NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

CMU-CS-81-139

IDL - Interface Description Language

Formal Description

J. R. Nestor, W. A. Wulf, D. A. Lamb

August 1981

Carnegie-Mellon University
Computer Science Department

Copyright © 1981 J. R. Nestor, W. A. Wulf, D. A. Lamb

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Table of Contents

List of Figures

Preface

This report defines a general mechanism, IDL, by which structured data can be precisely specified. The objective of this specification is to permit the data to be communicated between programs, or parts of a single program, in a safe and efficient manner.

IDL grew out of work on the Production Quality Compiler-Compiler (PQCC) project at Carnegie-Mellon University [5]. An notation called LG (for *Linear Graph*) was used to describe the data structures passed between phases of the compiler [7]. LG had a number of drawbacks. It was difficult to use, and was strongly oriented towards the particular implementation language (BLISS [1, 8]) and host machine (the PDP-10) used by the PQCC project. Nonetheless, it was a very useful tool.

During 1979 and early 1980 a consensus developed at CMU that we needed to generalise the data definition language to simultaneously meet the needs of several different projects, written in different implementation languages on several different computer systems. Within CMU there were compiler-related projects, such as the Gandalf program development environment effort [3], which ran on systems quite different from the ones used by PQCC. During this same period the community of implementors of the Ada programming language developed a strong interest in being able to share intermediate program representations.

In late 1980 there were two major candidates for a common intermediate representation of Ada programs: $TCOL_{Ada}$, developed at Carnegie-Mellon, and AIDA, developed at the University of Karlsruhe. A meeting was held at SofTech, Incorporated, in December 1980, to discuss these two representations; at this meeting, it was decided to attempt to merge the two notations. A one-week design session was held at Eglin Air Force Base in January 1980. The outgrowth of this meeting was a new intermediate representation, Diana. Since there was a need to define Diana precisely, and since any intermediate language such as Diana is structured data, we concurrently defined IDL. The definition of Diana was then written using IDL.

This document provides a formal description of IDL. A companion document, the Diana manual [2], uses IDL to describe Diana.

Acknowledgements

Acknowledgements

The design of IDL owes a lot to the developers of the LG system used in the PQCC project: Rick Cattell, David Dill, Paul Hilfinger, Steve Hobbs, Bruce Leverett, Joe Newcomer, and Andy Reiner. The Diana design team which met at Eglin Air Force Base provided valuable comments on the initial design of IDL: Manfred Dausmann, Gerhardt Goos, Rich Simpson, Michael Tighe, and Georg Winterstein. Ben Hyde, Anita Jones, Mary Shaw, and Walter Tichy gave us helpful feedback on earlier drafts of this document. James Saxe uncovered several problems with earlier versions of the formal definitions.

rata for Revision 1.0

e external ASCII form is case sensitive. Please change all statements that it is not.

1 cover page change AUGUST 1981 to SEPTEMBER 1981

nd add a line that reads Revision 1.0

eplace pages 47 - 50

eplace equation on page 51 by

$$VALUE \triangleq BVALUE \cup IVALUE \cup SVALUE \cup RVALUE \cup NV \cup Set \cup Seq \cup PVALUE \cup \{delvalue, undefvalue, novalue\}$$

eplace equation on page 52 by

$$vset(rational) = RVALUE$$

eplace page 53

elete CREATECOMP operation from top of page 54

eplace equation on page 54 by

$$vset([seq,types]) = \{ \langle types,locs \rangle \mid \forall loc \in locs, loc \neq undeflocation \wedge loc.1 \in types \}$$

eplace precondition of MAKE on page 55 by

pre: $loc \neq undeflocation \wedge loc.1 \in s.1$

eplace equation on page 55 by

$$vset([set,types]) = \{ \langle types,locs \rangle \mid \forall loc \in locs, loc \neq undeflocation \wedge loc.1 \in types \}$$

eplace precondition of INSERT on page 56 by

pre: $loc \neq undeflocation \wedge loc.1 \in s.1$

eplace precondition of REMOVE on page 56 by

pre: $loc \neq undeflocation \wedge loc.1 \in s.1$

eplace post condition of EMPTYATTR on page 57 by

post: $\forall nn,an \in \langle name \rangle, atype(nn,an) = \emptyset$

Part One: Description of IDL

1. Introduction

This report defines IDL, a mechanism for specifying properties of structured data. The objective of this specification is to permit the data to be communicated safely and efficiently among related programs. Before considering the mechanism itself we shall briefly discuss the motivation which led to its design.

A programming environment consists of a number of programs that assist a programmer in the program construction, test and validation process. These tools include editors, debuggers, compilers, pretty-printers, test-case generators, various kinds of analysis aids, and so on. Many of these tools operate on some intermediate representation of the program: a form that is below the level of the source text. Some of them also need access to data that is derived from the source text, but not explicit in it: procedure call graphs, data flow graphs, symbol tables, and various semantic attributes. Finally, some of the tools will need to access data that is specific to the installation or target machine but not otherwise related to a particular program: tables that define coding or reporting standards, tables that define local pretty-printing conventions, tables of simulated on-line testing data, and so on. The kind of situation we envision is illustrated in Figure 1-1.

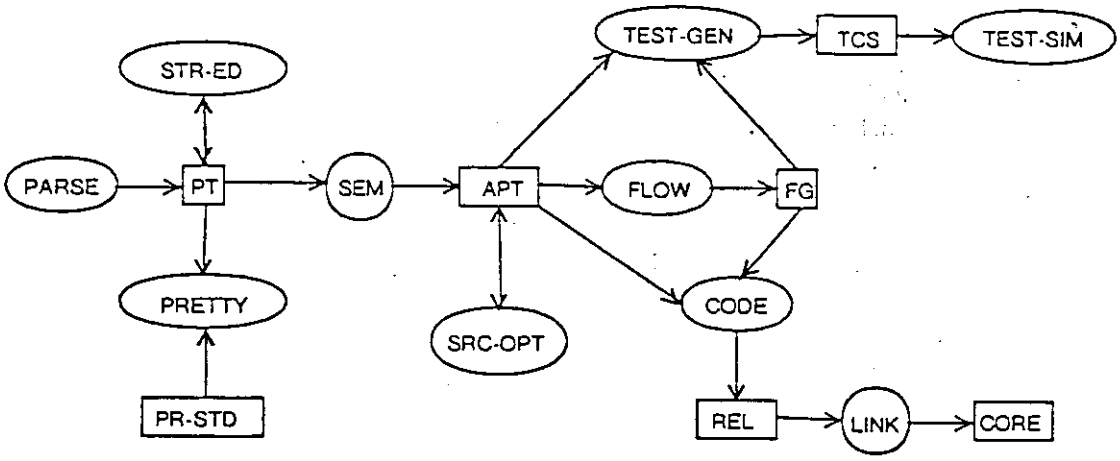


Figure 1-1: An Example Collection of Programs

In this figure rectangular boxes represent data and ovals represent programs; both boxes and ovals contain labels to suggest their roles. So, for example, a parser, PARSE, produces a parse tree, PT. A pretty-printer program, PRETTY, accepts PT and produces a listing using conventions defined in a database called PR_STD. A screen-oriented language-based editor, STR_ED, operates on the parse tree and produces another valid parse tree. A semantic analyzer, SEM, generates an attributed parse tree, APT, from the simpler tree generated by the parser and/or editor. Several tools operate from the attributed tree: FLOW creates a flow graph, FG; a source level optimizer, SRC_OPT, performs program transformations that are again

represented as valid attributed trees; a test-case generator, TEST_GEN, uses the attributed tree as well as the flow graph to produce a form, TCS, that can be used by the test-case simulator, TEST_SIM; finally, a code generator, CODE, uses the flow graph and attributed parse tree to generate a relocatable file, REL. A linker, LINK, converts relocatable code into an executable core image, CORE.

In order to work together harmoniously, the various programs in this example must have a precise and compatible definition of the data structures they use to communicate with each other. The primary purpose of IDL, then, is to provide such a definition. To meet this primary objective we must meet some secondary ones as well; these include

- *precision*: The IDL definition must be sufficiently precise to be used as a formal specification by those who are writing programs to process the data.

- *representation independence*: The IDL specification must not unduly constrain the internal representation of the data. Individual tools must be able to use internal representations that reflect their special processing requirements.

- *language independence*: The IDL mechanism must not be restricted to specifying data structures to be manipulated by a single target language. The tools in a programming environment may be written in different languages, and IDL must not preclude this.

- *maintainability*: The tools in a programming environment, like programs in general, will be developed incrementally and will be enhanced on the basis of experience using them. The various data structures through which they communicate will consequently also evolve. To retain compatibility in the face of this evolution, IDL must provide both humane and secure means for coping with changes.

- *communication form*: It must be possible to communicate data described in IDL between arbitrary programs and, indeed, between arbitrary computers. To support this requires at least one standard representation of the data and the ability to map between this form and the internal one chosen by specific tools. We choose to make the standard form have an ASCII manifestation to maximize its portability.

1.1. The Nature of an IDL Specification

Diagrams such as that in Figure 1-1 may be helpful in illustrating the relation between data and the programs which process it, but they are totally inadequate as a specification technique. In fact, one must be very careful not to read too much into such a diagram. It would be easy, for example, to infer that each of the boxes representing data is a file or that each of the ovals is a separate program. Neither of these is intended! It might also be inferred that there is a single internal representation for the data denoted by a box. This is also incorrect. To meet the objective we have set for IDL we need a specification technique that allows all of these things, as well as many other possibilities.

We want an IDL specification to describe a data structure without forcing a particular representation on the structure. We want individual instances of structures satisfying the specification to be implemented in a way that is appropriate for the particular program, or portion of program, that manipulates the data. The well-known methodology of *abstract data types* has the characteristics we want IDL to have.

The view that we shall adopt is that each box in Figure 1-1 denotes an instance of an abstract data type about which we can make various *assertions*. Each oval denotes an instance of an abstract process, which accepts one or more instances of a data abstraction as its 'input' and yields instances of other abstractions as its 'output'. In effect, the boxes in Figure 1-1 can be viewed as input-output assertions (or pre- and post-conditions) on the 'ovals'. For example, we can specify the effect of the semantic analyzer, SEM, as:

PT { SEM } APT

That is, if the input to SEM satisfies the definition of PT, then its output will satisfy APT. Similarly, we can define the effect of the code generator, CODE, as:

APT ∧ FG { CODE } REL

That is, if the inputs to CODE satisfy APT and FG, the output will satisfy REL. Saying it another way, the input to CODE must satisfy both specifications APT and FG, and the output of CODE is guaranteed to satisfy REL.

This view of the diagram in Figure 1-1 is obviously very abstract. For pragmatic reasons an implementation of the various programs in a specific situation will need to be concerned with lower level representation details, and later chapters of this document will deal with these legitimate concerns. For the moment, however, we will stick with the abstract view for several reasons. First, it provides the basis for the level of precision we are seeking. Second, it provides complete representation and language independence. Finally, coupled with a well-engineered specification technique, it allows for easy maintenance, and hence ensures compatibility in the face of evolution. We will later show how the abstract view taken here can be mechanically mapped into efficient implementations.

1.2. The Abstract Model

As noted above, we shall view each of the boxes in Figure 1-1 as an abstract data type; data input to the programs represented by the ovals are instances of these types. The first step in an IDL specification will be to define the abstract types under discussion.

An *abstract data type* consists of a set of *values* (the *domain* of the type), and a set of *operations* on these values. Any specification of an abstract type must define both of these; in IDL we choose to use the *abstract modeling* technique for doing this. In this technique one specifies the domain of the type in terms of

previously defined mathematical entities; the operations of the abstract type are then specified in terms of their effect on these entities.

We have chosen to require all specifications written with IDL to use the same model. This implies that the model must be a very general one, but it must have straightforward and efficient implementations. We have chosen typed, attributed directed graphs as our model. Informally, this domain is a collection of *objects*. Each object has a type, a location, and a value. One category of types in the model are *node types*[1]. The value of a node object is a collection of *attributes*; the particular attributes associated with a node object are a property of its type. No two attributes of the same node type have the same name; each attribute of a node object has an associated location. Attributes are also typed; the objects form a graph because some of the attributes may reference other objects.

Instances of these graphs are commonly represented by diagrams such as Figure 1-2. In this diagram, circles denote objects. Each attribute of a node object is denoted by an arrow ('arc') to the object that is its value. Node types are written within the node object.

