introduced the newer notation to describe arbitrary tree walks by combining the letters L (for visit left subtree), N (for process node), and R (for visit right subtree).

```
A => a1:Integer, ... ; -- declaration of node type A
B => b1:Integer, ... ; -- declaration of node type B
C => c1:Integer, ... ; -- declaration of node type C

D1 ::= A | B ;         -- class D1
D2 ::= B | C ;         -- class D2

D1 => x : Integer;     -- all members of D1 have x
D2 => y : Rational;    -- all members of D2 have y
```

Figure 3.7: Non-Tree Class Declaration

to narrower classes than the whole structure. An interesting special case is a partition consisting entirely of classes without attributes; we can represent the whole partition as an enumerated type.

A designer cannot use a nested representation unless the classes form a tree. The classes of Figure 3.7 do not form a tree, since classes D1 and D2 overlap without either containing the other.

Other implementation languages, such as C, have no difficulty with overlapping classes. Figure 3.8 shows a possible C representation of the structure of Figure 3.7. It has one **struct** per node type and one **union** per class type, and one **struct** per class type. Each **union** has one element per member of the class, plus a member called "self" that represents attributes directly declared for the class. Thus the "self" variant of union type `CD1` contains a `D1_x` field (line 3.3.3), since class `D1` of Figure 3.7 has an `x` attribute. Each **struct** has one field per attribute, plus extra fields to ensure proper alignment of fields corresponding to attributes common to more than one member of a class; Thus `RD2` has a "`Filler001`" field (line 3.3.3) so that its `D2_y` field (line 29) lines up with those in `RB` and `RC` (lines 3.3.3 and 3.3.3). Figure 3.9 shows a picture of the field layouts corresponding to Figure 3.8. An IDL translator can ensure that each **struct** in a class has attributes defined on that class in the same position within each **struct**. This may waste some space in some node types, but ensures fast access to attributes. Snodgrass and his colleagues at the University of North Carolina in Chapel Hill support this style of representation for non-overlapping classes in their C implementation [Shan86].

The C representation makes it easy for a program to declare parameters and local variables of precisely the correct type. Figure 3.10 shows some typical examples of direct access to attributes. An implementation that wishes to

```
/* node names */
#define RA struct rRA
#define RB struct rRB
#define RC struct rRC
#define RD1 struct rRD1
#define RD2 struct rRD2
/* class names */
#define CD1 union cCD1
#define CD2 union cCD2
  /* class D1 contains A and B, plus attributes of its own */
CD1 {RD1 *self;  RA *A;  RB *B; };
  /* class D2 contains B and C, plus attributes of its own */
CD2 {RD2 *self;  RB *B;  RC *C; };
RA {node_header header;    /* standard node header, includes node type */
    int D1_x;              /* from D1 */
    int A_a1; ... };       /* other A-specific attributes */
RB {node_header header;    /* standard node header, includes node type */
    int D1_x;              /* from D1 */
    real D2_y;             /* from D2 */
    int B_b1; ... };       /* other B-specific attributes */
RC {node_header header;    /* standard node header, includes node type */
    int C_c1;              /* forces alignment of Y with RB */
    real D2_y;             /* from D2 */
    ... };                 /* other C-specific attributes */
RD1 {node_header header;   /* standard node header, includes node type */
    int D1_x; };           /* same relative position as x in RA and RB */
RD2 {node_header header;   /* standard node header, includes node type */
    int Filler001;         /* forces alignment of y with RB and RC */
    real D2_y; };
```

Figure 3.8: C Representation of an IDL Structure

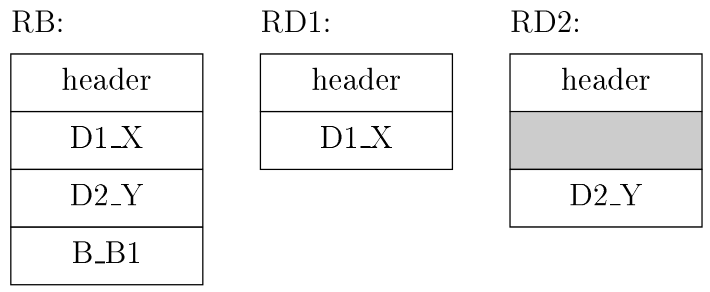


Figure 3.9: Layout of C **struct**s for Figure 3.8

```
/* take parameter of class D1 */

manip(D1obj)
CD1 D1obj;
{
    /* set attribute x */

    D1obj.self->D1_x = 1;

    /* do member-specific processing */

    switch (D1obj.self->header.node_kind)
    {
        case NA:D1obj.A->a1 = ...; break;
        case NB:D1obj.B->b1 = ...; break;
        default:error("D1 object must be A or B");
    }
}
```

Figure 3.10: Manipulation of C Representations

stress data abstraction would hide these within fetch/store macros.

Figure 3.8 represented a class as a **union** of pointers to **struct**s. An alternative is to represent a class as a **union** of **struct**s; this more closely resembles variant records. The former representation assumes only node types have a physical existence; the latter allows for allocating space for an object when you haven't yet decided what member of the class it will be. Each has implications for what information the header field must contain that go beyond the scope of this chapter.

3.4 Operations on Classes

We can view attribute fetching and storing operations (Section 3.2.1) as operations on classes. Aside from these, one can declare, create, delete, assign, and test for equality.

3.4.1 Creation

A target model will require some way for specifying what attributes to supply at node creation time. A sensible way to do this is to to use the view mechanism of derived structures. For example, given

```
structure S1 is
    A => A1 : T1, A2 : T2;
end
structure S2 from S1 is
    A => A3 : T3
end
```

you could direct the IDL translator to creating objects via `S1` and access objects via `S2`. Thus the node creation procedures will accept values for attributes `A1` and `A2` as parameters, and will assign a default value for `A3`. There should also be some way for users to specify the parameter order; the `Traversal` implementation note suggested in Section 4.2.2 is one possible mechanism.

As it stands, IDL does not provide a mechanism for specifying default initialization. However, this is a reasonable and simple extension; see Section 3.7.4.

3.4.2 Deletion

Since IDL structures can have arbitrary sharing, explicit object deletion is dangerous. Unless your IDL structures have very simple sharing, you must consider building a garbage collector. Chapter 6 outlines a garbage collection technique suitable for many IDL-based systems.

3.4.3 Assignment

The target model may need to include operations for assigning class-valued variables, since the target language's built-in rules for assignment may not be appropriate. In addition to explicit assignment, the target model needs to deal with implicit assignments, which show up in call-by-value and call-by-value-result parameter passing.

If the type model follows IDL closely, the source and destination of an assignment need not have the same type. For example, if we have the IDL declaration

```
Tree ::= inner | leaf;
inner ::= unary | binary;
```

then to a destination of type `Tree` you may assign a value of type `inner`, `unary`, `binary`, or `leaf`. We call this a *widening assignment*; it requires no run time checks, unless the target model calls for checking for undefined values.

Assignments in the other direction do require a check; we call these *narrowing assignments*. Section 2.2.7 introduced the idea of widening and narrowing.

3.4.4 Equality and Equivalence

To match IDL closely, the target model should represent variables as references (pointers, indices, and so on) to objects. Thus it makes sense to ask whether two variables reference the same object. In typical cases this might have a low-level implementation as pointer equality. However, some DIANA discussions led us to postulate representations that involved making multiple copies of the same "object" for performance reasons; here the equality test might involve comparing some form of global object identifier hidden away in the internal representation of the object. Thus the target language equality operator may not be appropriate.

3.5 Sequence and Set Representations

Sequence and set object representations typically involve a parameter for the element type. Implementations may involve instantiating generic packages, possibly with parameters provided by a representation specification. For example,

```
type t1 = seq of t2;
for t1 use circular_queue;
```

might lead the IDL translator to include the following Ada code in the interface module specification for the containing structure:

```
package t1Pkg is new circular_queue(t2_rep);
subtype t1_rep is t1Pkg.seqType;
```

where `t2_rep` is the Ada type that implements IDL type `t2`. If there is a second circular sequence type, with elements of type `t3` that happens to have the same Ada type as `t1`, a clever IDL translator need not instantiate package `circular_queue` again; instead, it could simply add a subtype declaration for `t3_rep`.

In languages that lack generics, we can achieve similar results by having the IDL translator expand some form of generic package description. The key point of this section is that implementation notes may need parameters, and we need not wire all that much information about specific representation packages into the IDL translator.

