ISO / IEC JTC1 / SC22
Programming languages, their environments and system software interfaces
Secretariat: CANADA (SCC)

ISO/IEC JTC1/SC22
N 1319

FEBRUARY 1993

TITLE:	SC22/WG11 Liaison Statement to SC21/WG8 RPC Rapporteur, re: Alignment of LIPC and RPC standards
SOURCE:	Secretariat ISO/IEC JTC1/SC22
WORK ITEM:	JTC1.22.16
STATUS:	New
CROSS REFERENCE:	N/A
DOCUMENT TYPE:	Liaison Statement
ACTION:	For information to SC22 Member Bodies. This document has been forwarded to the SC21 Secretariat.

Address reply to: ISO/IEC JTC1/SC22 Secretariat
J.L. Côté
Treasury Board Secretariat
140 O'Connor St., 10th Floor, Ottawa, Ontario, Canada, K1A 0R5
Tel.: (613)957-2496 Telex: 053-3336 Fax: (613)957-8700

--

SC22/WG11/N353

To:	David Robinson JTC1/SC21/WG8 RPC Rapporteur
From:	Willem Wakker JTC1/SC22/WG11 Convenor
Subject:	Liaison statement on Alignment of LIPC and RPC standards
Date:	January 1993

SC22/WG11 appreciated the effort made by SC21/WG8/RPC in scheduling their interim meeting to be co-located with the WG11 meeting in Paris. Due to this scheduling, a number of joint sessions were possible, with the result that the WG11 documents now going to SC22 ballot can be more closely aligned with the RPC Parts 1 and 2, useful technical work on all of the related standards projects was performed, and most of the alignment problems have been resolved.

During these sessions, a number of alignment issues were identified which the RPC group asked to receive in a formal liaison statement. This document is that liaison statement. Except as noted below, all of these comments were accepted by the RPC experts in attendance at the joint sessions, but we recognize that they may or may not be formally accepted in the RPC editing meeting.

Considerable time was spent understanding the model of procedure context, in both the local and RPC case. The results of this discussion are captured in the model in the new LIPC draft, and we would appreciate comments from the RPC Group on this model and its relevance to RPC as well as LIPC. The model was drafted with input and agreement from the RPC experts at the joint sessions.

We look forward to your responses.

Comments on Part 1

Item 1.1:	Change "calling procedure" to "caller".
Type:	Editorial.
Ref:	3.5.3 and all occurrences.

Change the term "calling procedure" to "caller". The text correctly defines a procedure to be closed, and be entered from and return control to an external source. The calling entity may be difficult or impossible to describe this way, and does not need to be a "procedure" for the rest of the LIPC/RPC concepts to apply.

This requires the definition in 3.5.3 to be replaced by something like:

caller: an entity or sequence of instructions which invokes a procedure.

Replace "calling procedure" by "caller" and the phrase "calling and called procedures" by "caller and the called procedure" in all occurrences.

Item 1.2:	Improve definition of "procedure return"
Type:	E
Ref:	3.5.5

The definition of "procedure return" seems circular. The important aspect is that this is where control is returned to the caller. The notion of "specific termination" is an added feature - the generic concept does not require the possibility that there is more than one. We suggest the following:

procedure return: The act of returning control to the caller, with a specific termination.

Item 1.3:	Correct definition of "formal parameter".
Type:	E
Ref:	3.5.9

An interface specification is NOT the "definition of a procedure", which lies in the domain of programming languages. And the term "identification" is ambiguous, in that it leads to arguments about "how" it identifies (by name or by position) rather than "what" it identifies. The syntax in Part 2 refers to a formal parameter as a "declaration", which is correct. We suggest the following:

formal parameter: the declaration of a parameter (value) to be passed to or returned from a procedure when it is called.

Item 1.4:	Emphasize "passing" of "actual parameter".
Type:	E
Ref:	3.5.10

The given definition puts the emphasis on the programming language concept of substitution during execution. For LIPC/RPC, this notion is out of scope, and the emphasis should be on the calling mechanism. (An intentional departure from ISO 2382 is appropriate here.) We suggest:

actual parameter: A value communicated between a caller and a called procedure via a procedure call.

Item 1.5:	Generalize definition of "termination".
Type:	E
Ref:	3.5.14

The definition of termination should be refined to the LIPC/RPC notion, making it clear that terminations are both an indication of "how" the procedure is returning (normally, or with a particular "predefined response"), and a set (possibly empty) of return values, which are predeclared, but not "predefined". We suggest the following:

termination: A pre-defined state and a corresponding collection of values supplied by the called procedure on a procedure return.

Item 1.6:	Add "procedure image" and "procedure closure".
Type:	E
Ref:	3.5.21

During the discussions, a "new" model of procedure calling and some associated terminology was proposed. This new model is being drafted, and is included in the balloted draft of LIPC. The joint discussions corrected the definitions of these terms in the LIPC, and we believe these terms are useful to the definition of the RPC Model as well. We suggest:

a. Add:

   procedure image: A representation of a value of a particular procedure type, which embodies a particular set of instructions to be performed when the procedure is called.

b. Use "procedure image" to replace the current RPC term "procedure body instance", which is undefined.

c. Revise 3.5.21 to read:

   procedure closure: The object which represents the pair of a procedure image and a particular execution context for that procedure.

Item 1.7:	Remove "RPC" from terms which are shared with LIPC.
Type:	E
Ref:	3.5.9, 3.5.10, 3.5.14, 3.5.15, 3.5.16, 3.5.17, 3.5.30

The LIPC definitions of the terms in the above clauses are now identical, except for occurrences of the word "remote" or "RPC". Alignment will be facilitated by removal of those words in the RPC definitions.