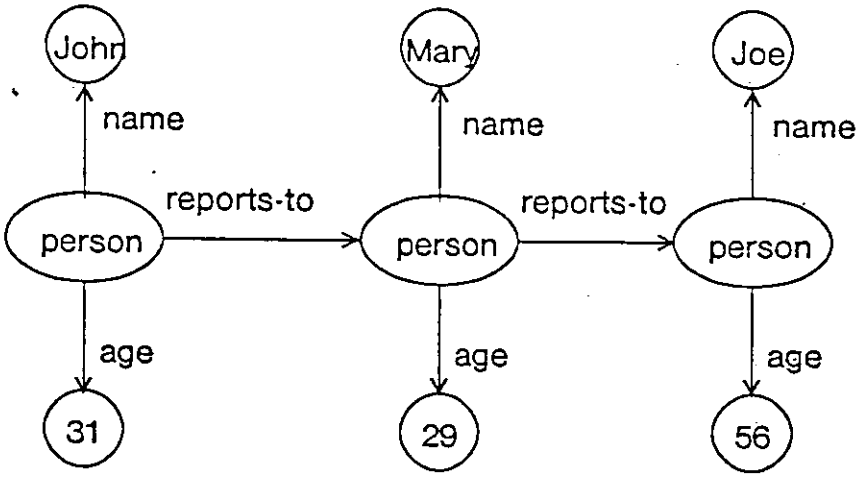


Figure 1-2: An Instance of a Typed, Attributed Directed graph

[1] The other categories are scalars (integers, rationals, booleans, and strings), sets, and sequences.

lthough diagrams such as that in Figure 1-2 may aid one's intuition, they are far from being sufficiently recise for our purposes. Again, such diagrams can be dangerous if they suggest too much to the reader. It vould be easy, for example, to assume that each of the node objects in such a diagram is to be implemented y a record in some programming language with components to represent its attributes, and that the links are epresented by pointers. This is certainly one possible implementation -- but it is not the only one, and is efinitely not the best one under many circumstances. For instance, while some node objects might be epresented as records, others which are referenced only once might be "up-merged" to become components f the records corresponding to the node objects that reference them. Remember, throughout this document ie graphs we are discussing are *abstract models* of abstract data types being defined. They in no way imply n implementation.

1.3. On the Structure of this Document

he remainder of this document defines IDL. Chapter 2 defines the syntax and semantics of an IDL pecification. Chapter 3 describes the sublanguage used to make assertions about components of an IDL pecification. In Chapter 4 we discuss an external (ASCII) representation of the data defined by an IDL pecification; this representation is essential for communication of data between computing systems. Finally, n Chapter 5 we outline how the abstract specification of IDL can be mechanically converted into a concrete mplementation.

art II gives a formalization of IDL. Chapter 6 describes the notation used in the formal model. Chapter 7 ives the basis of the mathematical graph model we use to describe IDL. Chapter 8 gives the type model. A ormal model of IDL structure specifications is given in Chapter 9. Chapter 10 formalises the external epresentation. A later version of this document will include a formal description of the assertion language.

2. IDL Definition

A complete IDL definition may be thought of as a precise definition of the intuitions captured by diagrams such as that in Figure 1-1. It defines both the data denoted by boxes and the processes or programs denoted by ovals. Data is viewed as an instance of an abstract data type about which various assertions can be made. Processing components are viewed as accepting one or more data types which satisfy these assertions and establishing others. We shall refer to descriptions of data as *structures* and to descriptions of programs as *processes.*

Although IDL takes a relatively abstract view of data and programs, we intend it to be a *very* practical tool in the construction and maintenance of collections of real systems. This implies that we must be ultimately concerned with implementation issues and with the paramount need to keep the formal IDL specification synchronized with implementations of it. More will be said about this later; for the moment we will simply assert that we intend for implementations to be mechanically derived from the formal definition, thus forcing synchrony. To do this implies that information about the intended implementation strategy must be present in the IDL definition. It must be present, however, in a manner that is disjoint from the logical portion of the specification; that is, we want a separation similar to the separation of specification and implementation in data abstraction languages.

In order to separate the logical properties of structures and processes from the implementation-specific properties, we split the definitions into two categories. *Abstract* structure and process definitions describe logical properties; *concrete* definitions provide implementation-specific properties.

We will also occasionally speak of *structure instances* and *process instances.* A structure instance is a particular data structure that meets the assertion represented by a particular structure definition. A process instance is a particular program that fits a particular process definition. We will sometimes speak about a structure or process when we mean "all structure instances satisfying some structure specification" or "all programs satisfying some process specification"; the meaning should be clear from context.

An IDL specification, then, contains four kinds of information:

- *Abstract Structure Specifications* -- Here we define the structures in terms of the abstract model (typed attributed directed graphs) discussed earlier. Each abstract structure specification defines the domain of a single abstract data type by giving the node types that can be used for objects in the domain. Defining a node type involves specifying the names and types of its attributes. A structure specification can also include assertions that specify constraints on instances of the structure. This level of data specification makes no commitment to representational details.

- *Concrete Structure Specifications* – Here we provide details of the representation of abstract

UNIVERSITY LIBRARIES
CARNEGIE-MELLON UNIVERSITY
PITTSBURGH, PENNSYLVANIA 15213

structures. For any particular abstract structure there may be many corresponding concrete structures. Concrete structure specifications can be organised into a hierarchy, with lower levels of the hierarchy containing more representation-specific information than higher levels. For each concrete structure specification which satisfies certain constraints specified later in this chapter, there is a standard external ASCII representation for the data described by that specification.

- *Abstract Process Specifications* -- Here we define each of the abstract processes (the 'programs'), in terms of what abstract structures they expect as input and what abstract structures they produce as output. These specifications attempt to capture the logical properties of a program without unduly constraining implementations.

- *Concrete Process Specifications* -- Here we provide implementation-specific details for the abstract processes. Information in this section includes bindings of abstract structures from the abstract process specifications to particular concrete structures, and restrictions on the set of operations the process may perform on the data.

In each of these specifications, IDL provides notation to describe certain structural properties of the component being specified. In addition an extensible assertion language is defined for expressing properties other than those captured by the structural and typing notation[2].

Although we intend that IDL be processable by machine, its most important use is to communicate specifications among people. IDL allows a great deal of flexibility in the way specification is written. Order of specifications is never significant; portions of declarations may be written separately and merged by the IDL processor. The order in which the rules are written, and the use of comments and indentation is very important for human understanding. Various orders and styles will make good sense in certain contexts. Unfortunately, sloppy use, poor mnemonics, and poor factorization of the specification can all detract from readability. We urge the wise use of these features.

Two of the operations defined for each structure are reading and writing external representations. The external ASCII representation is intended to allow for communication among arbitrary tools, written in arbitrary languages for arbitrary machines. Within a particular host environment there may also be several external binary representations used to communicate between tools written in different languages but running on the same machine. Programs written in the same language on the same machine may be able to communicate at the internal representation level as well.

The following sections define IDL. The syntactic definition of IDL is given in an extended BNF. Angle brackets ('<>') surround the name of a non-terminal. Braces ('{}') are used to group elements of a production; a trailing asterisk ('*') indicates zero-or-more occurrences; a trailing plus ('+') indicates one-or-more

[2] We expect that least some of these assertions will be automatically checked when data is read from (written to) external media. The extent to which this is done is implementation specific and may be disabled under certain circumstances.

occurrences; a trailing question mark ('?') indicates an optional item. A special lexeme such as a brace is included as a terminal by prefixing it with a double-quote mark ("). This notation is a slight simplification of the input language for the CMU Front End Generator [6].

2.1. Structure of an IDL Specification

An IDL specification consists of a sequence of structure and process specifications.

```
<specification> ::= { <decl> }+

<decl> ::= <structure decl> | <process decl>

<structure decl> ::= <abstract structure decl> | <concrete structure decl>

<process decl> ::= <abstract process decl> | <concrete process decl>
```

The declarations are not required to be in any particular order, and there may be more than one of each of them. This permits one to group related portions of a specification in ways that enhance readability.

The following lexical conventions are observed in an IDL specification:

1. A comment is introduced by double hyphens, '--', and terminated by the end of the line on which they occur.

2. The notation is 'case sensitive'. That is, identifiers with identical spelling except for the case of their letters are considered distinct[3].

3. Reserved identifiers in the IDL syntax have the first letter of each word capitalized, and all other letters in lower case. E.g., '`Structure`', '`ForAll`'.

4. Names (identifiers) consist of a letter followed by a sequence of letters, digits, or underscore characters.

2.2. Abstract Structure Specification

An abstract structure specification is divided into a set of structural constraints and a set of assertions. Structural constraints specify the node types that comprise the structure, together with their set of possible attributes. Assertions capture all the other interesting properties of the structure.

---

[3]Case sensitivity is a questionable language property; in this case it was adopted *only* to support the needs of the Diana description [2]. Diana's node, class, and attribute names are taken directly from the formal definition of Ada, which is case sensitive. We would gladly consider a modification of the formal definition that removed its case sensitivity and thus removed the need for this property in IDL.

```
<abstract structure decl> ::= Structure <name> Root <name> Is
                { <name list> Except }?
                { <abstract structure stmt> ; }+
                End

<name list> ::= <name> { , <name> }*

<abstract structure stmt> ::= <production> | <type decl> | <without clause>
                | <assertion>
```

Each abstract structure declaration defines a new abstract structure whose name follows the keyword `Structure`. Each abstract structure must have a distinct name. The `<name>` following the `Root` keyword names a class (see below) which is the type of the root node of the data structure; the root node is a distinguished object from which all others in the structure can be reached. An abstract structure can be specified in one of two ways.

1. As a new abstract structure. In this case the the `<name list> Except` clause is omitted and no `Without` clauses are permitted.

2. As a modification of other abstract structures. In this case the `<name list>` between the `Is` and `Except` keywords names the other abstract structures. The new structure is defined by copying and editing the old structures as described in Section 2.2.2.

The order in which `<abstract structure stmt>`'s appear is not significant.

2.2.1. Productions and Type Declarations

Productions and type declarations define structural constraints. Node productions define names and types of attributes for each node type. Type declarations define *private types*, which are types whose structure is not specified within the abstract structure specification. *Classes* are names used as abbreviations for collections of node types; when used as types for attributes, they indicate that the attribute may reference objects of any of the node types in the class. For each node type, a class of the same name is implicitly defined. Private type names and node type names must all be distinct.

```
<production> ::= <class production> | <node production>
```

The `::=` form of production is used to define class names.

```
<class production> ::= <name> ::= <name> { | <name> }*
```

Here the `<name>` that appears to the left of the `::=` is defined to be a class name. The names to the right of the `::=` must be class names. The new class consists of the union of all node types that are in any classes named on the right hand side. The same class name may appear on the left of several `::=` productions. In this case, the class consists of the union of the node types defined in all such productions. Class names may not depend upon themselves in a circular fashion involving only `::=` productions.

The => form of production is used to associate sets of attributes with node types. Each attribute is given a name and a type.

```
<node production> ::= <name> => { <attribute> { , <attribute> }* }?

<attribute> ::= <name> : <type>
```

The <name> to the left of the => is a class name. This <name> is defined as a node type name if it is not defined elsewhere as a class name (that is, on the left hand side of a ::= production). The <attribute>'s to the right of the => define a set of attributes that are to be associated with all of the node types belonging to the class whose name appears on the left. The same class name may appear on the left of several => productions. The attributes of a node type are the union of the attributes specified for all classses which contain the node type. The attributes of a node type must all have different names; however, attribute names need not be disjoint from node, class, and private names. Different node types may have attributes of the same name. Attribute types are discussed below (see Section 2.2.3).

The type declaration is used to define private names.

```
<type decl> ::= Type <name>
```

Private types name implementation-specific data structures that are inappropriate to specify at the abstract structure level. For instance, an abstract structure specification describing a compiler's parse tree might wish to include information in each node object about the position in the source file corresponding to that object. The notion of what constitutes a source position might be quite different in different environments.

2.2.2. Defining Abstract Structures in Terms of Other Abstract Structures

When an abstract structure declaration has a `Is <name list> Except` clause it is defined in terms of the other abstract structures whose names appear between the `Is` and the `Except`. The new abstract structure is derived in a three step process:

1. Copying. - All productions, type declarations, and named assertions from all of the abstract structures whose names appear after the `Is` are copied. Information duplicated in several abstract structres is copied only once. Specifically,

    - If there are two ::= productions with the same left hand side in two abstract structures, then each alternative that appears in both is copied only once.
    - If there are two => productions with the same left hand side in two abstract structures, then each attribute with the same name and type that appears in both is copied only once.
    - If there are two or more <type decl>s for the same type name that appear in two abstract structures, then only one is copied.
    - If two assertions have the same name, only one is copied. Unnamed assertions are not copied.

2. Deletion. - The `Without` clauses described below are used to delete some parts of the result of the

copy step.

3. Additions. - The productions, type declarations, and assertions specified as part of the new abstract structure are added to the result of the deletion step.