Some combinations of representational techniques and implementation languages may require that users supply parameters in the implementation note for things that we would prefer to keep hidden. For example, one sequence implementation requires threading the sequence through the nodes it contains; this requires that no node be on more than one such sequence. This requires adding a new attribute to each node to hold the pointer to the next element of the sequence; trying to use Ada generics for a "threaded" representation forces us to give such an attribute a name. It is best to supply the name in an implementation note rather than have the IDL translator invent one, since invented names inherently depend on processing details and could change from one run of the translator to the next.[2] Thus

```
type t1 = seq of t2;
for t1 use threaded(t1_thread);
```

leads to adding another fetch/store procedure pair

```
function Get_t1_thread(x:t2_rep) return t2_rep;
procedure Set_t1_thread(in out x:t2_rep; in a:t2_rep);
```

plus the type declaration

```
package t1Pkg is new
    threaded(t2_rep, Get_t1_thread, Set_t1_thread)
subtype t1_rep is t1Pkg.SeqType;
```

The operations provided by the generic package call the procedural parameters to build the sequence and iterate through it.

Ada forces a bit of additional complication on us. The threaded representation depends on `t2_rep` being an access type. To pass an access type as a generic parameter and allow the generic package to know it is an access type, you must pass both the access type and the type it points to. Figure 3.11 shows an outline of what the threaded package should look like.

To reiterate, this problem arose because we tried to use Ada package instantiation to define all the sequence operations on type `t1_rep`. We had to tell

[2]For example, the translator might invent `thread1` for the first thread field, `thread2` for the second, and so on. If you then delete the first threaded representation, the name for what used to be the second changes to `thread1`.

```
generic
    type ElementRep is private;
    type Element is access ElementRep;
    with function GetThread(x:Element) return Element;
    with procedure SetThread(in out x:Element; v:Element);
package threaded is
    type SeqType is private;
    procedure Append(in out S:SeqType; E:Element);
    function Get_Subscript(in out S:SeqType; I:integer) return Element;
    ...
private
    type SeqType is
        record
            First,Last:Element := null;
        end record;
end;

package body threaded is
    procedure Append(in out S:SeqType; E:Element) is
    begin
        if S.First = null
            then S.First := E
            else SetThread(S.Last, E)
        end if;
        S.Last := E; SetThread(E,null);
    end Append;

    function Get_Subscript(in out S:SeqType; I:integer) return Element is
        Cur : Element := S.First;
    begin
        if I<=0 then raise Subscript_Error end if;
        while I>1 loop
            if Cur = null then raise Subscript_Error end if;
            I := I-1; Cur := GetThread(Cur);
        end loop;
        return Cur;
    end Get_Subscript;
    ...
end;
```

Figure 3.11: Form of Threaded Sequence Package in Ada

the body of package "threaded" how to manipulate the t1_thread attribute, and the only way to do so was to pass the specification procedures as generic parameters to the specification. We had to define these procedures within the package defining the IDL structure.

Ideally, we shouldn't need to name the thread fields at all, since we wouldn't want users to manipulate them directly. In an alternative scheme, the IDL translator could "macro expand" the specification of the threaded package, plugging in `t2_rep` in the appropriate places. Figure 3.12 shows an IDL interface module that does so. In the body of the IDL structure package, we instantiate the threaded package and use it to define the functions whose headers we macro-expanded in the specifications.

3.6 Other Implementation Details

The previous sections showed that much of what an IDL translator does depends only on the names of the types and the names given in the implementation note. Some implementation notes require that the the IDL translator embody more information about representational details.

3.6.1 Size

Sometimes an IDL translator needs to know the size of a type. For example, C has union types rather than variant records. To keep attributes of a class in the same relative positions in different nodes within the class, while allowing for overlapping classes, the translator must align fields of different struct declarations.

This poses no problem for built-in types. However, the IDL translator does not know the size of any private types. Thus a structure to be implemented in C must have a size representation for each private type; for example, a user might say

```
type t1;
for t1 use package(p1,t2);
for t1 use size(bits(27))
```

Some target models might require additional information, such as whether the private type uses heap storage.

3.6.2 Operation Restrictions

Previous sections have discussed implementation notes that controlled representations of references and objects. Another note controls what operations a generated package should contain. The implementation notes

```
for <type reference>use operations(name_1, ..., name_N)
for <type reference> use forbid(name_1, ..., name_M)
```

```
package IDLstruc is
    ...
    -- defines Get, Set functions for all visible attributes
    -- does not define Get_t1_thread, Set_T1_thread
    ...
    type t1_rep is private;
    procedure Append(in out S:t1_rep; E:t2_rep);
    function Get_Subscript(in out S:t1_rep; I:integer) return t2_rep;
private
    type hidden_t2_rep is private;
    type t2_rep is access hidden_t2_rep;
    type hidden_t1_rep is private;
    type t1_rep is access hidden_t1_rep;
end;

package body IDLstruc is
    function Get_t1_thread(x:t2_rep) return t2_rep is ...;
    procedure Set_t1_thread(in out x:t2_rep; in a:t2_rep) is ...;
    package t1pkg is new
        threaded(t2_rep, hidden_t2_rep, Get_t1_thread, Set_t1_thread);
    subtype t1_rep is t1pkg.SeqType;

    procedure Append(in out S:t1_rep; E:t2_rep) is
    begin
        t1pkg.Append(S,E);
    end Append;

    pragma inline(Append);
    ...
end;
```

Figure 3.12: Hiding the Thread Fields in Ada

define what operations the designer wants to make available on each type. `operations` names allowed operations; `forbid` names forbidden operations. You would normally use only one or the other; if you use both, any conflict between the two is an error.

The two operations on attributes are `fetch` and `store`. Restricting either of these operations eliminates the corresponding member of the procedure pair from the interface. Restricting both means the attribute is physically present but invisible. This might happen with a series of derived structures, where a process using the first sets an attribute, intermediate processes ignore it, and a process at the end of the series fetches the attribute. In addition to reducing the size of an interface module, this gives fine-grain control over who can change an attribute.

3.7 Readers and Writers

Unless two processes have identical representations for a data structure (down to the exact values of pointers), one or the other must transform the representation of the data. In an IDL-based system, both processes cooperate; one process writes the information in an exchange representation, and the other reads the exchange representation into an internal form. This section discusses readers and writers for the ASCII external form introduced in Section 2.3.4 on page 14.

3.7.1 Reader Design

An IDL reader is a parser for the external representation scheme. This grammar requires a lookahead of 2 because of labels, but a lexer could reduce a parser's lookahead to 1 by treating a name followed by `^` or `:` as a single symbol.

A top-down parser requires less overhead than a bottom-up parser. A top-down parser can create a node representation as it begins to parse a node production, then store individual attributes in the current node as it parses attribute productions. A bottom-up parser must keep the attribute/value pairs around in some auxiliary structure until it is ready to build the whole node.

When designing the reader, you must decide the extent to which it will detect and recover from errors. If you assume its input always comes from a correct IDL writer, you might have little error detection. If you assume input might come from hand-generated files, or incorrect processes, you might need more error detection. Figure 3.13 gives a partial list of errors you should consider having your reader check for.

1. Syntax error (violation of the external form BNF). The message should show clearly what the parser found and what it expected to find.
2. Invalid attribute name for this node type.
3. Invalid attribute value. The message should say what type of value the reader expected.
4. Unknown node type (possibly a special case of error 3, except that it includes the root node).
5. This node type is not a member of the necessary class (possibly a special case of error 3, except that it includes the root node).
6. Invalid value for this sequence (or set).
7. The representation of this attribute does not permit this value. You might consider special cases of this message, such as "expecting an integer in range A..B", or "expecting a string of exactly K characters".
8. Too many distinct values for this attribute. This is a special case of error 7, but applies only to indexed or hashed attributes.
9. Inappropriate sharing (a special case of error 7). A value is shared in the input file, but the representation does not permit sharing. This could happen because the attribute is embedded, the two attributes that share the value have different representations, or the value is part of two distinct threaded lists of the same type.
10. Undefined label. Since IDL allows forward references, the reader can only give this message at the end of the input file. Ideally the message should give some way to find all the places in the input that reference the label.
11. Assertion failure. This message ought to identify the assertion that failed and explain why it failed. Simple readers might not check all assertions.

Figure 3.13: Partial List of Reader-Detected Errors

Because of labels and cross-references, the reader must contain the same label-resolution logic found in many single-pass compilers.

3.7.2 Writer Design

If a writer has no other purpose but to emit an exchange representation, it has a simple structure. The writer skeleton shown in Figure 3.14 makes two passes over the data structure: one to detect shared nodes (for later labeling), and one to write out the data structure. The alternative is to make one pass, writing out a label for each node. Performance figures show that label processing by the reader is expensive, while the extra pass to detect sharing is cheap [Lamb87a]. Thus the two pass method is more efficient.

This skeleton assumes that each node has a pair of bookkeeping flags, `Touched` and `Shared`. At any call on `Writer`, the `Touched` field of each node equals the current value of `TouchPhase` (the reader leaves these fields **false**). At the end of the call on `Writer`, the `Touched` attribute of each node has the same value as at the beginning. It also requires that you can reach nodes in a structure from the root node by following attributes, and so need no extra links to guide iteration.

For many data structures, you consider some attributes to be structural and others to be auxiliary. For example, the DIANA attributes whose names start with "`as_`" are structural attributes. You might add implementation notes to identify the structural attributes; this would allow a more complex writer to use nesting only for structural attributes, and cross-references for non-structural ones. This makes the nesting of the external representation match the designer's idea of what the spanning tree for the structure should be, and prevents forward references from distorting the structure. Figure 3.15 shows two variant external representations for a tree for the expression A+(B*A). The `next` field of a tree node points to the next occurrence of node representing the same "common subexpression". In 3.15a, printing structural attributes first forces the external form to follow the intuitive tree structure. In 3.15b, the writer printed the `next` field first, thus printing all the "A" leaves first; understanding the conventional tree structure requires following cross-references rather than following the nesting.

You might choose to complicate the basic structure of the writer to serve additional purposes. For example, a debug writer might label each node with its memory address. Also, a debug writer cannot assume that the data structure is correct; in particular, it cannot assume that Touched and Shared flags are in a consistent state. Thus it may need to maintain auxiliary data structures to record sharing.

```
procedure Writer(Root:NodeType);

    procedure MarkShared(Node:NodeType);
    begin
        if Node.Touched = TouchPhase
            then begin Node.Shared := true;  AssignLabel(Node) end
            else begin
                Node.Touched := TouchPhase; Node.Shared := false;
                case Node.Kind of
                    { code to call MarkShared for all subnodes }
                    end
                end
    end { MarkShared };

    procedure WriteNode(Node:NodeType);
    begin
        if Node.Touched = TouchPhase
            then Write(Node.Label," ^")
            else begin
                Node.Touched := TouchPhase;
                if Node.Shared then Write(GetLabel(Node),":");
                case Node.Kind of
                    { code to write the node type, followed by all its
                      attribute, using WriteNode for node-valued attributes
                    end
                end if
    end { WriteNode };

begin
    TouchPhase := not TouchPhase;  MarkShared(Root);
    TouchPhase := not TouchPhase;  WriteNode(Root);
end { Writer }
```

Figure 3.14: Pascal Writer Skeleton

```
tree ::= inner | leaf;
leaf => name : String;
inner => left : tree,   right : tree,   op : String;
leaf => next : tree_void;
tree_void ::= tree | void;
```

Representations of A+(B*A) with forward links from each node to the next occurrence of a "common subexpression."

```
inner [
  op "+";
  left
    leaf [ name "A";
          next L1 ^ ];


  right
    inner[
      op "*";
      left leaf [name "B";
                 next void];
      right L1:
        leaf [name "A";
              next void]
    ]
]
```

(a) Structural Attributes First

```
inner [
  op "+";
  left
    leaf [name "A";
      next L1:
        leaf [name "A";
              next void ] ];
  right
    inner[
      op "*";
      left leaf [name "B";
                 next void];
      right L1 ^


    ]
]
```

(b) Depth First Order

Figure 3.15: Influence of Structural Attributes on External Form

3.7.3 Private Types

IDL provides *private types* for declaring objects and attributes whose representation is unknown to the IDL translator. One tells the IDL translator what external representation to use for the type, then provides a package that implements the internal representation. For example, to declare a stack type one might say

```
    type MyStack = seq of ElementType;
    for MyStack use package(myPackage,repType);
```

Externally, `MyStack` is represented by a sequence of objects of type `ElementType`. Internally, one uses type `repType` from package `myPackage`.

To allow reading and writing of private types, a package must provide operations that interact with the graph readers and writers. One might choose a variety of ways of doing this; we chose a method that prohibits private packages from directly reading or writing files. This permits the readers and writers to hide the details of the external representation. The IDL translator, run-time library, readers, writers, and generated interface understand the representational details of all the non-private types in a structure. The input/output interface for a private type must use these known types.

Each IDL system might have its own conventions for how the reader/writer communicates with private packages. Figure 3.16 shows the IDL declarations for a private type and the Pascal module that implements it, using the conventions of my Ph.D. thesis system [Lamb83]. The types `Nxsrc` and `Rxsrc` come from the interface module, typically by file inclusion.

A module implementing a private type must supply

- A type declaration for the internal representation of the private type (`PosType` in Figure 3.16).
- An input mapping routine that accepts the reader's representation of the data and returns a value of the private type (`IPosType` in Figure 3.16).
- A "detect sharing" routine that accepts a value of the private type and marks it or portions of its substructure that are shared (`MPosType` in Figure 3.16). The writer needs this routine because it does not know whether the representation of the private type has room to record sharing (such as via marker flags) or whether it needs some auxiliary data.
- An output mapping routine that accepts a value of the private type and returns the writer's representation for the external representation (`OPosType` in Figure 3.16). If substructure of the private type already has a representation that the reader/writer understands, the output routine can reference it rather than create new objects. If the private value is shared (as recorded by the detection-of-sharing routine), the output

```
{ main file for PosModule: implementation of SrcPos }
{ IDL declarations:
  type srcpos = xsrc;
  xsrc => line: Integer, charp: Integer;
  for srcpos use package(PosModule,PosType);
}

const charsPerLine = 256;
type PosType = integer;
  { Nxsrc, Rxsrc imported from generated interface module, probably by
    file inclusion }
  Nxsrc =  ^ Rxsrc;
  Rxsrc = record
    line, charp: integer; { plus overhead fields }
   end;

{ input mapping routine }

procedure IPosType(var D:PosType; X:Nxsrc);
   begin
   D := X ^.line*CharsPerLine + X ^.charp
   end;

{ detection-of-sharing routine }

procedure MPosType(var D:PosType);
   begin { PosType objects are never shared } end;

{ output mapping routine }

procedure OPosType(var Ret:Nxsrc; X:PosType);
   begin
   new(Ret);
   Ret ^.line := X div CharsPerLine;
   Ret ^.charp := X mod CharsPerLine;
   end;

{ deletion of generated output representation }

procedure DPosType(var X:Nxsrc);
   begin Dispose(X); X := nil end.
```

Figure 3.16: Pascal Module for a Private Type

mapping routine must return the same reader/writer representation object every time writer passes it the same value, so that the external representation can preserve sharing.

- A deletion routine that accepts the object returned by the output mapping routine and deletes it (`DPosType` in Figure 3.16). In a target system with garbage collection this routine need not do anything. It should only delete the portions of the object generated by the output mapping routine.

The reader's representation and writer's representation for a type are identical (type `Nxsrc` in Figure 3.16). The IDL translator can derive the writer's data structure for the external representation directly from the structure declaration.

- If the external representation is an integer, rational, boolean, or string, the writer's data structure is a string containing the characters of the external representation. This allows the private type module maximum generality. A debugging version of the writer might also check that the string was lexically correct.
- If the external representation is a set or sequence, and the target language allows some form of generic sequence type, the writer's data structure is a sequence of records representing the elements of the set or sequence, in the order they appeared in the input. Alternatively, the user's package could provide a trio of input routines, "create empty sequence (or set)", "append element to sequence (or set)", and "finish appending to sequence (or set)", plus a similar trio of output routines.
- If the external representation is a node, the writer's data structure is the default representation the IDL translator would build for such a node – typically a record or element of a variant record.

The easiest way for the reader/writer to deal with the primitive integer, rational, boolean, and string types is via modules like those for private types.

3.7.4 Data Transformations

In large systems one may have a series of data structures, each only slightly different from the previous. A designer may use the IDL derivation facilities (Section 2.4.2) to derive one structure from another. The new structure may add new node types, delete old ones, add new attributes to existing nodes, and delete old attributes.

With process declarations a designer can specify that the output of a particular process becomes the input of another. The structure definition for the input need not be the same as that of the output, provided one was derived from the other. The IDL system can supply a writer/reader pair that makes some simple transformations from one structure to another.

Some transformations are easy. Deleting an attribute is trivial. If one adds an attribute, the reader can supply a default value; by saying

```
for node.attr use default("xxx");
```

the designer could specify a default in the IDL description. If one adds new node types, the reader has no extra work: the input cannot have any such nodes in it. If one deletes a node type, the reader must report errors if any such nodes exist in the input.

We can speculate about improved IDL systems that could support more complex transformations or initializations. IDL contains a sublanguage for making assertions about properties of a structure. An IDL system could generate a program for verifying whether an instance of the structure satisfies the assertions; **?CS?** describes such a checker for a large subset of the assertion language. For example, the checker could determine whether a particular integer attribute recorded the depth of the node that contains it from the root. It may be possible to interpret some assertions about attributes as computations for defining their values.