The `Without` clause is used to specify deletions.

```
<without clause> ::= Without <without item> { , <without item> }*

<without item> ::= Assert <name>

<without item> ::= <name>

<without item> ::= { <name> | * } { => | ::= }  { <name> }?
```

If the `Without` clause contains multiple `<without items>` then it is equivalent to a sequence of `Without` clauses, one for each `<without item>`. The `::=` and `=>` forms of the `Without` clause remove the class name or attribute name (respectively) given on the right from those productions with the same left hand side. If no name appears to the right of the arrow then all productions of the corresponding type (`::=` or `=>`) with the specified left hand side are deleted. If the left hand side is an asterisk (`*`) then this is equivalent to replicating the item for all names that appear on the left hand sides of the specified kind of production. The `<name>` form of the `<without item>` removes the provate type with the specified name. The `Assert <name>` form removes the assertion with the specified name.

All of these rules are entirely syntactic; no semantic information is used in the editing process. It is therefore possible to convert a node name to a class name by adding a `::=` production with the node name on its left hand side. It is similarly possible to convert a class name to a node name by deleting all `::=` productions with the class name on the left hand side.

2.2.3. Basic Types

In this section we define the set of permitted attribute types.

```
<type> ::= Boolean | Integer | String | Rational | Set Of <type> | Seq Of <type> | <name>
```

These basic types are:

1. `Boolean` -- the boolean type with values are true and false.

2. `Integer` -- the 'universal integer' type.

3. `String` -- ASCII strings. Any ASCII character may be represented. This includes printing characters, blanks, and non-printing control characters.

4. `Rational` -- the 'universal rational number' type. This type includes all values typically found in computer integer, floating point and fixed point types.

5. `Set Of <type>` -- An unordered collection (set) of values of `<type>`. Duplication of values (i.e. multisets) are not permitted.

6. `Seq Of <type>` -- An ordered collection (sequence) of values of `<type>`.

7. `<name>` -- where `<name>` is a private name from a `<type decl>`. The set of values for this type is defined by the package that implements it.

8. `<name>` -- where `<name>` is a class name. A value of this type is a node object whose type is one of the elements of the class.

There are no enumeration types *per se*; a class of node types, all of which have no attributes, can serve this purpose.

2.2.4. Example

Suppose that we wish to define a data structure to represent the abstract syntax of arithmetic expressions involving simple integer variables. First we provide the abstract structure definition:

```
Structure PT Root exp Is

-- First we define two node types; 'leaf' objects appear at the leaves
-- of the expression tree and 'tree' objects appear at its interior.
-- The class 'exp' is an abbreviation for either of these types.

        exp ::= leaf | tree;


-- Second  we define some node types that serve as enumeration literals.

        oper_name ::= plus | minus | times | divide;  -- operator names
            plus => ; minus => ; times => ; divide => ;

        context_name ::= value | flow;                -- expression context
            value => ; flow => ;


-- Finally we define the attributes associated with the various node types.

        leaf => value: Integer;

        tree => left: exp,
                right: exp,
                op: oper_name;

        exp =>  context: context_name;

    End
```

Although this example is extremely simple, it illustrates several things about IDL. As can be seen, only leaf nodes have an integer 'value' attribute. Only interior nodes of the tree have 'left', 'right', 'op' and attributes. The 'left' and 'right' attributes are references to 'exp' nodes -- that is, to either leafs or trees. The 'op' attribute is a reference to an object of one of the node types 'plus', 'minus', and so on. Since these node types have no

attributes they may be thought of as elements of an enumeration type. The null => productions for 'plus' etc. were needed to define them as node types; node type names are never implicitly defined. Both leaf and tree nodes have the 'context' attribute; this is indicated by the use the class name 'exp', which is a shorthand for 'leaf' and 'tree'.

The following example shows how an abstract structure may be defined in terms of a previously defined abstract structure.

```
Structure APT Root exp Is PT Except

        Without Leaf => Value;

        -- We define variable_sym as a private type name.
        -- It will serve as a "symbol table entry".

        Type variable_sym;

        leaf => definition: variable_sym;

        exp => next : exp;

    End;
```

The abstract structure APT is defined in terms of our previous example abstract structure, PT. Like PT, APT's root must be a tree or a leaf. The 'value' attribute of leaf nodes has been deleted in APT, but two new attributes have been added: leaf nodes now have a 'definition' attribute and both tree and leaf nodes have a 'next' attribute. The type of the definition attribute is the private type 'variable_sym'.

2.3. Abstract Process Specification

An abstract process specification defines the input and output data structures of a program.

```
<abstract process decl> ::= Process <name> Is { <abstract process stmt> ; }+ End

<abstract process stmt> ::= <pre stmt> | <post stmt> | <assertion>
```

The `<name>` of the abstract process follows the keyword `Process`. All abstract process names must be distinct from each other.

```
<pre stmt> ::= Pre <port list>

<post stmt> ::= Post <port list>

<port list> ::= <port decl> { , <port decl> }*

<port decl> ::= <name> : <name>
```

The `Pre` and `Post` statements are used to specify *ports*, which are formal input and output parameters of a process. Each `<port decl>` specifies a port name (before the ':') and an abstract structure name (after the ':'). All the port names of an abstract process must be disjoint. The abstract structure associated with a port serves as a precondition (postcondition) of the data structure bound to the port. These preconditions and

ostconditions are expected to hold only before or after the execution of any instance of the abstract process.

Assertions in an abstract process declaration are used to express relationships between two or more ports.

2.3.1. Example

The abstract process PT_compare takes two parse trees, primary_pt_port and secondary_pt_port, and produces a data structure, dt_port, describing the way one parse tree differs from the other.

```
Process PT_compare Is

    -- a program that compares two PTs and produces an annotated
    -- tree

    Pre primary_pt_port : PT, secondary_pt_port: PT;
    Post dt_port : DT;
End
```

2.4. Concrete Structure Specifications

Concrete structure specifications provide implementation-specific information about abstract structures.

```
<concrete structure decl> ::= Concrete Structure <name> Is <name> With
                { <concrete structure stmt> ; }+
                End

<concrete structure stmt> ::= <type rep> | <production> | <assertion>
```

The name after the `Structure` keyword is the name of the new concrete structure. The name after the `Is` keyword is the name of an abstract or concrete structure from which the new concrete structure is derived. The new concrete structure specification contains all of the information of the old, together with new specifications given by the `<concrete structure stmt>` list following the `With` keyword.

2.4.1. Type Representations

A concrete structure specification can contain internal type representations and private type representations.

```
<type rep> ::= <internal type rep> | <private type rep>
```

An internal type representation can be used to specify a private type that is to be used to implement some existing attribute type.

```
<internal type rep> ::= For <type reference> Use <type>

<type reference> ::= <name> . <name> { ( * ) }*
```

The first name in the type reference must be a class name. The name after the dot must be the name of an attribute declared in some => production for the class. The parenthesized star forms can be added to descend

through sets or sequences to their element type. The name after the Use is the name of a private type which is then used to represent the specified attribute. An IDL implementation may create a set of predefined private types with standard implementations. For instance, a particular system might have a `List_sequence` private type and an `Array_sequence` private type which could be used in a concrete structure declaration to specify implementations of various sequence-valued attributes.

An IDL implementation may extend the syntax of the `<internal type rep>` to provide additional implementation-specific details not covered here; such extensions must be done in a way that is compatible with the rest of the IDL syntax.

A private type representation may be used to define the way in which a private type is to be represented externally, and the package in which the internal representation of the private type is defined.

```
<private type rep> ::= For <name> Use <private rep>

<private rep> ::= <name> { . <name> }? | External <type>
```

The `<name>` after the `For` must indicate a private type. The `<type>` following the `External` keyword may be any of the predefined types outlined in Section 2.2.3, or a node type. The private type will be represented externally as if it had been the indicated type. The `Use <name>` form gives the name of a package that defines the private type.

2.4.2. Productions in Concrete Structures

In order to give external representations for some private types, it may be necessary to introduce new node types not defined by the abstract structure from which a concrete structure is defined. A concrete structure specification may include `::=` and `=>` productions for this purpose. However, the only names which can appear on the left hand sides of such productions are private type names, or new node type names and class names introduced in the concrete structure specification. Productions here cannot add new attributes to node types defined in the abstract structure specification, nor can they add node types to classes defined in the abstract structure specification.

2.4.3. Example

This example provides a concrete structure for the APT structure discussed in earlier sections. A user-supplied package called 'variable_package' defines the 'variable_sym' type. In the external representation an object of this type is represented as a node with an integer-valued attribute and an expression-valued attribute.

```
Concrete Structure particular_APT Is APT With

   -- we provide a specification for the variable_sym private type of APT
```

```
    For variable_sym Use variable_package;
    For variable_sym Use External variable_external_rep;

     variable_external_rep =>
         usage_count: Integer,
         original_def: Exp;

End
```

2.4.4. Concrete Structure Specification Hierarchies

Specifying concrete structures in terms of other concrete structures organises them into a hierarchy, with lower levels of the hierarchy being more implementation-specific than higher levels. There are two interesting boundaries in any such hierarchy:

- The *externally adequate* level. At this point, sufficient information has been provided to define an external representation for all instances of the structure. This level is reached when a concrete structure supplies a representation for all types defined in the abstract structure from which it is descended. See Chapter 4 for a discussion of external representations.

- The *internally adequate* level: At this point, enough information is present specify internal representations for all node types and attributes defined in the abstract structure. Internal representations for types may be given by naming packages which define the types; this may be done for the predefined types as well as for user-defined types.

A structure can be internally adequate without being externally adequate, if implementation packages are given for private types without giving external representations. The reverse is not possible, since an external representation implies a default internal representation if no specific internal representation is given.

2.5. Concrete Process Specifications

A concrete process specification gives implementation-specific properties of processes.

```
<concrete process decl> ::= Concrete Process <name> Is <name> With
                { <concrete process stmt> ; }+
                End

<concrete process stmt> ::= <port assoc> | <restriction> | <group decl> | <assertion>
```

The first `<name>` after the `Process` keyword is the name of the new concrete process. The `<name>` after the `Is` keyword is the name of an existing abstract or concrete process from which the new one is to be derived. As with concrete structures, concrete processes can be organised into hierarchies, with lower levels binding more details than higher levels.

2.5.1. Port Associations

A port association names a particular concrete structure which specialises the abstract structure associated with the port in the abstract process from which the concrete process was ultimately derived. The concrete structure must be ultimately derived from the abstract structure specified in the port declaration of the abstract process.

```
<port assoc> ::= For <name> Use <name>
```

The first <name> indicates a port of the abstract process. The second <name> indicates a concrete structure.

2.5.2. Group Declaration

When a process instance produces an output data structure it can do so in one of two ways. Either it modifies some combination of its input data structures 'in place', or it creates a new data structure. In the first case there must be an intimate relationship between the representations of the input structures and the output structures, while in the second they can be decoupled. The Group construct captures the notion that a group of input and output structures may be represented as a single data structure within a program.

```
<group decl> ::= Group <name list> Inv <name>
```

The <name list> consists of names of input and output ports of the abstract process from which the concrete process is derived. The <name> after the Inv keyword gives the name of a concrete structure, which must declare all the node types and attributes in all the port structures. The structure may declare additional node types and attributes that the process needs internally in order to perform its work. Inv is short for 'invariant;' the concrete structure serves as an invariant assertion about the process in the same way that structures associated with ports provide preconditions and postconditions of the process.

2.5.3. Restriction of Operations

Restriction specifications provide information about the operations a concrete process is allowed to perform.

```
<restriction> ::= Restrict <name> To <oper list>

<oper list> ::= <oper> { , <oper> }*

<oper> ::= <node oper> | <attribute oper>
```

The <name> following the keyword Restrict must be a class name. The operation list gives the set of operations that are permitted on objects of the node types in the class, and operations permitted on attributes of the objects.

The node operations are those that are used to create or destroy whole node objects.

```
<node oper> ::= Create | Destroy
```

Attribute operations apply to the attributes of node objects.

```
<attribute oper> ::= { Fetch | Store} ( <name list> )
```

The names in the `<name list>` must be the names of attributes of the node of this restriction specification.

A complete list of node and attribute operations is implementation-specific. An implementation may extend IDL by adding additional operations to these lists; the ones listed above are the minimum which must be supported.