Chapter 4

Extending the IDL System

Given the basic IDL notation for describing programs and their data, and tools to process the notation, it is easy to imagine several extensions one might wish to make. This chapter discusses extensibility mechanisms build into IDL, and some other extensions I have considered that require changing the IDL notation. These ideas are sketchy; I expect to revise them as I extend the Queen's IDL system.

4.1 Built-In Extensibility

IDL has four built-in mechanisms to allow different implementations to extend the system: implementation notes, private types, atomic types, and type constructors.

4.1.1 Implementation Notes

In all versions of IDL, implementation notes (once called representation specifications) have been the main route by which the designers expected to see variations among different systems. Appendix B summarizes the standard implementation notes implementations might wish to support.

The main purpose of implementation notes is to name different representations for IDL types; the only limit to their variability is how much information an IDL translator implementor wishes to embed in the translator. For example, one might provide several implementations of sequences, such as fixed-length arrays, singly-linked lists, doubly-linked lists, or threaded lists.

However, the only real restriction on implementation notes is that they should not change the semantics of the abstract IDL types. Thus, for example, in Section 3.7.4 we suggested a note of the form

```
for node.attr use default(value)
```

which would specify the default value to which a reader initializes an attribute.

4.1.2 Atomic and Private Types

In previous versions of IDL the predefined type names `integer`, `boolean`, `rational`, and `string` were all reserved words. There is no good reason why they shouldn't simply be predefined identifiers. This makes it easier to extend the set of predefined types.

One obvious extension is to predefine type names that imply particular representations (unless the user overrides the default with an implementation note). For example, the target model may define `integer` as having a 32-bit representation. The IDL translator may supply a `short_integer` predefined type that has a 16-bit default representation.

Private types are an alternative mechanism for extending the set of representations. For example, if a user wanted an arbitrary-precision rational number implementation, she could write such a package and say

```
    type myInteger = Integer;
    for myInteger use package(myPackage,arbitrary_int)
```

Any particular system will have conventions for how private types will interact with the rest of the run-time system; Section 3.16 shows one possible set of conventions for a Pascal-based implementation.

4.1.3 Type Constructors

In earlier versions of IDL, `set` and `seq` were reserved words. They should be identifiers to allow implementations to extend the collection of type constructors.

One extension I am thinking of is to add map and relation extensions.

- A relation is a set of ordered n-tuples, which one might think of as a set of nodes in IDL. Operations on relations include the ones that relational databases typically provide. Thus one might declare

  ```
      row => f1:t1, ..., fN:tN;
      x => rel : relation of row;
  ```

- A map is a data structure representing a function. One way to represent a map is as a set of ordered pairs, with the restriction that no value may occur more than once as the first element of the pair. Thus one could think of a map as a set of objects of some node type where the node had exactly two attributes. Typical operations on maps are application (table lookup) and element redefinition (table entry). Thus one might declare

  ```
      elem => key:keyType, val:valType;
      x => attr : map Of elem;
  ```

4.2 Notational Extensions

Some extension to IDL might require changing or augmenting the syntax of the IDL notation.

4.2.1 Abbreviations

One of our design constraints with IDL was to avoid notational abbreviations. We wanted to avoid adding such mechanisms until we were certain we understood the base functionality. Typical extensions one might consider include

- Allow several implementation notes in the same clause.
  ```
  for x.y use rep1(arg1),rep2(arg2);
  ```

- Allow several fields of the same type.
  ```
  x => f1, f2 : t1, f3, f4 : t2;
  ```

4.2.2 Referencing Multiple Attributes

Sometimes one would like to refer to several attributes of a class at once. The form

```
<implementation reference> ::= <multiple attribute
reference>
<multiple attribute reference> ::= <name>0.'{' { <name>1 #
, }+ '}'
```

is a Queen's extension; it refers to a group of attributes whose representations you need to coordinate. `<name>`$_0$ is the name of a class; each `<name>`$_1$ is one of its attributes. For example,

for `someClass.{attr`$_1$`, attr`$_2$`, ...}` **use** `Traversal`

says that depth-first tree traversals should first visit `attr`$_1$, then `attr`$_2$, and so forth.

As another example, Figure 4.1 shows a structure, `tree`, that has several distinct node-valued attributes per node. We might want a representation that makes generic tree-walks over this structure easy. The derived structure `tree2` does so by representing all the node-valued attributes of each node as a single sequence; clients of this representation could refer to `x.children[i]` instead of `x.left` or `x.right`.

```
structure tree root expr is
    expr ::= if_exp | oper_exp | leaf;
    oper_exp ::= bin_exp | unary_exp;
    if_exp => cond: expr,   then: expr,   else: expr;
    oper_exp => op: operator;
        bin_exp => left: expr,   right: expr;
        unary_exp => child: expr;
    operator ::= plus | minus | ... ;
        plus =>; minus =>;
    leaf => ...;
end

structure tree2 from tree is
    expr => children: seq of expr;
    for bin_exp.{left, right} use Sequence(children);
    for unary_exp.child use Sequence(children);
    for if_node.{cond, then, else} use Sequence(children);
    for leaf.{} use Sequence(children);
end
```

Figure 4.1: Sequence Representation of Attribute Group

4.2.3 Other Notations

If a tool designer needs some completely different notation to capture something she wants to do with IDL, one possible mechanism is to hide the extensions in annotations. Annotations are comments with a particular easy-to-recognize prefix. For example, my research project keeps the grammar for an input language in the same file as the IDL description of the abstract syntax, for ease of maintenance. A simple tool extracts the grammar and passes it to a parser generator. The section for a simple expression might look like

```
expr ::= binary;
binary => left : expr,   op : operator,   right : expr;
--G expr ::= expr addOp factor | factor
--G factor ::= factor mulOp term | term
```

The `--G` looks like a comment to IDL, but is easy for the grammar extractor to find.

Chapter 5

Formal Semantics

A formal semantics is a precise description of the meaning of a notation using a method that itself has a precisely defined syntax and semantics. Using a formal semantics instead of some more informal description has several advantages.

1. Formal methods give a precise specification for what the notation means. Informal methods are prone to ambiguity and omission.

2. Formal methods give a rigorous way to reason about the notation. In particular, it gives a way to verify that an implementation of the notation is correct.

3. Formal methods typically give some way to abstract away from implementation-specific details, and specify properties that must be true for all possible implementations.

The technique we have used in the IDL work is *denotational semantics*. Tennent [Tenn81] gives an introduction to the basic ideas of using denotational semantics to programming languages; Gordon [Gord79] and Stoy [Stoy81] give more advanced treatments. The key idea is as follows. One bases a denotational description of a language on the abstract syntax of the language. One gives the meaning of each construct of the language as a *denotation*: a mathematical object that models the meaning of the construct. Typically a denotational description has one *semantic function* for each major syntactic class (such as expressions, commands, and definitions) that gives the meanings of all abstract syntax trees that are instances of that class. To define the semantic functions, one gives a set of *semantic equations*, one for each member of the syntactic class. Thus if a language has assignment commands, sequences of commands, if commands, and while commands, one would define the semantic function for commands with four equations, one for each type of command.

Typically each semantic function takes arguments that supply contextual information, such as the meanings of the identifiers. One typically gives the meaning of a construct in terms of the contextual information and the meanings of the components of the construct. This structurally-oriented approach leads naturally to a structural method of reasoning about instances of the notation; one proves or defines a property for the primitive notions, then deals with the constructs via structural induction.

Formal methods in general, and the ones we have used with IDL in particular, suffer from two important problems.

- Most people who would want to understand IDL have difficulty following mathematical descriptions.
- Without tools to help check the semantic equations, semanticists are prone to making errors in their descriptions.

I know of no good solution to the first problem, other than education. For the second, I hope to supervise a series of student projects to import or construct a suitable set of tools.

Chapter 6

Garbage Collection

Most IDL-based systems would use a heap to allocate space for IDL objects. In a large system, it is likely that many objects have limited lifetime, so the space needed for all objects at any one time is probably much smaller than the space needed for all objects over the life of the program. Thus an IDL-based system must almost certainly solve the problem of releasing objects once they are no longer useful.

Explicit deletion of objects in a system with sharing is dangerous; you can rarely be sure you have zeroed the last pointer to a block before releasing it. Garbage collection is a much safer alternative. This chapter describes a method of garbage collection (designed by Nestor, Newcomer, and Steele) suited for IDL implementations in languages like C.

6.1 Self-Identifying Storage

A major argument against garbage collection strategies in conventional languages is that it is impossible to know where to find all the pointers. In an IDL-based system with self-identifying storage, this argument is false.

Many storage allocation methods require a small amount of overhead in each block of the heap. If we are careful to design this overhead so that the length of each block is encoded in the block, then each storage block is self-identifying. In an IDL-based system, we can store node tags and bookkeeping flags in the overhead words, since we can use the node tags to index into a table of block sizes.

6.2 Garbage Collector Structure

A garbage collector typically has three phases:

- Mark all heap objects as potential garbage.

- Find all pointers, and mark the objects they reference as non-garbage.
- Return all unmarked heap objects to the free lists.

Pointers to IDL objects can occur in one of four places:

- In other IDL objects.
- In registers.
- In static (own) storage.
- On the stack.
- Implicitly, in "hidden" pointers such as those of the `Indexed` implementation note.

Pointers in IDL objects are no problem, since the storage blocks are self-identifying, and the IDL system can create tables that describe where in each node to find the pointers. Pointers in registers are a special case of pointers on the stack, since procedure call conventions can force all registers onto the stack at the call to the garbage collector. Section 6.4 discusses how to handle hidden pointers; however, many useful IDL-based systems might have no hidden pointers. Thus the interesting problem is how to handle pointers in static and stack storage.

6.3 Finding Pointers in the Stack

With help from the compiler, we can treat static storage and stack frames as IDL objects. However, for stacks this method imposes the additional overhead of creating the object header for the stack frame on each call, which can make procedure calls unacceptably slow. Furthermore, most implementors cannot afford to modify the compiler in this way.

We can avoid the need for compiler help and runtime overhead in procedure calls with a technique John Nestor developed. It relies on four observations about IDL-based systems:

- It is easy to find the beginning and end of the stack, and thus iterate over it.
- It is easy to tell if an address is on the heap, by scanning a list of begin/end pairs for each storage area.
- The word before any IDL object is a storage header.
- It is easy to scan the heap in order of increasing addresses.

The garbage collector scans the stack looking for words that might be pointers to IDL objects. It rejects many words immediately because they could not be heap addresses. It ignores any word that appears to point into the heap if the word before the one it points to has the bit set that says the garbage collector has already scanned it, *whether that word is really a storage header or not.* It stores any words that remain into a small buffer area.

When the buffer fills up, or the scanner finishes with the whole stack, the garbage collector sorts the buffer in increasing order of addresses. It then scans the heap in storage order, comparing the address of each storage block with words in the header; this is a linear operation since both lists are sorted. When it finds a word in the buffer that points to the word after a storage header, it can mark the storage block as non-garbage. When it finishes with the buffer, it can go back to scanning the stack from where it left off.

This method may fail to find some garbage, since some words on the stack might accidentally look like pointers. This is unlikely, but it is safe: storage leaks lead to performance problems, which are easier to live with than the errors that result from freeing non-garbage. It requires a fixed overhead for the buffer area, which it can allocate at system startup time. Given a list of pointers to the beginning and end of static storage segments, it can also find pointers in static storage.

6.4 Hidden Pointers

Hidden pointers are references represented by a mechanism such as that of the `Indexed` implementation note. Finding such pointers in IDL objects is no problem, since the IDL translator can create tables that say where in each object to find them, and how to turn them into full pointers. However, hidden pointers on the stack are impossible to distinguish from small integers.

One way to deal with hidden pointers is to assume that any object such a hidden pointer might reference is non-garbage. For example, for each `Indexed` attribute there is a table that contains full pointers; the garbage collector could locate these tables through the runtime symbol table and mark all objects they reference.

An alternative relies on having the user call the garbage collector explicitly at certain "safe" points. One IDL rule is that when a structure is consistent, every node that is part of the structure is reachable from a root object. User code could pass these roots to the garbage collector, and declare that only the objects reachable from those roots are worth keeping. This *can* lead to the same dangling pointer problems as explicit deletion, but is much less likely to do so.

Appendix A

Extended BNF

This report uses an extended version of BNF to describe context-free syntactic rules; John Nestor developed the original version of these conventions for the C-MU Front End Generator [Nest81]. Its primary differences from other BNF variations are:

- All its constructs are linear sequences of characters, suitable for machine processing. Other BNFs sometimes use superscripts or subscripts.
- It aims to minimize how much a designer has to write. Thus other BNFs may require

  ```
  A ::=  X ( ',' X )...
  ```

  instead of

  ```
  A ::= { X # ',' }+
  ```

This section gives both the concrete and abstract syntax of a BNF grammar. The concrete syntax gives the rules for what you would type to describe a grammer; the abstract syntax is an IDL description for an internal representation of the grammar. *newNT* stands for a new non-terminal; in any one description, it is the same nonterminal in all occurrences.

- Grammar. A grammar is a sequence of rules; there may be special means for describing the start symbol and other grammar properties.

  ```
  <grammar> ::= { <production> ';' }* ;
  <production> ::= <nonterminal> '::=' <bnf> ;
  ```

  ```
  grammar => rules: seq of production;
  production => lhs : nonterminal, rhs : bnf;
  ```

- Symbols. In the concrete syntax, a nonterminal is any sequence of letters, digits, spaces, and underscores enclosed in angle brackets (<>). A terminal is any sequence of printable characters enclosed in single quotes ('); to include quotes within a terminal, double them (thus 'it''s' is a terminal containing the word "it's").

```
<bnf primary> ::= <terminal> | <nonterminal>;
<nonterminal> ::= ' <' { <letter> | ' ' | '_' | <digit>
}+ ' >';
<terminal> ::= '''' { { <printable> # '''' } | '''''' }+
'''';
```

```
Type Token;

symbol ::= nonterminal | terminal;
    nonterminal =>; terminal =>;
symbol => lx_symrep : Token;
bnf_primary ::= symbol;
```

- Alternation.

```
<bnf> ::= <bnf alternation>
<bnf alternation> ::= { <bnf seq> # '|' }*;
```

```
bnf ::= bnf_list
bnf_list ::= bnf_alt; bnf_alt =>;
bnf_list => list : seq of bnf
```

An expression of the form

bnf_1 | ... | bnf_N

is equivalent to an occurrence of a new nonterminal *newNT* where

newNT ::= bnf_1; ... *newNT* ::= bnf_N;

are new productions.

- Sequences. Any sequence of BNF expressions is a BNF expression. Sequencing has higher precedence than alternation.

```
<bnf seq> ::= { <bnf primary> }*;
```

```
bnf_list ::= bnf_seq; bnf_seq =>;
```

- Grouping.

```
<bnf primary> ::= '{' <bnf> '}';
```

```
bnf_primary ::= bnf_single;
bnf_single ::= bnf_group; bnf_group =>;
bnf_single => item : bnf;
```

An expression of the form

{ *bnf* }

is equivalent to a new nonterminal *newNT* where

newNT ::= *bnf*;

is a new production.

- Non-empty list.

```
<bnf primary> ::= '{' <bnf> '}+' ;
```

```
bnf_single ::= bnf_plus; bnf_plus =>;
```

An expression of the form

{ *bnf* }+

is equivalent to a new nonterminal *newNT* where

newNT ::= *bnf* | *newNT bnf*;

is a new production.

- Possibly empty list.

```
<bnf primary> ::= '{' <bnf> '}*' ;
```

```
bnf_single ::= bnf_star; bnf_star =>;
```

An expression of the form

{ *bnf* }*

is equivalent to a new nonterminal *newNT* where

newNT ::= | *newNT bnf*;

is a new production.

- Non-empty separated list.

```
<bnf primary> ::= '{' <bnf> '#' <bnf> '}+' ;
```

```
bnf_double ::= bnf_sep_plus; bnf_sep_plus =>;
bnf_double => element : bnf, separator : bnf
```

An expression of the form

{ bnf_1 `#` bnf_2 }`+`

is equivalent to a new nonterminal *newNT* where

newNT `::=` bnf_1 `|` *newNT* bnf_2 bnf_1`;`

is a new production. Alternatively, it is equivalent to

bnf_1 { bnf_2 bnf_1 }`*`

- Possibly empty separated list.

```
<bnf primary> ::= '{' <bnf> '#' <bnf> '}*' ;
```

```
bnf_double ::= bnf_sep_star; bnf_sep_star =>;
```

An expression of the form

{ bnf_1 `#` bnf_2}`*`

is equivalent to

{ { bnf_1 `#` bnf_2}`+` }`?`

- Elements once in any order.

```
<bnf primary> ::= '{' { <bnf> # '#' }+ '}?';
```

```
bnf_list ::= bnf_once; bnf_once =>;
```

The expression

{ bnf_1 `#` ... `#` bnf_N}`?`

is equivalent to

{ bnf_1 `|` ... `|` bnf_N}`*`

restricted so that each bnf_i can appear at most once. The special case with exactly one element has the obvious meaning of an optional element.