2.5.4. Example

Continuing the example from Section 2.2.4 we can define the process that maps from the abstract structure PT to the abstract structure APT as follows.

```
-- first we define an abstract structure that describes the local data of the
-- process's package

Structure PT_to_APT Root Exp Is APT Except

        tree => tempattr: Integer               ;
End

-- next we provide a concrete structure for PT_to_APT

Concrete Structure c_PT_to_APT Is PT_to_APT With
    For variable_def Use variable_package;
End

-- next we define the process

Process PT_to_APT Is

    Pre  Inport:  PT            ;
    Post Outport: APT           ;

End

-- Finally we define the concrete process.  It augments PT to produce
-- c_PT_to_APT, modifies it 'in place', and produces APT.

Concrete Process p_PT_to_APT Is PT_to_APT With

    Group Inport,Outport Inv c_PT_to_APT;

    Restrict exp To
        Create, Destroy,
        Fetch( value, tempattr );
    Restrict tree To
        Fetch( left, right, op),
        Store( tempattr );
        -- and so on

End -- p_PT_to_APT
```

The concrete process inputs a parse tree from Inport, and outputs an attributed parse tree to Outport. Internally its data structures are represented as described by structure c_PT_to_APT, which is an APT with

an additional 'tempattr' attribute in all tree nodes. The `Group` specification indicates that the representation of the parse tree is modified 'in place' to produce the attributed parse tree.

. Assertion Language

e domain of a structure is expressed using productions and private type definitions. The assertion language ows the expression of additional restrictions on a structure. The assertion language also can be used with cesses to relate the preconditions on the input ports to the postconditions on the output ports. Finally, the ertion language can be used with concrete processes to state invariants on a groups.

s useful to consider two major kinds of assertion that can be made.

- Value assertions: These assertions can be used to further limit the domain of some value (e.g. restrict an integer value to some specified range) or to express relationships between values (e.g. require that one integer value always be less than a second integer value).

- Object assertions: These assertions can be used to express structural properties beyond those captured by productions. These structural properties can be either local (e.g. require that two attributes reference the same object) or global (e.g. require that some set of nodes and attributes have the form of a tree).

practice, an assertion may actually express a combination of value and object properties.

any operations of the assertion language are in many cases distinguished based on whether they apply to lues of objects or to the objects themselves.

- Value operations: The form a=b compares the values of objects a and b.

- Object operations: The form a Same b compares objects a and b. It returns true if and only if a and b are the same object.

.1. Assertions

```
<assertion> ::= <assert stmt> | <definition>

<assert stmt> ::= { <name> }? Assert <expression>
```

ıe `<expression>` must be a boolean expression. It is required to be true for all instances of the structure or ocess in which the assertion appears. The optional `<name>` can be used to reference the assertion in `ithout>` clauses.

.2. Expressions

ıe syntax of the expressions of the assertion language is given here.

```
<expression> ::= <1expression> | <expression> <1op> <1expression>

<1op> ::= Or | Union

<1expression> ::= <2expression> | <1expression> <2op> <2expression>

<2op> ::= And | Intersect

<2expression> ::= { <3op>  }? <3expression>

<3op> ::= Not

<3expression> ::= <4expression> | <3expression> <4op> <4expression>

<4op> ::= = | /= | < | <= | > | >= | In | Same | Psub | Sub

<4expression> ::= { <5op> }? <5expression> | <4expression> <5op> <5expression>

<5op> ::= + | -

<5expression> ::= <primary expression> | <5expression> <6op> <primary expression>

<6op> ::= * | /
```

These rules define a conventional expression grammar with operators organized into precedence levels. The operators `Or` and `Union` have lowest precedence, while * and / have highest priority.

```
<primary expression> ::= { <name> : }? <type>
                       | <literal>
                       | ( <expression> )
                       | <primary expression> . <name>
                       | <name> ( <actuals> )
                       | <if expression>
                       | <quantified expression>

<literal> ::= True | False
            | { <name> : }? Root
            | Empty
            | <integer>
            | <rational>
            | <string>

<actuals> ::= ( <expression> { , <expression> }* )
```

The `( <expression> )` form of `<primary expression>` is used only for grouping and has no other effect. The semantics of the other syntactic forms are discussed below.

Each expression will have a type. There are two possible kinds of expression types.

- IDL types. An expression may have integer, boolean, rational, string, sequence, or set types[4]. Operations used in value expressions are discussed in Section 3.4.
- Object set types. Here the expression represents a set of objects of some class. Operations used in structural expressions are discussed in Section 3.3.

[4] I.e., any of the types defined in Section 2.2.3 except class types.

3.3. Operations on Objects and Sets of Objects

The expressions discussed in this section all produce sets of objects. When such a set contains exactly one object we do not distinguish between a result which is this object and a result that is a set whose only member is this object.

The following expression forms all specify sets of objects:

- `Empty` : the literal for the empty object set
- `{ <port name> : }? <type>` : stands for the set of all objects (from the structure associated with the specified port) with the specified type. The port specification can appear if and only if the assertion appears within a process definition.
- `{ <port name> : }? Root` : is the literal for a set containing only the root node object. The port name rules are the same as the previous case.
- `<name>` : where `<name>` is a quantifier (see Section 3.6).
- `Members(setv)` : produces the object set of all objects whose locations are in the set value `setv`.
- `Head(seqv)` : where `seqv` is a non-empty sequence value produces the object set containing the object whose location is first in the sequence.

The following expression forms take existing object sets and produce a new object set.

- `Union, Intersect` : These are the object set union and intersection operations.
- `Type(n)` : where `n` is an object set produces the set of all node objects with the same types as those in `n`.
- Dot qualification : of an object set containing only node objects produces an object set which consists of the objects that are associated with the specified attribute of all these nodes.

The following operations are used to compare object sets to produce a boolean value result.

- `Same` : This is the object set equality operation. Two objects sets are equal if and only if they contain exactly the same objects.
- `Sub, Psub` : These are the subset and proper subset operations.

3.4. Operations on Values

The operations and literals listed here all produce values, as opposed to objects. They have conventional semantics and will not be further explained.

- boolean: `=, ~=, And, Or, Not, True, False`

- integer and rational: `=`, `~=`, `<`, `<=`, `>`, `>=`, `+`, `-`, `*`, `/`, `<integer>`, `<rational>` (operations that involve mixtures of integer and rational values are permitted).
- string: `=`, `~=`, `<`, `<=`, `>`, `>=`, `Size`, `<string>`
- set: `=`, `~=`, `In`, `Size`
- sequence: `=`, `~=`, `Size`, `Tail`
- node: `=`, `~=`

The lexical form of the literals used here is the same as that given in Section 4.1.

If an object-producting expression is used as an operand of an operation or an actuals of a built-in functions listed above, the value of the object is used as the value of the operand or actual parameter. In general, object-producing expressions produce sets of objects. In a value-producing expression the set must always contain exactly one object. To ensure this we restrict the object-producing expression forms that are permitted here to:

- A quantifier name (see Section 3.6).
- The `{ <name> : }? Root` form.
- The `Head(<expression>)` form.
- Dot qualification of one of these forms.
- `If` expressions where all expressions following the `Then` and `Else` have one of these forms.

All of these forms are guaranteed to produce a result which consists of a single object.

3.5. If Expressions

```
<if expression> ::= If <expression> Then <expression>
                    { OrIf <expression> Then <expression> }*
                    Else <expression>
                    Fi
```

The `<expression>`'s following `If` and `OrIf` must be boolean expressions. The `<expression>`'s following `Then` and `Else` must all have the same type which will be the type of the entire `<if expression>`[5].

[5] When `<if expression>`s are evaluated only one of the expressions following a `Then` or `Else` is evaluated. This fact is useful in ensuring that recursive value definitions are not cyclic.

3.6. Quantified Expressions

```
<quantified expression> ::= { ForAll | Exists } <name> In <expression>
                            Do <expression> Od
```

The expression following `In` must be an object set expression. The expression between `Do` and `Od` must be a boolean expression. The `<name>` before the `In` is defined to be a quantifier name, has an object set value, and may be referenced only within the boolean expression.

Both forms of quantified expression index over all the members of the object set specified by the object set expression and take as values each of the objects in this object set. The boolean expression is evaluated for each of these indexed objects. The result is a boolean value which is true if and only if all (at least one) of the indexed boolean expression evaluations are true for the `ForAll` (`Exists`) form.

3.7. Definitions

```
<definition> ::= Define <name> { <formals> }?
                 { = <expression> | Returns <type> }

<formals> ::= ( <formal> { , <formal> }* )

<formal> ::= <name> : <type>
```

There are two kinds of definitions

- User-defined functions - the `Returns` form of definition. This introduces the name of a user-defined function, whose body must be linked with the assertion checker.
- Value definitions - The expression after the = must have a value type. Invocation of a value definition produces a value result. Recursion is permitted but value definitions may not be cyclic (i.e. their evaluation must not involve cyclic identical calls).
- Object set definitions - The expression after the = must define an object set. Invocation of an object set definition produces an object set. Recursive and cyclic definitions are permitted. Cyclic definitions produce the minimum fixed point solution. The body of such a definition may not include `If` expressions; this restriction preserves monotonicity.

These functions are invoked using the `<name> ( <actuals> )` form of `<primary expression>`. The type of each actual expression must match the specifed type of the corresponding formal of the user defined function.

Overloading of definitions is permitted provided they can be distinguished by their formal parameter types. It is possible that the IDL translator could resolve the overloading of a user-defined function when the target language for an assertion checker could not; in this case an the IDL translator will issue an error message.

3.8. Example

The first example shows a collection of assertions which specify that a data structure is a tree.

```
Structure Tree Root exp Is

    exp ::= inner | leaf;

    inner_void ::= inner | void;

    void => ;

    inner => left:exp,
             right:exp;

    leaf => val: Integer;
        Assert ForAll l In leaf Do l.val <= 100 Od;

    exp => parent: inner_void;
        Assert ForAll e In exp Do
            If e Same Root then e.parent Sub void
                        else e Same e.parent.left Or
                             e Same e.parent.right    Fi
            Od;

    Define IDesc(n:leaf) = Empty;
    Define IDesc(n:inner) = n.left Union n.right;

    Define Desc(n:exp) = Reach(IDesc(n));
    Define Reach(n:exp) = n Union Desc(n);

    Assert ForAll n In inner Do Reach(n.left) Intersect Reach(n.right) Same Empty Od;
    Assert ForAll n In exp Do Not(n Sub Desc(n)) Od;
    Assert Reach(Root) Same exp;

End
```

The two overloaded 'IDesc' functions define the set of immediate descendants of leafs (the empty set) and inner nodes (the union of the values of the right and left attributes). 'Desc' defines the descendants of a node as all the nodes reachable from its immediate descendants. 'Reach' defines the nodes reachable from a node as itself plus all of its descendants. The first `ForAll` states that the set of nodes reachable from the left subnode of a inner node does not intersect the set of nodes reachable from its right subnode. The second says no node is a descendant of itself. The last says that all expression nodes are reachable from the root.

The second example shows an assertion that the input and output of a process are isomorphic.

```
Process A Is

    Pre input:Tree;
    Post output: Tree;

    Define Compatible(A:exp,B:exp) =
        If Type(A) Same Type(B) Then
            If Type(A) Same leaf Then
                True
            Else
                Compatible(A.left,B.left) And Compatible(A.right,B.right)
            Fi
        Else
            False
        Fi;
```

```
    Assert Compatible(input:Root.output:Root);
End
```

4. The External ASCII Representation

In order to communicate data between arbitrary programs, possibly written in different languages and running on different computers, there must be a canonical external representation for each concrete structure. We chose an ASCII encoding to maximize the portability of the data. This section defines that encoding.

The package that provides the interface between a process instance and data on its ports is *required* to provide operations for mapping to and from the ASCII representation. Programs are not required to use this representation, however, and operations to map to other, more efficient representations *are* permitted. Indeed, these alternative representations would be the preferred means of communication between production versions of the various processes.

The external representation of a concrete structure is completely defined by the abstract structure except for the representation of private types. The syntax of the external representation has free form lexical rules, so that variations based on spacing and comments are not significant. The representation of an object can be nested within the representation of the node that references it or placed at the highest level so as to produce a "flat" form. The distinction between nested and flat representations can be made on a object-by-object basis and is not significant.

Each private type must have an external representation which fits within the fixed syntax given below. The representation is specified by private type representations with `External` clauses is concrete structures derived from the abstract structure defining the type. For two programs to communicate via the external representation, they must use concrete structures which are descended from the same externally adequate concrete structure.