- Difference.

```
<bnf primary> ::= '{' <bnf> '#' <bnf> '}-' ;
```

```
bnf_primary ::= bnf_diff;
bnf_diff => left : bnf, right : bnf;
```

An expression of the form

$\{\ bnf_1$ `#` $bnf_2\ \}-$

means any bnf_1 that is not a bnf_2. In general this is only computable for regular expressions.

Appendix B

Standard Implementation Notes

This appendix summarizes standard implementation notes. All implementations should consider implementing these notes, but need not do so; however, if you are an IDL implementor and want to support an implementation note with functionality similar to one of these, you should use the name I give here rather than inventing your own. This is the appropriate compromise between rigidly defining a set of implementation notes and providing no guidelines at all.

Three implementation note apply to any nameable IDL entity.

- Supply the name to use for the entity (node, class, type, attribute, structure, process, port, definition) in the target model.

  for *reference* **use** `target_name(`*identifier*`)`

  This gives a way for you to name entities in the target model that conflict with IDL keywords.

- Name the operations allowed on the entity.

  for *reference* **use** `operations(`*identifier*`,...,`*identifier*`)`

  This restricts the operations to a subset of those the target model defines.

- Name the operations forbidden on the entity.

  for *reference* **use** `forbid(`*identifier*`,...,`*identifier*`)`

  This is an alternative way to restrict the operations to a subset of those the target model defines.

B.1 Structures

If you apply a process-specific note (such as **use** `generate`) to a structure, an IDL translator invents a process with a default body; the default may vary from translator to translator. For example, one suitable default might be a full-blown interface module, a nested ASCII writer, and an ASCII reader.

B.1.1 References

Section 3.2 discusses reference representations in more detail. Except for `hash` (which applies only to attributes), all these implementation notes apply to any object reference (attribute, or element of composite type).

- `pointer`. Represent the reference as an implementation-language pointer.
- `embed`. An embedded object exists only as a part of the object that references it; it cannot be shared. Embed is the default for types boolean, integer, and rational. The enumeration and nil representations are special forms of the embed representation.
- `unique`. Here there is exactly one object of the given type with any given value.
- `index`. The reference is a small integer index into an auxiliary table.
- `hash`. This note applies only to attribute references. The reference is stored in a hash table, with the node's address as a key. It is especially useful when most attribute instances have a particular default value, which need not be stored in the table.

All references have the operations `fetch` and `store`, which you may use in `operations` and `forbid` notes.

The `reference_size` note controls the space allocated for a reference. It takes the same parameters as the `size` note (see Section B.1.2). Thus, for example,

```
type X = ...;
a => b : X;
for a.b use pointer;
for a.b use reference_size(bytes(2));
```

uses two-byte pointers pointers; saying

```
for a.b use size(bytes(2));
```

would mean that the objects referenced from `a.b` attributes (a subset of the objects of type X) are two bytes long.

The reasoning for the distinction between `size` and `reference_size` is:

- You might want to supply a `reference_size` on a type, implying a default for any references to objects of that type.
- You might want to supply a `size` on a reference, controlling the size of the subset of objects of the attributes type that one might reference through the attribute.
- For `index` attributes, you might need to specify both sizes.

- We expect specifying the size of embedded objects to be the most common use of either size, hence object sizes should have the shorter name.

B.1.2 Objects

All non-embedded objects have the operations `create` and `destroy`. If an object type `T` appears only as embedded references, target models obey an implicit

```
for T use forbid(create,destroy)
```

For embedded references (and objects in general), the `size` note controls the size of the object. The following are mutually exclusive.

- `bits`.
  ```
  for x.i use size(bits(n));
  ```
  where n is an integer and n>0.

- `bytes`.
  ```
  for x.i use size(bytes(n));
  ```
  where n is an integer and n>0.

- `words`. This depends on the machine word size, and so means different things on different systems. It is a way of giving a convenient, easily-processed unit.
  ```
  for x.i use size(words(n));
  ```

- `address`. The size of a word address. This depends on the machine word size, and so means different things on different systems. It is a way of giving a convenient, easily-processed unit.
  ```
  for x.i use size(address);
  ```
  If addresses of different kinds of objects are of different sizes, an IDL implementation may need to provide several other address-like implementation notes.

The `default` note specifies a default initial value for objects of a type. A typical use is to give a default value to attributes:

```
x => a : string;
for x.a use default("Fred")
```

Node creation procedures would use these to initialize the attributes, as would readers.

B.2 Types

B.2.1 Boolean

There are no special implementation notes for boolean. All object and reference notes apply. Many implementations will default boolean to `embed` and `size(bits(1))`.

We don't list any operations on type boolean, since we don't see any particular reason to restrict any of them. A good starting place for a list would be the operations on booleans in the assertion language.

B.2.2 Classes

The special representations on classes are:

- `enumeration`. This is a special case of `embed`.
- `nil`. This is also a special case of `embed`. No class may contain two different subclasses with a `nil` representation.

The only operations on classes are the `create` and `destroy` for any non-embedded objects, and `member`, which tests whether an object is a member of the class.

B.2.3 Integer

All object and reference notes apply. Many implementations will default integer to `embed`. In addition, there are the following specific implementation notes.

- Range.
  ```
  for x.i use range(n1,n2);
  ```
  N1 and n2 are integers, and n1 $\leq$ n2. The size must be large enough to hold all numbers in the range, including the sign.

- Bias. This is a number to be added to the stored representation to yield the actual integer value. The normal bias is zero.
  ```
  for x.i use bias(n);
  ```
  N is any integer; if you omit it, the default is the lowest value of the range. Biased representations permit smaller representations. For example,
  ```
  for x.i use range(250,260);
  for x.i use bias;
  ```
  allows a four-bit representation instead of a nine-bit representation.

We don't list any operations on type integer, since we don't see any particular reason to restrict any of them. A good starting place for a list would be the operations on integers in the assertion language.

B.2.4 Private Types

The primary implementation note for private types is `package`, which names the module (and optionally the type within the module) that implements the private type.

```
type T;
for T use package(pack,sometype)
```

says that type `sometype` of module `pack` implements IDL type `T`. If you omit the name after the dot, it defaults to the name of the private type. Thus given a private type `T`, the following are equivalent.

```
for T use package(pack);
for T use package(pack,T);
```

Some target models may require that you also specify the `size` implementation note for any private type. We don't list any operations on private types, since each will have its own operations, defined in the package that implements it.

B.2.5 Rational

Rationals allow the `range` and `bias` notes, as do integers; the limits and bias may be rational literals as well as integer literals.

The main additional notes specify classes of implementations. The following are mutually exclusive.

- Floating point. Specify the minimum decimal digits of precision via **use** `digits(n)`; n must be a positive integer.
- Fixed point. Specify the exact difference between successive elements via **use** `delta(n)`; n must be a positive rational. A fixed point rational with **use** `delta(1)` is equivalent to an integer.
- Pair of integers: **use** `pair`. Specify integer implementation notes (size, range, bias, and so on) for the numerator and denominator as arguments to `pair`, or independently as **use** `numerator(`*notes*`)` and **use** `denominator(`*notes*`)`.

We don't list any operations on type rational, since we don't see any particular reason to restrict any of them. A good starting place for a list would be the operations on rationals in the assertion language.

B.2.6 Sequence

The following sequence representations are mutually exclusive.

- `array(`*`integer`*`)`. Fixed length. If you supply an integer argument, the array length is a constant for all instances. Otherwise, the size is determined at allocation.
- `chained_array(`*`integer`*`)`. A linked list of arrays of elements. The integer gives the length of the individual arrays.
- `array_stack(`*`integer`*`)`. Standard array implementation of a stack, with index of top of stack. If you supply an integer parameter, it gives the array length for all instances. Otherwise, the size is determined at allocation.
- `array_queue(`*`integer`*`)`. Standard array implementation of a queue, with indices of first and last elements. If you supply an integer parameter, it gives the array length for all instances. Otherwise, the size is determined at allocation.
- `list`. This is a partially resolved representation. A translator would use default rules, possibly taking into account operation restrictions, to resolve it to `single_list`, `circular_queue`, `double_list`, `circular_deq`, or some other list representation.
- `single_list`. Non-circular singly linked list.
- `circular_queue`. Circular singly linked list; the type is a pointer to the last element of the list, allowing fast insertions at either end and fast removal from the front.
- `double_list`. Non-circular doubly linked list.
- `circular_deq`. Circular doubly linked list.