4.1. Lexical Rules

The lexemes permitted in the external representation are given below. Unlike the IDL specification, the external representation is not case sensitive, except within `<string>`'s. This implies a constraint on the use of case sensitivity in an IDL specification: two names which differ only in case of letters may not be used if both might appear in an external representation. Only node type names and attribute names appear in the external representation; there is no representation of class names. Thus node and attribute names may have the same spelling, ignoring case, as class names.

```
<token> ::= <basic token> | <punctuation>

<basic token> ::= TRUE | FALSE | <name> | <integer> | <rational> | <string>

<punctuation> ::= "{ | "} | < | > | : | := | * | [ | ]

<name> ::=  letter { letter | digit | _ }*

<integer> ::= { + | - }? <unsigned integer>

<unsigned integer> ::= { digit }+

<rational> ::= { + | - }? <unsigned rational>

<unsigned rational> ::= <basic rational>
                      | { <unsigned integer> | <basic rational } /
                        { <unsigned integer> | <basic rational> }

<basic rational> ::=   <unsigned integer> . <unsigned integer> { <exponent> }?
                     |   <unsigned integer> { . <unsigned integer> }? <exponent>
                     |   <unsigned integer> # <based> { . <based> }? # { <exponent> }?

<exponent> ::=  E <integer>

<based> ::= { digit | A | B | C | D | E | F }+

<string> ::= " { string_character  }* "
```

The `<rational>` literal can be used to represent any rational number. The form `i/j` is the rational number produced by dividing `i` by `j`. The form with the `#` can be used to represent numbers in any base between 2 and 16. The first `<unsigned integer>` gives the base in base 10 and must have a value between 2 and 16. The next part gives the value in that base. The exponent is given in base 10 and specifies the power of the base by which the number is to be multiplied. Representations that specify the same rational value (e.g. `1/2` and `0.5`) are considered to always be equivalent.

The `<string>` literal can represent any ASCII string. It may directly contain blanks and the ASCII printing characters, except " and ~. Each of the other ASCII characters is represented by a two character escape sequence. The character " is represented by "". The non-printing characters with octal values 0, 1, ..., 37 are represented by the escape sequences ~@, ~A, ..., ~_ (i.e. the control-shift equivalents of a standard ASCII keyboard). The character ~ is represented by ~~. The character whose octal value is 177 is represented by ~|.

Break symbols include blanks, comments, and "end-of-line"s. Comments start with -- and are terminated by the end of the line on which they appear. Any number of break symbols may appear between any two `<token>`s with no effect. Break symbols may not appear within tokens. Two adjacent `<basic token>`'s must be separated by at least one break symbol.

The second uses a "flat" form.

```
1 ^
2: tree[ context value; op times; left 3^; right 4^]
3: leaf[ context value; value 35 ]
4: leaf[ context value; value 23 ]
5: leaf[ context value; value 10 ]
1: tree[ context value; op plus; left 2^; right 5^]
```

Here, node 1 is the root node. Since the root node or a reference to it must come first in the external representation, the `1 ↑` was needed in the first line. If the representation of node 1 came first, the `1 ↑` could be omitted.

4.4. Mapping Between Internal and External Representations

Every package instantiated from an IDL definition will include a pair of reader/writer operations for mapping to/from the external representation.

The reader must be able to accept any legal form for its input; it must be able to read nested forms, "flat" forms, and mixtures of these.

There are a wide variety of choices for how the writer decides on output format. A particular implementation might provide defaults via site-specific extensions to the concrete process descriptions, or might have the writer driven by run-time options. It is not necessary that a writer be able to produce all possible variations between fully nested and completely flat; it may chose to implement only one preferred form.

5. Instantiation of IDL Specifications

A prime purpose of IDL is to provide a notation for describing data structures so that an automated tool can generate a variety of data declarations, data structures, and code segments from the description. From the IDL description of a system, it is possible to generate

- The specification of a package that defines the operations a concrete process may perform on the internal data structures[6].
- The implementation of the operations for manipulating the internal representation of a concrete structure.
- Tables or code for a reader that inputs the external representation described in Chapter 4 and maps it into whatever internal representation is needed for a particular concrete process, and for a writer that performs the opposite transformation.
- Tables or code for a checker that verifies that a particular data structure satisfies the assertions of some structure.

This chapter discusses the issues involved in instantiating an IDL description. These issues are also covered in the CMU IDL implementation document [4].

5.1. Implementation of Concrete Structures

Implementing a concrete structure involves deciding how to implement IDL nodes, IDL classes, and attributes of IDL nodes. Because IDL supports a wide range of target languages, the implementation of IDL data structures will vary from one target language to another. When provided by the target language, use of an abstract type facility is the preferred approach. In this case the IDL internal level will be divided into two parts: one for the abstract specification (i.e. the externally visible types and operations) and a second part for the implementation. For languages lacking an abstract type facility, an attempt should be made to follow the abstract type methodology.

The straightforward implementation of an IDL structure is to define an implementation language record type for each IDL node type, and to represent IDL attributes as fields of the records. IDL classes complicate this view slightly, since they are used as types of attributes. In a language that allows untyped pointers there is no need for a representation of classes, since node-valued attributes can be represented as untyped pointers. In a language with union types, each IDL class can be represented as a union of the node types comprising the class. In a strongly typed language with variant records, it might be convenient to represent all node types as variants of a single type.

---

[6] An example of an Ada package is given in the Diana report.

The record implementation is one among many alternative implementations allowed by the abstract/concrete split in IDL. The following paragraphs discuss some of the implementation options.

- *A Coroutine Organization.* It is common for the Front and Back Ends of a compiler to be organized in a coroutine manner; the Front End produces a portion of the intermediate representation after which the Back End produces code for this portion and then discards the unneeded pieces of the intermediate representation. In this organization there would never be a complete representation for the entire structure used to communicate between the two phases. Instead, only a consistent subgraph for the portion being communicated is needed. To use this style of compiler organization, the user needs only to ensure that the values of all of the attributes for that portion of the tree being communicated are defined properly.

- *Non tree structures.* IDL is oriented towards graph-structured and tree-structured data. Many simple compilers use a linear representation, such as polish postfix. Such a representation simplifies certain tree traversals, and indeed may be obtained from a tree representation by such a traversal. Such representations may also have an advantage in that they are more efficient where storage is limited or paging overheads are high. An IDL description might suggest a tree structure, but a linear representation is entirely within the spirit of IDL. Where an IDL description requires a (conceptual) pointer it may be replaced by an index into the linear representation.

- *Attributes outside the nodes.* There is no need for the attributes of a node to be stored contiguously. There are many variations on this theme, but we will illustrate with just one here. Suppose that the general storage representation to be used involves storing each node as a record in the heap and using pointers to encode structural attributes. Because there are a number of different attributes associated with each node type, one may not wish to store these attributes directly in the records representing the nodes. Instead, one might define a number of vectors (of records) where the records in each vector are tailored to the various groupings of attribute types in IDL nodes. Using this scheme, the nodes themselves need only contain indices into the relevant vectors. Such a scheme has the advantage of making nodes of uniform size as well as facilitating the sharing of identical sets of attribute values.

- *Nodes inside other nodes.* An attribute of a node may 'reference' another node, but this does not necessarily imply that a pointer is required; the referenced node may be directly included in the storage structure of the outer node so long as the processing permits this. This is especially important where the referenced node has no attributes. If a class consists entirely of node types with no attributes, and node objects within the class are never shared, then the class can be implemented as an enumerated type, with the node types in the class as literals of the enumerated type.

5.2. Implementation of the Reader

The syntax of the external representation can be described by a fairly small LALR(1) grammar, as well as by an LL(2) grammar (see Figure 5-1). The parsing component of the reader can be generated automatically. Building the internal data structures can be more difficult.

The primitive syntactic elements of the external representation are labels, strings, integers, lists of

```
<program> ::= <node-ref> <lnode-list> $$
<lnode-list> ::= <lnode> <lnode-list> | $$
<lnode> ::= <label> : <node> $$
<lvalue> ::= <label> : <value> $$
<value> ::= <simple-value> | { <value-list> } | "< <value-list> "> | <node> $$
<reference> ::= <value> | labelref $$
<value-list> ::= <pvalue> <value-list> | $$
<pvalue> ::= <reference> | <lvalue> $$
<label> ::= integer | name $$
<node-ref> ::= <label> ↑ | <node> $$
<node> ::= name <factor-1> $$
<factor-1> ::= [ <pair-list> ] | $$
<pair-list> ::= <pair> <pair-list-p> | $$
<pair-list-p> ::= ; <pair> <pair-list-p> | $$
<pair> ::= name <pvalue> $$
<simple-value> ::= integer | rational | string | true | false $$
```

Figure 5-1: LL(2) Grammar for IDL External Representations

ttribute/value pairs, and lists of values. From these the reader must build the internal data structures. If the mplementation language is loosely typed or typeless, the reader can be driven by a set of tables describing the ayout of each node type. When the reader encounters a node, it fetches the description of the node type from symbol table, using the node name as the key. For each of the attribute/value pairs in the node's external epresentation, the reader applies one of a small set of transformations in order to convert it into an internal epresentation, and places the result in an appropriate place in the node representation. Labels can be andled in a second fixup pass, in the same manner as most assemblers.

a strongly typed language, the strong typing prevents this kind of table-driven approach. The IDL ocessor must generate code in this case. Furthermore, the symbol table needed for label processing requires at the objects stored in the symbol table be of some single type. This may require that all values that can be belled be represented by a single type, and thus may force all the IDL types to be represented as variants of ingle record type.

ivate types require the definition of an interface between the reader and the package implementing the ivate type. One possible interface is to have the private package provide a subroutine which takes the der's representation of the components of the external representation as parameters, and which returns a

value of the private type as its result.

5.3. Implementation of the Writer

The writer is subject to some of the same considerations as the reader. In a typeless language it can be table-driven; in a strongly typed language it is likely to be "hard code."

The key problem for the writer is generating labels for node objects referenced from more than one attribute. This may require additional data structures to hold the labels, or may require a 'label' attribute in every node. In the latter case the label attribute should be added automatically by the IDL processor, rather than requiring users to insert such an attribute. To generate a flat form, where all nodes are labelled and all node-valued attributes are represented as labels, requires some way for the writer to touch all nodes in a structure. To generate a nested form requires knowledge of which attributes are node-valued. If the data structure is known to be a tree then the writer can emit the nested form by a single tree walk. If the structure might be a graph, a pre-pass is needed to assign labels to nodes that might be referenced more than once.

Part Two: Formal Model of IDL

6. Notes on the Formal Notation

This chapter introduces some of the notational conventions used throughout Part Two of this document.

6.1. Sets

Let S and T be sets with s,s1,s2,...,sn $\in$ S and t $\in$ T.

The notation [S,T] denotes the cartesian product of S and T. Elements of cartesian products are refered to by ordered pairs and subscripts are used to select out each component. So, for example, if st$\in$ [S,T] then we may write either

$$st = \langle s,t \rangle$$

or

$$s = st.1 \wedge t = st.2$$

The notation [S]* stands for the set of all ordered sequences of values from S. We will write sequences in the form $\langle s1,s2,...,sn \rangle$. Sequences are accessed with two functions: with $car(\langle s1,s2,...,sn \rangle) = s1$ and $cdr(\langle s1,s2,...,sn \rangle) = \langle s2,...,sn \rangle$. Sequences may be constructed with $cons(s1,\langle s2,...,sn \rangle) = \langle s1,s2,...,sn \rangle$. The predicate $s \in \langle s1,s2,...,sn \rangle$ is true iff s is equal to some si in the sequence. The notation $\mathcal{P}(S)$ stands for the power set of S.

The notation S+T stands for the disjoint union of S and T.

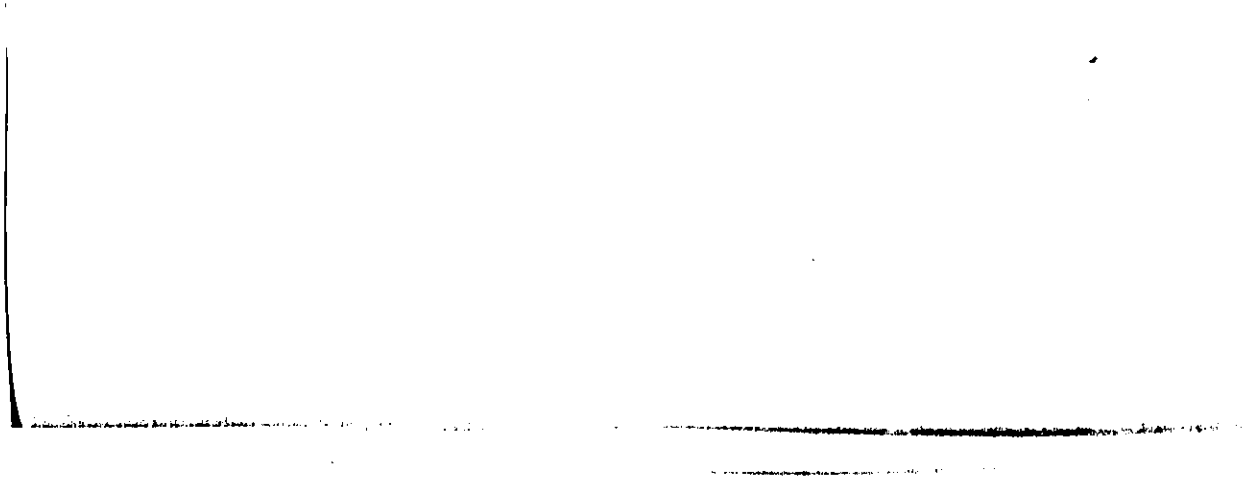
The notation S $\rightarrow$ T denotes the set of all (total and partial) computable functions from S to T.

6.2. Operation Definitions

In the following chapters, we define a number of operations on model domains. Each of these definitions is of the form:

OP: domain $\rightarrow$ range
- **note:** some prose that describes the intuitive effect of the operation
- **use:** an example of programming language-like use of the operation
- **pre:** the precondition of the operation (in case the precondition is true we omit it)
- **post:** the postcondition of the operation

The **use** clause in this definition is often used to establish names which are subsequently used in the **pre** and **post** conditions.



7. Formal Model of IDL Graphs

This chapter formally defines the model of IDL attributed graphs. We first define the domain of graphs and then specify graph operations. We begin by defining the domain; to do this we will some auxiliary domains that are used in the definition:

TYPE	is a countable set of 'types'.
VALUE	is a countable set of 'values'.
TAG	is a countable set of 'tags'.

We shall have more to say about these domains and their elements later. Intuitively, however, TYPE is a collection of types, VALUE is a collection of values of these types, and TAG is a collection of values used to distinguish between objects of the same type. For the moment we only need the fact that they are mutually disjoint and that there exists a function which maps from types to the possible values of that type.

vset: TYPE → $\mathcal{P}$(VALUE)

That is, 'vset' maps each type into a set of values in VALUE. Note that we do not require that this function nduce a partition on VALUE, thus a single value can be in the vset of more than one type. We will also need ome distinguished values in these domains; these distinguished values will be used to model deleted objects nd undefined values and attributes:

delvalue ∈ VALUE	where ∀t∈TYPE **delvalue**∉vset(t)
undefvalue ∈ VALUE	where ∀t∈TYPE **undefvalue**∉vset(t)
undeftag ∈ TAG	

Ve will also need to know what types the root node may have.

roottypes⊆TYPE

'e will also need the following derived domain:

LOCATION ≜ [TYPE,TAG]

ituitively LOCATION is a domain of 'typed addresses'.

'e can now define a domain that characterizes the model:

GRAPH ≜ [LOCATION, LOCATION → VALUE]

where

$\forall$graph$\in$GRAPH,graph.1.2$\neq$**undeftag**$\Rightarrow$
graph.1.1$\in$roottypes

$\forall$graph$\in$GRAPH, $\forall$<type,tag>$\in$LOCATION, $\forall$value$\in$VALUE,
graph.2(<type,tag>) = value $\wedge$ value$\neq$ **delvalue** $\wedge$ value$\neq$ **undefvalue**
$\Rightarrow$ value$\in$vset(type)

$\forall$graph$\in$GRAPH, $\neg$($\exists$type$\in$TYPE, $\exists$value$\in$VALUE, graph.2(<type,**undeftag**>) = value)

where (a) the first LOCATION is distinguished and called the 'root' of the graph, and where (b) 'LOCATION $\rightarrow$ VALUE' is an abstract store that associates values with locations. Intuitively, each graph$\in$GRAPH consists of a distinguished root location and a function which given a 'location' returns the value that is 'stored' there. We can also think of a graph$\in$GRAPH as describing a set of *objects*, each of which has a type, tag, and value. The root object is distinguished and is used as a means of gaining access to all the other objects. We will model changes to the data structure by operations that take an existing 'graph' and produce a new updated 'graph'. The first restriction ensures that the root object has a correct type. The second restriction given with the domain ensures that the values in the 'graph' are compatible with their location type. The third restriction ensures that there will never be an object with the **undeftag** tag.

The following definitions are for the operations permitted on graphs.

LOCATIONS: GRAPH $\rightarrow$ $\mathcal{P}$(LOCATION)
note: Returns the set of (locations of) all objects in a graph.
use: s := LOCATIONS(g)
post: s = { loc$\in$LOCATION | $\exists$value$\in$VALUE, g.2(loc) = value }

EMPTYGRAPH: $\rightarrow$ GRAPH
note: Constructor used to obtain the empty graph.
use: g := EMPTYGRAPH
post: LOCATIONS(g) = $\emptyset$
$\wedge$ g.1.2 = **undeftag**

CREATE: [GRAPH,TYPE] $\rightarrow$ [GRAPH,LOCATION]
note: Allocates a new object of the specified type and returns its location; the new object is uninitialized.
use: <g1,loc> := CREATE(g,type)
post: loc$\notin$LOCATIONS(g)
$\wedge$ LOCATIONS(g1) = LOCATIONS(g) $\cup$ {loc}
$\wedge$ loc.1 = type
$\wedge$ $\forall$loc1$\in$LOCATION (loc1$\neq$loc $\Rightarrow$ g1.2(loc1) = g.2(loc1))
$\wedge$ (loc1 = loc $\Rightarrow$ g1.2(loc1) = **undefvalue**)

DESTROY: [GRAPH,LOCATION] → GRAPH

note: Frees (deallocates) the object at the specified location; the object is not actually destroyed, but instead is given the distinguished value **delvalue.** This allows other preconditions on other operations to prohibit dereferencing a 'dangling pointer'.

use: g1 := DESTROY(g,loc)

pre: loc∈LOCATIONS(g)
∧ g.2(loc)≠**delvalue**

post: ∀loc1∈LOCATION, (loc1≠loc ⇒ g1.2(loc1)=g.2(loc1))
∧ (loc1=loc ⇒ g1.2(loc1) = **delvalue**)

FETCH: [GRAPH,LOCATION] → VALUE

note: Retrieves the value associated with the specified object.

use: value := FETCH(g,loc)

pre: loc∈LOCATIONS(g)
∧ g.2(loc)≠**delvalue** ∧ g.2(loc)≠**undefvalue**

post: value = g.2(loc)

STORE: [GRAPH,LOCATION,VALUE] → GRAPH

note: Sets the value of the object at the specified location to a specified value. The previous value of this object is lost. Note that the value of a freed object cannot be altered and that the type of the object must be that of the value to be stored -- the type of an object's value cannot be changed.

use: g1 := STORE(g,loc,value)

pre: loc∈LOCATIONS(g)
∧ g.2(loc)≠**delvalue**
∧ value∈vset(loc.1)

post: ∀loc1∈LOCATION, (loc1≠loc ⇒ g1.2(loc1)=g.2(loc1))
∧ (loc1=loc ⇒ g1(loc1) = value)

FETCHROOT: GRAPH → LOCATION

note: Returns the distinguished root of the graph.

use: loc := FETCHROOT(g)

pre: g.1.2 ≠ **undeftag**

post: loc = g.1

STOREROOT: [GRAPH,LOCATION] → GRAPH

note: Sets the distinguished root of the graph to the specified location.

use: g1 := STOREROOT(g,loc)

pre: loc.1∈roottypes

post: g1 = ⟨loc,g.2⟩

8. Formal Model of IDL Types

In this chapter the IDL types and their associated value sets are formally defined. We begin by giving the definition of the complete IDL TYPE and VALUE domains and then in the following sections describing each type and its values.

TYPE ≜ {boolean,integer,string,rational} ∪ NN ∪ [{ seq,set},TYPES] ∪ PT

VALUE ≜ BVALUE ∪ IVALUE ∪ SVALUE ∪ RVALUE ∪ NV ∪ Set ∪ Seq ∪ PVALUE ∪ {delvalue,undefvalue}

The auxiliary domain TYPES is defined as:

TYPES ≜ { {type} | type∈TYPE } ∪ $\mathcal{P}$(NN)

Members of this domain will be used to constrain the types of objects that node attributes and elements of sets and sequences can reference. In the simple case, a reference can be to only one type of object. For references to nodes, however, the reference can be to any node object having any of a specified set of node types. This feature of the model provides support for the kinds of data structuring done in many programming languages with variant records or union types.

8.1. Boolean Type

The IDL `Boolean` type has type

boolean

and value set

BVALUE ≜ {false,true}

and type-value set association

vset(boolean) = BVALUE

8.2. Integer Type

The IDL `Integer` type has type

integer

and value set

IVALUE ≜ {...,-2,-1,0,1,2,3,...}

and type-value set association

vset(integer) = IVALUE

8.3. String Type

The IDL `String` type has type

string

and value set

SVALUE ≜ [CHAR]*

where CHAR is the set of all ASCII characters and type-value set association

vset(string) = SVALUE

8.4. Rational Type

The IDL `Rational` type has type

rational

and value set

RVALUE ≜ { ⟨i,j⟩∈[IVALUE,IVALUE] | j≠0 }/Req
where ⟨i,j⟩ Req ⟨k,l⟩ ≡ i*l=j*k

and type-value set association

vset(Rational) = RVALUE

8.5. Node Types

Node types are characterized by a finite domain of node names.

NN is a finite set of 'node names'.

We will also need a domain for the names of the attributes of nodes.

AN is a finite set of 'attribute names'.

Each of these domains must be disjoint from the VALUE and TAG domains. The NN and AN domains may overlap.

We can now define the domain of node values:

NV ≜ [NN, AN→LOCATION]

Note that a node value consists of a node name and a function that maps attribute names to locations of objects in which the attribute values are 'stored'.

We must also have a function:

$$\Lambda\text{Type: [NN,AN]} \rightarrow \text{TYPES}$$

Given a node name and the name of one of its attributes, this function produces the set of types (of objects) that the attribute can reference. One major purpose of the IDL notation is to provide a humane way of defining this function.

We also require here that the 'vset' function will map each node type (i.e. node name) to a set of node values.

$$\forall nn \in NN,\ vset(nn) = \{\langle nn,f\rangle \in NV \mid \forall an \in AN,\ \forall types \in TYPES,\ (types = \Lambda Type(nn,an)) \Leftrightarrow (\exists tag \in TAG,\ \exists type \in types,\ f(an) = \langle type,tag\rangle)\}$$

The predicate ensures that all node values are compatible with ΛType (i.e. that they conform to their IDL node specification).

The operations on node values are defined below.

NAMES: NV → $\mathcal{P}$(AN)

note: Returns the set of all attribute names of a node value.
use: ans := NAMES(nv)
post: ans = { an∈AN | ∃loc∈LOCATION, nv.2(an) = loc }

FETCHCOMP: [NV,AN] → LOCATION

note: Returns a reference associated with the specified attribute. The attribute must have been initialized.
use: loc := FETCHCOMP(nv,an)
pre: an∈NAMES(nv)
∧ nv.2(an).2 ≠ undeftag
post: loc = nv.2(an)

STORECOMP: [NV,AN,LOCATION] → NV

note: Sets the reference associated with a specified attribute. The type of the location must be one of those permitted for the attribute.
use: nv1 := STORECOMP(nv,an,loc)
pre: an∈NAMES(nv)
∧ loc.1 ∈ ΛType(nv.1,an)
post: nv1.1 = nv.1
∧ ∀an1∈AN, (an1≠an ⇒ nv1.2(an1) = nv.2(an1))
∧ (an1 = an ⇒ nv1.2(an1) = loc)

CREATECOMP: [NN] → NV
note: Creates a node value for the specified type; the attributes of the node are uninitialized.
use: nv := CREATECOMP(nn)
post: nv.1 = nn
∧ ∀an∈AN, (∃types∈TYPES, Atype(nn,an) = types) ⇔ nv.2(an).2 = undeftag

8.6. Sequence Types

A sequence type has the form

[seq,types]

where types ∈ TYPES and controls what types of values elements of the sequence may reference. Informally a sequence value is a typed n-tuple of locations; formally,

Seq ≜ [TYPES,[LOCATION]*]

The vset for sequences is defined by

vset([seq,types]) = { <types,locs> | ∀loc∈locs.loc.1∈types }

The restriction here ensures that elements of a sequence may only reference objects of the permitted types. The operations on sequence values are formally defined below.

HEAD: Seq → LOCATION
note: Returns the first element of the specified sequence.
use: loc := HEAD(s)
pre: ¬ IsEMPTY(s)
post: loc = car(s.2)

TAIL: Seq → Seq
note: Removes the first element of the specified sequence and returns the remainder.
use: sl := TAIL(s)
pre: ¬ IsEMPTY(s)
post: sl = <s.1,cdr(s.2)>

IsEMPTY: Seq → boolean
note: Returns true iff the specified sequence is empty.
use: b := IsEMPTY(s)
post: b ≡(s.2 = <>)

MAKE: [LOCATION,Seq] → Seq
- **note:** Constructor that returns the sequence consisting of the specified object as its head and the specified sequence as its tail.
- **use:** sl := MAKE(loc,s)
- **pre:** loc.1 ∈ s.1
- **post:** sl = <s.1,cons(loc.s.2)>

EMPTYSEQ: TYPES → Seq
- **note:** Constructor for the empty sequence.
- **use:** s := EMPTYSEQ(types)
- **post:** s = <types,<>>

8.7. Set Types

A set type has the form

[set,types]

where types ∈ TYPES and controls what types of values elements of the set may reference. Informally a set value is a typed set of locations; formally,

Set ≜ [TYPES,$\mathcal{P}$(LOCATION)]

The vset for sets is defined by

vset([set,types]) = { <types.locs> | ∀loc∈locs,loc.1∈types }

The restriction here ensures that elements of a set may only reference objects of the permitted types.