The `threaded` applies to any sequence with a list implementation; it runs the links through attributes of the nodes rather than in a separate object. This note limits how many such sequences each node can be on, since each requires a different collection of attributes. For singly-linked lists, it takes the name of a single attribute; for doubly-linked lists, it takes the names of two attributes, for forward and backward links (in that order).

There are many possible operations on sequences; not all representations support all operations. Possible operations include `concatenate` two sequences, `delete_front` (pop, dequeue), `delete_back`, `get_element` (fetch via subscript), `get_first` (top, front), `get_last`, `insert_front` (push), `insert_back` (append, enqueue), `length`, `set_element` (store via subscript), `set_first`, `set_last`. Sequence packages may also provide iterators to access each element of a sequence in order, and editors for walking over a sequence and adding and deleting elements in the middle. Operations on iterators might include `get_current` and `set_current`. Operations on editors might include those for iterators, plus `delete_current`, `insert_after` current element, and `insert_before` current element.

B.2.7 Set

There are many possible representations of sets. The following are mutually exclusive.

- `bitmap(`*`integer`*`)`. An array of booleans, indexed by values of the element type. This is only suitable for small sets of integers, fixed-point rationals, fixed-length strings (typically of length 1, for characters), or enumeration classes. If you supply the optional integer argument, it fixes the length of the array; otherwise the size is set at allocation.

- `flag`. This is only suitable for sets of elements from a class where there is only one such set in the system. The parameter names a boolean-valued attribute to use as the set membership flag.

- `hash_set(`*`integer`*`)`. A hash table, similar to those for the `hash` attribute representation. If you supply the optional integer argument, it fixes the length of the table; otherwise the size is set at allocation.

- `sorted_array(`*`integer`*`)`. A partially-filled array where you test for membership with binary search. If you supply the optional integer argument, it fixes the length of the array; otherwise the size is set at allocation.

- `unsorted_array(`*`integer`*`)`. A partially-filled array where you test for membership with linear search. If you supply the optional integer argument, it fixes the length of the array; otherwise the size is set at allocation.

B.2.8 String

The following representations are mutually exclusive.

- `varying(n)`. The string length may vary up to some maximum. If you supply the optional positive integer argument `n`, the maximum size is `n` for all such strings. Otherwise, each string has its maximum size fixed at allocation.

- `fixed(n)`. Once allocated, a string's size never varies. If you supply the optional positive integer argument `n`, the size is `n` for all such strings. Otherwise, each string has its size fixed at allocation.

- `string_common`. Strings may be of any size, but are allocated as part of a separately-managed string space. Strings in the SAIL language are of this type [vanL73].

Operations on strings include `concatenate`, `get_element`, `get_substring`, `length`, `set_element`, and `set_substring`.

B.3 Assertions

The `package` note for a user-supplied definition names the package that supplies the body for the definition.

B.4 Processes

The only process-wide note is **use** `generate`.

If you use a structure-specific note in a process, it supplies a default for all structures associated with the process.

B.4.1 Modules

Module notes affect aspects of the target model associated with the generated module. With a **for** clause, they apply to a particular module or structure. Without a **for** clause, they apply to all modules or structures in a process.

- Specify the target model. This normally implies the target implementation language; it would be typical to use a model with the same name as the language. However, some models support several languages at once.

```
use model(name)
```

- Specify the target implementation language. This normally implies a default target model.

```
use language(name)
```

- Prefix all names for implementations of nodes, classes, private types, and procedures in the module with a string to make them unique.

```
use prefix(XX_);
use prefix("Y$");
```

You can use a string where the target language allows characters in identifiers that IDL forbids.

B.4.2 Traversals

Writers and other iterators have a choice of what algorithm to use to decide what order to access elements of the structure.

- Traverse all objects in storage order.
  for *writer* **use** `iterator(storage)`

- Traverse all objects reachable from some node.
  for *write* **use** `iterator(reachable)`

B.4.3 Input and Output

Several notes specify detailed properties of the readers and writers. The following notes are mutually exclusive.

- `ascii`. Use an ASCII external form. For writers, you may supply a parameter to choose between nested and flat forms:
  for *writer* **use** `ascii(flat);`
  for *writer* **use** `ascii(nested)`

  Readers should read both nested and flat forms.

- `binary`. Use a binary external form. For writers, you may supply a parameter to choose between nested and flat forms:
  for *writer* **use** `binary(flat);`
  for *writer* **use** `binary(nested)`

  Readers should read both nested and flat forms.

- `repack`. The reader/writer pair in the connection will cooperate to repack an existing structure in memory. If the two internal representation happen to be identical, no work is necessary.

For ASCII and binary writers, the `compress` specifies how much to compress the output. The following arguments are mutually exclusive.

- `none`. No compression. Use the full-blown ASCII external form, and tag each attribute in a binary external form with a value corresponding to the attribute name. This allows for arbitrary rearrangement and repacking on input.
- `defaults`. Tag each attribute with the attribute name, as in `none`, but eliminate those attributes with default values.
- `tags`. Eliminate attribute tags, writing out attributes in a predefined order.

The `medium` note takes an argument specifying how to pass the encoded information from one process to another.

```
    for R use medium(file)
```

Both sides of a connection must specify (or default to) the same medium. The following media are mutually exclusive:

- `file`. Pass the information in an operating system file. The target model must give a way to specify the name of the file. This is the most sensible default for `ascii` and `binary` I/O.
- `memory`. Pass the information in memory; this is the default medium for `repack`. It makes less sense for `ascii` and `binary`, but is still possible.
- `message`. Use message-passing primitives.

Appendix C

History

IDL is an outgrowth of an earlier system called LG developed at Carnegie-Mellon University for the PQCC (Production Quality Compiler-Compiler) project [Leve80]. PQCC was a research project investigating the design and construction of optimizing compilers. Several of the project's research goals guided the design of LG:

- The PQCC compiler would be a multiple-phase program, to allow different researchers to work on different parts of the PQCC system independently.
- Phases would need to exchange program trees, flow graphs, and similar graph structures.
- The implementation language, BLISS-10 [Wulf71] (and later BLISS-36 [DEC77]), had no convenient means for declaring structured data.
- Debugging a phase often required examining some portion of the graphs produced by the phase. This meant there had to be some human-readable form for the graphs.
- A phase might need to be tested before some predecessor was working. Typically this would mean that a human would simulate the missing phase by hand. This meant there had to be a human-editable form for the graph structures.
- The wide variety of graph structures needed during the project made it highly desirable to automatically generate declarations and input/output routines, rather than writing each by hand.

This led to a generic graph reader/writer, a notation (never named) for describing nodes, and a program (the 'Require file generator') for generating BLISS data declarations and tables for the reader/writer. Between 1976 and

1981, when the project shut down, several people helped design and implement portions of the LG system, including David Dill, Gary Feldman, Paul N. Hilfinger, Steven O. Hobbs, Joseph M. Newcomer, and Wm. A. Wulf. Wulf was in charge of the PQCC project, and wrote generic set and list processing packages. Newcomer, Hilfinger, and Hobbs were responsible for the original design and implementation. Newcomer had primary responsibility for the Require file generator, and modified the existing BLISS debugger to interface with the reader/writer to print and modify nodes. Newcomer and Dill built the first binary reader/writer package during 1978-9.

During 1979 and early 1980 Intermetrics, Inc. developed its own variant of LG to support several languages, including Fortran, PL/1, C, and Pascal [Inte80]. Also during this period, the PQCC project published its representation scheme for Ada programs, TCOL_{Ada} [Scha79,Newc79,Bros80]. Experience convinced us that the input language for the Require file generator was a poor way to communicate such descriptions to other people. In addition, there was some interest in trying to be able to communicate between the Gandalf project [Habe79] and PQCC. Gandalf was written in C [Kern78] and ran on a VAX-11, a machine with 32-bit words, while PQCC was written in BLISS and ran on a DECsystem20, a machine with 36-bit words. Thus there was motivation to develop a replacement for LG that would permit communication between programs written in different languages on different hardware, with a more human-readable graph description language.

During the summer of 1980, Wm. A. Wulf, John R. Nestor, and David Alex Lamb began to work on IDL as a replacement for LG; they completed the initial design during December 1980. In January 1981 they met with Gerhard Goos' team from the University of Karlsruhe at Eglin Air Force Base to design DIANA [Goos83], an intermediate representation for Ada programs. DIANA contained ideas from TCOL_{Ada} and AIDA, Karlsruhe's intermediate representation. The DIANA definition was the first major use of IDL. David Lamb wrote the first IDL translator as part of his PhD research during 1981-2. The initial target for IDL was Bliss-36 using LG support. David A. Syiek produced a Pascal target model to support his Master's project.