The operations on set values are defined below.

SELECT: Set → LOCATION
- **note:** Returns some (unspecified) element of the set. Note that SELECT may be used in conjunction with REMOVE and IsEMPTY to iterate over a set of values.
- **use:** loc := SELECT(s)
- **pre:** ¬ IsEMPTY(s)
- **post:** loc ∈ s.2

IsEMPTY: Set → boolean
- **note:** Returns true iff the specified set is empty.
- **use:** b := IsEMPTY(s)
- **post:** b≡(s.2 = ∅)

INSERT: [Set,LOCATION] → Set
note: Adds specified element to the specified set.
use: s1 := INSERT(s,loc)
pre: loc.1 ∈ s.1
post: s1 = <s.1,s.2 ∪ {loc}>

REMOVE: [Set,LOCATION] → Set
note: Removes specified element from the specified set (if present).
use: s1 := REMOVE(s,loc)
pre: loc.1 ∈ s.1
post: s1 = <s.1,{loc1|loc1∈s ∧ loc1≠loc}>

EMPTYSET: TYPES → Set
note: Constructor for the empty set.
use: s := EMPTYSET(types)
post: s = <types,∅>

8.8. Private Types

The type for private types is

PT - a finite set of private types

and the value set for private types is

PVALUE - a countable set of private values

The PVALUE set may include arbitrary values that are not present in the value sets of the built-in types; however, the PVALUE set need not be disjoint from the value sets os the built-in types. The type-value set association for private types must obey

∀pt∈PT,vset(PT)⊆PVALUE

9. Formal Model for Productions

Previous chapters of part two have discussed a formal model for IDL data structures that is universal in that it applies to all IDL structures. To specialize this model to a paticular structure, it is necessary to fully specify the NN, AN, and PT domains, the roottypes set, and the AType function. This chapter shows how this information is derived from the productions of some specific IDL structure. The formal technique used here is a denotational definition that operates over a somewhat simplified syntax for structures. In particular, given a structure, <abstract structure decl>, then the information can be found by:

$\langle NN,AN,PT,roottypes,AType\rangle \triangleq \mathcal{AMD}\ [\![\langle \texttt{abstract structure decl}\rangle]\!]$

Auxiliary Domain

NAMES $\triangleq \mathcal{P}(\langle \texttt{name}\rangle)$

Attribute Types Domain and Operations

ATYPES $\triangleq$ [<name>,<name>] $\rightarrow$ TYPES

EMPTYATTR: $\rightarrow$ ATYPES
- **note:** Returns an attributes types function.
- **use:** atype := EMPTYATTR
- **post:** $\neg(\exists$node,attr$\in\langle$name$\rangle$, $\exists$types$\in$ TYPES, atype(nn,an)=types)

ADDATTR: [ATYPES,<name>,<name>,NAMES] $\rightarrow$ ATYPES
- **note:** Adds an attribute, consisting of a node and attribute name together with the types permitted for the attribute to the attributes types function.
- **use:** atype1 := ADDATTR(atype,node,attr,types)
- **pre:** $\neg(\exists$types$\in$TYPES, atype(node,attr)=types)
- **post:** $\forall$nn1,an1$\in\langle$name$\rangle$, ($\langle$nn1,an1$\rangle\neq\langle$nn,an$\rangle \Rightarrow$ atype1(nn1,an1)=atype(nn1,an1))
  $\wedge$ ($\langle$nn1,an1$\rangle=\langle$nn,an$\rangle \Rightarrow$ atype1(nn1,an1)=types)

Environment Domain and Operations

ENV $\triangleq$ <name> $\rightarrow$ [{node,class,private},NAMES]

EMPTYENV: $\rightarrow$ ENV
- **note:** Returns an empty environment.
- **use:** env := EMPTYENV
- **post:** $\neg(\exists$name$\in\langle$name$\rangle$, $\exists$info$\in$ [{node,class,private},NAMES], env(name)=info)

DEFINE: [ENV,<name>,{node,class,private},NAMES] → ENV
- note: Adds the specified <name> and its type and value to the environment.
- use: env1 := DEFINE(env,name,typ,val)
- pre: ¬(∃info∈[{node,class,private},NAMES], env(name)=info)
- post: ∀name1∈<name>, (name1≠name ⇒ env1(name1)=env(name1))
  ∧ (name1=name ⇒ env1(name1)=<typ,val>)

FINDTYPES: [<name>,ENV] → NAMES
- note: Looks up the specified <name> in the environment and returns its associated value.
- use: names := FINDTYPES(name,env)
- pre: ∃info∈[{node,class,private},NAMES], env(name)=info
- post: names = env(name).2

Denotational Function Domains

$\mathcal{AMD}$: <abstract structure decl> → [NAMES,NAMES,NAMES,NAMES,ATYPES]

$\mathcal{AMS}$: <abstract structure stmts> → [ENV,ENV,ATYPES] →
[ENV,NAMES,NAMES,NAMES,ATYPES]

$\mathcal{CLS}$: <names> → ENV → NAMES

$\mathcal{ATT}$: <attributes> → [ENV,<name>,ATYPES] → [NAMES,ATYPES]

$\mathcal{TYP}$: <type> → ENV → TYPES

Denotational Rules

```
<abstract structure decl> ::= Structure <name> Root <name> Is
          <abstract structure stmts>  End
```

$\mathcal{AMD}$ [[<abstract structure decl>]] =
LetRec <envall,nn,an,pt,atype> = $\mathcal{AMS}$ [[<abstract structure stmts>]]
<EMPTYENV,envall,EMPTYATTR> In
Let rt = FINDTYPES([[<name>]],envall) In
<nn,an,pt,rt,atype>

```
<abstract structure stmts> ::= <abstract structure stmt> ;
```

$\mathcal{AMS}$ [[<abstract structure stmts>]] = $\mathcal{AMS}$ [[<abstract structure stmt>]]

```
<abstract structure stmts> ::= <abstract structure stmts>1 <abstract structure stmt> ;
```

$\mathcal{AMS}$ [[<abstract structure stmts>]] <env,envall,atype> =
Let <env1,nn1,an1,pt1,atype1> = $\mathcal{AMS}$[[<abstract structure stmts>1]]
<env,envall,atype> In
Let <env2,nn2,an2,pt2,atype2> = $\mathcal{AMS}$ [[<abstract structure stmt>]]
<env1,envall,atype1> In
<env2,nn1∪nn2,an1∪an2,pt1∪pt2,atype2>

<abstract structure stmt> ::= <class production>

$\mathcal{AMS}$ [[<abstract structure stmts>]] = $\mathcal{AMS}$ [[<class production>]]

<abstract structure stmt> ::= <node production>

$\mathcal{AMS}$ [[<abstract structure stmts>]] = $\mathcal{AMS}$ [[<node production>]]

<abstract structure stmt> ::= <type decl>

$\mathcal{AMS}$ [[<abstract structure stmts>]] = $\mathcal{AMS}$ [[<type decl>]]

<class production> ::= <name> ::= <names>

$\mathcal{AMS}$ [[<class production>]] <env,envall,atype> =
Let names = $\mathcal{CLS}$ [[<names>]] envall In
Let env1 = DEFINE(env,[[<name>]],class,names) In
<env1,Ø,Ø,Ø,atype>

<names> ::= <name>

$\mathcal{CLS}$ [[<names>]] envall =
FINDTYPES([[<name>]],envall)

<names> ::= <names>1 | <name>

$\mathcal{CLS}$ [[<names>]] envall =
$\mathcal{CLS}$ [[<names>1]] envall ∪ FINDTYPES([[<name>]],envall)

<type decl> ::= Type <name>

$\mathcal{AMS}$ [[<type decl>]] <env,envall,atype> =
Let env1 = DEFINE(env,<name>,private,{[[<name>]]}) In
<env1,Ø,Ø,{<name>},atype>

<node production> ::= <name> =>

$\mathcal{AMS}$ [[<node production>]] <env,envall,atype> =
Let env1 = DEFINE(env,[[<name>]],node,{[[<name>]]}) In
<env1,{[[<name>]]},Ø,Ø,atype>

<node production> ::= <name> => <attributes>

$\mathcal{AMP}$ ⟦<node production>⟧ <env,envall,atype> =
Let env1 = DEFINE(env,⟦<name>⟧,node{⟦<name>⟧}) In
Let <an1,atype1> = $\mathcal{APT}$ ⟦<attributes>⟧ <envall,⟦<name>⟧,atype> In
<env1,{⟦<name>⟧},an1,∅,atype1>

<attributes> ::= <attribute>

$\mathcal{APT}$ ⟦<attributes>⟧ = $\mathcal{APT}$ ⟦<attribute>⟧

<attributes> ::= <attributes>1 , <attribute>

$\mathcal{APT}$ ⟦<attributes>⟧ <env,nn,atype> =
Let <an1,atype1> = $\mathcal{APT}$ ⟦<attributes>1⟧ <env,nn,atype> In
Let <an2,atype2> = $\mathcal{APT}$ ⟦<attribute>⟧ <env,nn,atype1> In
<an1∪an2,atype2>

<attribute> ::= <name> : <type>

$\mathcal{APT}$ ⟦<attribute>⟧ <env,nn,atype> =
Let types = $\mathcal{TYP}$ ⟦<type>⟧ env In
<{⟦<name>⟧},ADDATTR(atype,nn,⟦<name>⟧,types)>

<type> ::= Boolean

$\mathcal{TYP}$ ⟦<type>⟧ env = {boolean}

<type> ::= Integer

$\mathcal{TYP}$ ⟦<type>⟧ env = {integer}

<type> ::= String

$\mathcal{TYP}$ ⟦<type>⟧ env = {string}

<type> ::= Rational

$\mathcal{TYP}$ ⟦<type>⟧ env = {rational}

<type> ::= Set Of <type>1

$\mathcal{TYP}$ ⟦<type>⟧ env = {<set,$\mathcal{TYP}$ ⟦<type>1⟧>}

<type> ::= Seq Of <type>1

$\mathcal{TYP}$ ⟦<type>⟧ env = {<seq,$\mathcal{TYP}$ ⟦<type>1⟧>}

<type> ::= <name>

$\mathcal{TYP}[\![\langle type \rangle]\!]\ env =$
FINDTYPES($[\![\langle name \rangle]\!]$,env)

10. Formalization of the External Form

10.1. Formal Mapping from the External Form

This section formally specifies the mapping from externals forms to the formal attributed directed graph domain. The approach used here is to specify the semantics of external forms denotationally in terms of the operations of the formal model given in previous chapters. In particular, if <ASCII rep>1 is some particular instance of an external form of some structure, then $\mathcal{E}[\![$<ASCII rep>1$]\!]$ will be the graph$\in$GRAPH that it represents.

An external form is considered to be valid for some externally adequate structure iff all of the preconditions of the operations used to convert it to its corresponding graph are satisfied.

This section is now incomplete. It is missing:

- *Rules for the external representation of private types.*
- *Rules for handling the fact that IDL is case sensitive but the external form is not.*
- *Rules for the semantics of literals (i.e. The $\mathcal{B}$ semantic rules).*
- *More informal descriptions of the rules.*

These will all be included in later versions of this document.