When Wulf, Nestor, and Anita Jones founded Tartan Laboratories in 1981, they adopted Lamb's IDL translator as a key tool. The main players in building the Tartan technology were Kenneth J. Butler, Steven B. Byrne, Susan Dart, Edward N. Dekker, David Alex Lamb, Don Lindsay, Joseph M. Newcomer, John R. Nestor, Guy L. Steele, David A. Syiek, and Leland Szewerenko. Syiek and Dekker built the first target model for the `Gnal` language; Newcomer and Nestor built the second during 1983-4. These were the first real target models for a modern language that supported information hiding. Syiek and Dekker were the first to suggest the need for a Constant Structure Generator tool; Newcomer implemented it in 1984. Steele, Newcomer, and Nestor designed and built a highly optimized storage allocator and garbage collector.

Tartan supported other work on IDL, including a revision of the formal definition [Gian86], IDL-based extensions to attribute grammars [Nest83], and a portable runtime system.

Several other organizations built their own IDL translators, often as part of efforts to build Ada compilers using DIANA. Intermetrics has two such translators, and has built several compiler development tools on top of IDL. For example, BONSAI [Inte86] is a tool that aids in generating tree transformation phases; it uses IDL to describe the data structures. Snodgrass and his colleagues at the University of North Carolina in Chapel Hill have an extensive collection of IDL-based tools, including an IDL-to-C translator [Snod87]. They were the first group to make extensive use of IDL facilities other than structure declarations, such as process declarations and assertions. They are preparing a book describing their tools [Snod89].

In February, 1985, Intermetrics, Inc. sponsored a DIANA/IDL Workshop in Airlie, Virginia as part of a DIANA revision contract. At this meeting Ben Hyde showed Lamb the need to distinguish reference representations from object representations (Section 3.2).

During 1986-7 the Software Engineering Institute at Carnegie-Mellon University explored IDL as a tool for interfacing large sets of software tools. The project brought together IDL developers and users from industry, academia, and government. An SEI-sponsored workshop at Kiawah, South Carolina, May 19-21, 1986, brought together several practitioners with IDL experience. Participants in the workshop made several valuable suggestions about interface package design issues. One result of the workshop was a special issue of *SIGPLAN Notices* on IDL [Morg87].

Several other organizations have supported IDL research and development efforts, including the Defense Advanced Research Projects Agency (DARPA), the Software Engineering Institute of Carnegie-Mellon University, and the Natural Sciences and Engineering Research Council of Canada (NSERC). The members of IFIP Working Group 2.4 have given much helpful feedback over several years.

Bibliography

Ben M. Brosgol, Joseph M. Newcomer, David Alex Lamb, D. Levine, Mary S. van Deusen, and William A. Wulf, "TCOL.Ada: Revised Report on An Intermediate Representation for the Preliminary Ada Language," Technical Report CMU-CS-80-105, Computer Science Department, Carnegie-Mellon University (February 1980).

Lori A. Clarke, Jack C. Wileden, and Alexander L. Wolf, "Graphite: A Meta-Tool for Ada Environment Development," Technical Report COINS Technical Report 85-44 (March 1986).

Digital Equipment Corporation, "BLISS Language Guide" (1977).

Paola Giannini, "A Formal Semantics for IDL," Computer Science Department, Carnegie-Mellon University (1986).

Gerhard Goos, William A. Wulf, Art Evans, and Kenneth J. Butler, *DIANA: An Intermediate Language for Ada.* Springer-Verlag (1983). Lecture Notes in Computer Science #161.

M.J.C. Gordon, *The Denotational Description of Programming Languages: an Introduction.* Springer-Verlag, New York (1979).

A. N. Habermann, "The Gandalf Research Project," in *Computer Science Research Review*, Computer Science Department, Carnegie-Mellon University (1978-79).

Intermetrics, Inc., 733 Concord Ave., Cambridge, MA 02138, "Intermetrics LG System Description," Technical Report IR 536 (August 1980).

Intermetrics, Inc., 733 Concord Ave., Cambridge, MA 02138, "Compiler Retargeting Tools User's Guide," Technical Report IR-MA-623 (5 May 1986).

B. W. Kernighan and D. M. Ritchie, *The C Programming Language.* Prentice-Hall, Inc., Englewood Cliffs (1978).

Jerry Scott Kickenson, "An Interface Description Language Assertion Checker," Technical Report TR86-014, Computer Science Department, University of North Carolina (Chapel Hill) (1986).

David Alex Lamb, *Sharing Intermediate Representations: The Interface Description Language*, Ph.D. dissertation, CMU-CS-83-129, Carnegie-Mellon University (May 1983).

David Alex Lamb, "Implementation Strategies for DIANA Attributes," Technical Report 85-176, Queen's University Department of Computing and Information Science (November 1985).

David Alex Lamb, "IDL: Sharing Intermediate Representations," *ACM Transactions on Programming Languages and Systems* Vol. **9**(3):267-318 (July 1987).

David Alex Lamb, "Program Component Generator Project Implementor's Guide, Version 1.4," Technical Report Internal Technical Report ISSN-0836-0235-87-IR-01, Queen's University Department of Computing and Information Science (October 1987).

David Alex Lamb, "Generating Interface Packages, Readers, and Writers from IDL Descriptions," Technical Report 87-190, Queen's University Department of Computing and Information Science (October 1987).

David Alex Lamb, "Implementation Strategies for DIANA Attributes," *SIGPLAN Notices* Vol. **22**(11) (November 1987).

David Alex Lamb and Robin Dawes, "Testing for Class Membership in Multi-Parent Hierarchies," *Information Processing Letters* Vol. **28**(1):21-25 (May 1988).

Bruce W. Leverett, Richard G. G. Cattell, Steven O. Hobbs, Joseph M. Newcomer, Andrew H. Reiner, Bruce R. Schatz, and William A. Wulf, "An Overview of the Production Quality Compiler-Compiler Project," *IEEE Computer* Vol. **13**(8):38-49 (August 1980).

C. Robert Morgan, *SIGPLAN Notices Special Issue on the Interface Description Language IDL.* (November 1987). SIGPLAN Notices 22,11.

John R. Nestor and Margaret A. Beard, "Front End Generator System," in *Computer Science Research Review*appendix B , pages 79-92, Computer Science Department, Carnegie-Mellon University (1980-1981).

John R. Nestor, William A. Wulf, and David Alex Lamb, "IDL - Interface Description Language: Formal Description," Computer Science Department, Carnegie-Mellon University (June 1982). Draft revision 2.0. Available from the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA 15213.

John R. Nestor, Bhubaneswar Mishra, William L. Scherlis, and William A. Wulf, "Extensions to Attribute Grammars," Technical Report TL 83-86, Tartan Laboratories, Inc. (April 1983).

Joseph M. Newcomer, David Alex Lamb, Bruce W. Leverett, D. Levine, Andrew H. Reiner, Michael Tighe, and William A. Wulf, "TCOL.Ada: Revised Report on An Intermediate Representation for the DOD Standard Programming Language," Technical Report CMU-CS-79-128, Computer Science Department, Carnegie-Mellon University (June 1979).

Bruce R. Schatz, Bruce W. Leverett, Joseph M. Newcomer, Andrew H. Reiner, and William A. Wulf, "TCOL.Ada: An Intermediate Representation for the DOD Standard Programming Language," Technical Report CMU-CS-79-110, Computer Science Department, Carnegie-Mellon University (March 1979).

Karen Shannon and Richard Snodgrass, "Mapping the Interface Description Language Type Model into C," Technical Report SoftLab Document 24, Computer Science Department, University of North Carolina (Chapel Hill) (March 12, 1986).

Richard Snodgrass, "IDL Toolkit Release Notes (Version 3.2)," Technical Report SoftLab Document 33, Computer Science Department, University of North Carolina (Chapel Hill) (November 1987).

Richard Snodgrass, Karen P. Shannon, Jerry S. Kickenson, Michael A. Shapiro, Dean D. Throop, William B. Warren, David A. Lamb, John R. Nestor, and William A. Wulf, *The Interface Description Language: Definition and Use.* Computer Science Press, Rockville, MD (1989).

J.E. Stoy, *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory.* MIT Press, Cambridge (Massachusetts) (1981).

R. D. Tennent, *Principles of Programming Languages.* Prentice-Hall, Englewood Cliffs, NJ (1981).

Kurt A. van Lehn, “SAIL User Manual,” Technical Report Memo AIM-204, Stanford Artificial Intelligence Laboratory (July 1973). Also available as Stanford Computer Science Department Report STAN-CS-73-373, or from the National Technical Information Service, Springfield, VA 22151.

William A. Wulf, D. B. Russell, and A. Nico Habermann, “BLISS: a Language for Systems Programming,” *Communications of the ACM* Vol. **14**(12):780-790 (December 1971).