Label Table Domain and Operations

LABELS $\triangleq$ <label> $\rightarrow$ LOCATION

EMPTYLABELS: $\rightarrow$ LABELS
- **note:** Returns an empty label table.
- **use:** L := EMPTYLABELS
- **post:** $\neg(\exists$label$\in$<label>, $\exists$loc$\in$LOCATION, L(label) = loc)

ADDLABELS: [<label>,LOCATION,LABELS] $\rightarrow$ LABELS
- **note:** Adds the specified <label> and its location to the label table.
- **use:** L1 := ADDLABELS(label,loc,L)
- **pre:** $\neg(\exists$loc$\in$LOCATION, L(label) = loc)
- **post:** $\forall$label1$\in$<label>, (label1$\neq$label $\Rightarrow$ L1(label1) = L(label1))
  $\wedge$ (label1 = label $\Rightarrow$ L1(label1) = loc)

FINDLABELS: [<label>,LABELS] → LOCATION

note: Looks up the specified <label> in the label table and returns its associated location.

use: loc := FINDLABELS(label,L)

pre: ∃loc∈LOCATION, L(label)=loc

post: loc = L(label)

Denotational Function Domains

$\mathcal{S}$: <ASCII rep> → GRAPH

$\mathcal{N}$: <node> → [LABELS,LABELS] → GRAPH → [NV,TYPE,LABELS,GRAPH]

$\mathcal{A}$: <attribute> → [LABELS,LABELS] → GRAPH → NV → [NV,LABELS,GRAPH]

$\mathcal{R}$: <reference> → [LABELS,LABELS] → GRAPH → TYPES →
[LOCATION,LABELS,GRAPH]

$\mathcal{V}$: <value> → [LABELS,LABELS] → GRAPH → TYPES →
[VALUE,TYPE,LABELS,GRAPH]

$\mathcal{L}$: <labeled nodes> → [LABELS,LABELS] → GRAPH → [LABELS,GRAPH]

$\mathcal{B}$: <literal> → VALUE

ASCII rep Denotational Rules

<ASCII rep> ::= <reference> <labeled nodes>

$\mathcal{S}$ [[<ASCII rep>]] =
LetRec <Lall,g4> =
Let g = EMPTYGRAPH In
Let <loc,L1,g1> = $\mathcal{R}$ [[<reference>]] <EMPTYLABELS,Lall> g In
Let g2 = STOREROOT(g1,loc) In
Let <L2,g3> = $\mathcal{L}$ [[<labeled node>]] <L1,Lall> g2 In
<L2,g3>
In
g4

<labeled nodes> ::=

$\mathcal{L}$ [[<labeled nodes>]] <L1,Lall> g = <L1,g>

```
<labeled nodes> ::= <labeled nodes>1 <label> : <node>
```

$\mathcal{L}$ [[<labeled nodes>]] <L1,Lall> g =
Let <L2,g1> = $\mathcal{L}$ [[<labeled nodes>1]] <L1,Lall> g In
Let <nv,type,L3,g2> = $\mathcal{N}$ [[<node>]] <L2,Lall> g1 In
Let <g3,loc> = CREATE(g2,type) In
Let g4 = STORE(g3,loc,nv) In
Let L4 = ADDLABELS([[<label>]],loc,L3) In
<L4,g4>

Node Denotational Rules

```
<node> ::= <name>
```

$\mathcal{N}$ [[<node>]] <L,Lall> g =
< CREATECOMP([[<name>]]) , [[<name>]] , L , g>

```
<node> ::= <name> [ <attributes> ]
```

$\mathcal{N}$ [[<node>]] <L,Lall> g =
Let nv = CREATECOMP([[<name>]]) In
Let <nv1,L1,g1> = $\mathcal{A}$ [[<attribute>]] <L,Lall> g nv In
<nv1,[[<name>]],L3,g1>

Attribute Denotational Rules

```
<attributes> ::= <attribute>
```

$\mathcal{A}$ [[<attributes>]] = $\mathcal{A}$ [[<attribute>]]

```
<attributes> ::= <attributes>1 ; <attribute>
```

$\mathcal{A}$ [[<attributes>]] <L,Lall> g nv =
Let <nv1,L1,g1> = $\mathcal{A}$ [[<attributes>1]] <L,Lall> g nv In
$\mathcal{A}$ [[<attribute>]] <L1,Lall> g1 nv1

```
<attribute> ::= <name> <reference>
```

$\mathcal{A}$ [[<attribute>]] <L,Lall> g nv =
Let <loc,L1,g1> = $\mathcal{R}$ [[<reference>]] <L,Lall> g In
< STORECOMP(nv,[[<name>]],loc) , L1 , g1 >

Reference Denotational Rules

```
<reference> ::= <value>
```

$\mathcal{R}$ ⟦<reference>⟧ <L,Lall> g types =
Let <v,type,L1,g1> = $\mathcal{V}$⟦<value>⟧ <L,Lall> g types In
Let <g2,loc> = CREATE(g1,type) In
Let g3 = STORE(g2,loc,v) In
<loc,L1,g3>

<reference> ::= <label> : <value>

$\mathcal{R}$ ⟦<reference>⟧ <L,Lall> g types =
Let <v,type,L1,g1> = $\mathcal{V}$⟦<value>⟧ <L,Lall> g types In
Let <g2,loc> = CREATE(g1,type) In
Let g3 = STORE(g2,loc,v) In
Let L2 = ADDLABELS(⟦<label>⟧,loc,L1) In
<loc,L2,g3>

<reference> ::= <label> ↑

$\mathcal{R}$ ⟦<reference>⟧ <L,Lall> g types =
< FINDLABELS(⟦<label>⟧,Lall) , L , g >

Value Denotational Rules

<value> ::= TRUE

$\mathcal{V}$⟦<value>⟧ <L,Lall> g types =
<true,boolean,L,g>

<value> ::= FALSE

$\mathcal{V}$⟦<value>⟧ <L,Lall> g types =
<false,boolean,L,g>

<value> ::= <integer>

$\mathcal{V}$⟦<value>⟧ <L,Lall> g types =
< $\mathcal{B}$ ⟦<integer>⟧ , integer , L , g>

<value> ::= <string>

$\mathcal{V}$⟦<value>⟧ <L,Lall> g types =
< $\mathcal{B}$ ⟦<string>⟧ , string , L , g>

<value> ::= <rational>

$\mathcal{V}$⟦<value>⟧ <L,Lall> g types =
< $\mathcal{B}$ ⟦<rational>⟧ , rational , L , g>

<value> ::= <node>

$\mathcal{T}$[[<value>]] <L,Lall> g types =
Let <nv,type,L1,g1> = $\mathcal{N}$[[<node>]] <L,Lall> g In
<nv,type,L1,g1>

<value> ::= { <set values> }

$\mathcal{T}$[[<value>]] = $\mathcal{T}$[[<set values>]]

<set values> ::=

$\mathcal{T}$[[<set values>]] <L,Lall> g types =
Let {type} = types In
< EMPTYSET(type.2) , type , L , g>

<set values> ::= <set values>1 <reference>

$\mathcal{T}$[[<set values>]] <L,Lall> g types =
Let <v,type,L1,g1> = $\mathcal{T}$[[<set values>1]] <L,Lall> g types In
Let <loc,L2,g2> = $\mathcal{R}$ [[<reference>]] <L1,Lall> g1 type.2 In
Let v1 = INSERT(v,loc) In
<v1,type,L2,g2>

<value> ::= < <seq values> >

$\mathcal{T}$[[<value>]] = $\mathcal{T}$[[<seq values>]]

<seq values> ::=

$\mathcal{T}$[[<value>]] <L,Lall> g types =
Let {type} = types In
< EMPTYSEQ(type.2) , type , L , g >

<seq values> ::= <reference> <seq values>1

$\mathcal{T}$[[<value>]] <L,Lall> g types =
Let {type} = types In
Let <loc,L1,g1> = $\mathcal{R}$ [[<reference>]] <L,Lall> g type.2 In
Let <v,type1,L2,g2> = $\mathcal{T}$[[<seq values>1]] <L1,Lall> g1 types In
Let v1 = MAKE(loc,v) In
<v1,type,L2,g2>

10.2. Formal Mapping to the External Form

Previous sections defined the many-to-one mapping from external forms to internal forms. We define the possible mappings from internal form to external form as the many possible inverse mappings. Given a graphϵGRAPH, then its possible external forms are

External_Forms(graph) = { <ASCII rep> | $\mathcal{E}$ [[<ASCII rep>]] = graph }

That is, the possible external forms of graph are all <ASCII rep>s such that the $\mathcal{E}$ map applied to such an <ASCII rep> yields graph. The writer must be able to produce at least one of these forms.

References

[1] Digital Equipment Corporation.
BLISS Language Guide.
1977.

[2] G. Goos and W. A. Wulf (editors).
Diana Reference Manual.
Technical Report CMU-CS-81-101, Carnegie-Mellon University, Computer Science Department, March, 1981.

[3] A.N. Habermann.
The Gandalf Research Project.
In *Computer Science Research Review*, . Carnegie-Mellon University, Computer Science Department, 1978-79.

[4] David Alex Lamb.
IDL Processor Implementation Description.
Technical Report to be published, Carnegie-Mellon University, December, 1981.

[5] B.W. Leverett, R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz, W.A. Wulf.
An Overview of the Production Quality Compiler-Compiler Project.
Technical Report CMU-CS-79-105, Carnegie-Mellon University, Computer Science Department, February, 1979.

[6] J. R. Nestor, M. Beard.
Front End Generator User's Guide.
Technical Report to appear, Carnegie-Mellon University, Computer Science Department, 1981.

[7] J.M. Newcomer, R.G.G. Cattell, P.N. Hilfinger, S.O. Hobbs, B.W. Leverett, A.H. Reiner, B.R. Schatz, W.A. Wulf.
PQCC Implementor's Handbook.
Internal Documentation, Carnegie-Mellon University, Computer Science Department, October, 1979.

[8] W.A. Wulf, D.B. Russell, and A.N. Habermann.
BLISS: a Language for Systems Programming.
Communications of the ACM 14(12):780-790, December, 1971.

Appendix I
IDL BNF Summary

```
<1expression> ::= <2expression> | <1expression> <2op> <2expression>

<1op> ::= Or | Union

<2expression> ::= { <3op> }? <3expression>

<2op> ::= And | Intersect

<3expression> ::= <4expression> | <3expression> <4op> <4expression>

<3op> ::= Not

<4expression> ::= { <5op> }? <5expression> | <4expression> <5op> <5expression>

<4op> ::= = | ~= | < | <= | > | >= | In | Same | Psub | Sub

<5expression> ::= <primary expression> | <5expression> <6op> <primary expression>

<5op> ::= + | -

<6op> ::= * | /

<abstract process decl> ::= Process <name> Is { <abstract process stmt> ; }+ End

<abstract process stmt> ::= <pre stmt> | <post stmt> | <assertion>

<abstract structure decl> ::= Structure <name> Root <name> Is { <name list> Except }? {
        <abstract structure stmt> ; }+ End

<abstract structure stmt> ::= <production> | <type decl> | <without clause> | <assertion>

<actuals> ::= ( <expression> { , <expression> }* )

<assert stmt> ::= { <name> }? Assert <expression>

<assertion> ::= <assert stmt> | <definition>

<attribute oper> ::= { Fetch | Store} ( <name list> )

<attribute> ::= <name> : <type>

<class production> ::= <name> ::= <name> { | <name> }*

<concrete process decl> ::= Concrete Process <name> Is <name> With { <concrete process stmt> ;
        }+ End

<concrete process stmt> ::= <port assoc> | <restriction> | <group decl> | <assertion>

<concrete structure decl> ::= Concrete Structure <name> Is <name> With { <concrete structure
        stmt> ; }+ End

<concrete structure stmt> ::= <type rep> | <production> | <assertion>

<decl> ::= <structure decl> | <process decl>

<definition> ::= Define <name> { <formals> }? { = <expression> | Returns <type> }

<expression> ::= <1expression> | <expression> <1op> <1expression>

<formal> ::= <name> : <type>
```

```
<formals> ::= ( <formal> { , <formal> }* )

<group decl> ::= Group <name list> Inv <name>

<if expression> ::= If <expression> Then <expression> { OrIf <expression> Then <expression> }*
       Else <expression> Fi

<internal type rep> ::= For <type reference> Use <type>

<literal> ::= True | False | { <name> : }? Root | Empty | <integer> | <rational> | <string>

<name list> ::= <name> { , <name> }*

<node oper> ::= Create | Destroy

<node production> ::= <name> => { <attribute> { , <attribute> }* }?

<oper list> ::= <oper> { , <oper> }*

<oper> ::= <node oper> | <attribute oper>

<port assoc> ::= For <name> Use <name>

<port decl> ::= <name> : <name>

<port list> ::= <port decl> { , <port decl> }*

<post stmt> ::= Post <port list>

<pre stmt> ::= Pre <port list>

<primary expression> ::= { <name> : }? <type> | <literal> | ( <expression> ) | <primary
       expression> . <name> | <name> ( <actuals> ) | <if expression> | <quantified expression>

<private rep> ::= <name> { . <name> }? | External <type>

<private type rep> ::= For <name> Use <private rep>

<process decl> ::= <abstract process decl> | <concrete process decl>

<production> ::= <class production> | <node production>

<quantified expression> ::= { ForAll | Exists } <name> In <expression> Do <expression> Od

<restriction> ::= Restrict <name> To <oper list>

<specification> ::= { <decl> }+

<structure decl> ::= <abstract structure decl> | <concrete structure decl>

<type decl> ::= Type <name>

<type reference> ::= <name> . <name> { ( * ) }*

<type rep> ::= <internal type rep> | <private type rep>

<type> ::= Boolean | Integer | String | Rational | Set Of <type> | Seq Of <type> | <name>

<without clause> ::= Without <without item> { , <without item> }*

<without item> ::= Assert <name>

<without item> ::= <name>

<without item> ::= { <name> | * } { => | ::= } { <name> }?
```