Item 1.8:	LIPC vs. RPC terms
Type:	E

Ref: 3.5.17, 3.5.18, 3.5.19, 3.5.22, 3.5.29

The terms "interface-type", "interface-type definition", "IDN" "interface instance" and "interface-type identifier" are also terms in the LIPC, with the same generic meaning. In each case, the RPC meaning is restricted by requirements of the "remote"-ness. Somehow this relationship should be stated. We suggest RPC include the "general" definitions and then define the RPC specializations, using the form:

RPC xxx: A xxx (in) which ...

Item 1.9:	Interface-reference.
Type:	E
Ref:	3.5.28, 3.5.31

3.5.28 defines "RPC interface reference", while 3.5.31 refers to "interface-reference" without "RPC". They should be changed to match. The term "interface-reference" does not appear in LIPC, so removing "RPC" would not be a problem, but it does appear to be an RPC-specific notion.

Item 1.10:	Marshalling and Unmarshalling
Type:	E
Ref:	3.5.32, 3.5.33

The terms "marshalling" and "unmarshalling" are defined more generally in LIPC. Presumably, the notion "message" here means APDU. Here repeating the LIPC definition and adding an RPC-specific phrase within, or at the end of, the definition is preferred. E.g.:

marshalling: the process of collecting actual parameters, possibly converting them and assembling them (into a RPC message) for transfer.

unmarshalling: the process of disassembling the parameters transferred (in a RPC message), and possibly converting them, for use by the called procedure on invocation or the caller on return.

Item 1.11:	Clarify the "RPC client" notion.
Type:	E
Ref:	3.5.35

In the definition of "RPC client", it is not at all clear that "initiates ... a sequence of remote procedure calls" was intended to distinguish "top-level" calls from "callback" calls. If this is indeed what was intended, it should be stated explicitly. We suggest:

RPC client: A component in a distributed application which initiates a non-callback remote procedure call.

See also Item 1.16 below.

Item 1.12: Remove references to "computational semantics".

Type:	E
Ref:	3.5.37, 5.

The use of "computational semantics" in the RPC draft is gratuitous and subject to much argument in the SC22 community. There is no standard "computational model" to which the RPC can relate, nor is it clear that all ISO languages have the same one. The RPC model presents a semantic view of procedure calls which can be mapped to many programming language models. Delete the definition in 3.5.37, and the first sentence of the second paragraph of clause 5, which appears to be the only use.

Item 1.13:	Use LIPC Terms for Parameter-Passing.
Type:	E
Ref:	5.6.

LIPC defines four parameter passing styles, two, or possibly three, of which are clearly the ones intended by the wording in subclause 5.6 of Part 1. In the interest of aligning the RPC and LIPC Standards, RPC should use the parameter-passing terminology defined in LIPC, rather than attempting to word around a definition.

Value Sent on Initiation is a value passed to the called-procedure at the time of the call.

Value Returned on Termination is a value passed to the caller at the time of return, whether normal or some termination.

Value Sent on Request is a value passed to the called-procedure when some "out-of-band" request is made, e.g. when an aliased pointer is dereferenced.

Value Returned when Available is a value returned to the caller by some "out-of-band" request, e.g. when an assignment is made through an aliased pointer or a registered event is triggered.

It is not clear whether either of the latter two is actually supported by RPC. It is clear (and a requirement of the LIPC) that the first two must be.

Item 1.14:	Alignment of the Termination Model.
Type:	Note
Ref:	5.12

RPC and SYSTEM terminations are subdivisions of LIPC's (new) to split this class into RPC-defined and generated terminations and terminations defined and generated by other providers in the joint environment.

Marshalling errors are NOT RPC-specific terminations. There may be RPC-specific marshalling errors, but there may also be language-specific marshalling errors which are visible to an RPC user.

Item 1.15:	Alignment of Cancel.
Type:	E
Ref:	5.13

Cancel, as currently defined by RPC, is not required in LIPC, and it is not clear that LIPC should provide for it. The example given in 5.13 does not seem to have an LIPC analog - in LIPC the conceptual single thread of control is what is interrupted. Except for the unusual (RPC) situation in which the conceptual single thread of control is implemented by two actual threads on separate hosts. Cancel is only meaningful for "asynchronous" calls (in which there are conceptually multiple threads of control). At this time, LIPC describes "asynchronous calling" as out of scope, since it is a property of the call, not of the interface, and is only supported by a few programming languages (Modula2, Ada, and PL/I). RPC Part 1 is also totally silent on the subject of "asynchronous calls".

Thus Cancel is part of the RPC service model, but not a part of the procedure-call model. It should be described within RPC as a separate function provided by the RPC 'server' (although all the providers implementating the procedure calling function may be involved).

Item 1.16:	The Procedure Call Context
Type:	m
Ref:	5.5, 5.8

It is clear that the initiation of a client call creates something and that something is, in a larger sense, what an "interface reference" refers to. Specifically, LIPC introduces the notion of a "call context", which is conceptually created at the time of the (primary) call and is the (shared) context in which the called procedure is executed. Among other things, the call context contains the actual parameter values and other values accessible through them. The execution of a called-procedure in an interface instance depends on both the call context and the more static execution context.

It is specifically, the "call context" which distinguishes a primary call from a callback. Regardless of direction of the callback, the callback creates a call-context for the called procedure which is derived from the call context of the primary (client) call. By comparison, a primary call derives its call context solely from the (shared) execution context and the parameters of the call.

We suggest that RPC could benefit from the use of the call context notion in explaining the "interface reference" and the "client call" and "callback" concepts.

Item 1.17:	RPC calls cannot be completely transparent.
Type:	Note.
Ref:	5.

It is NOT the case that a remote procedure call can be made generally indistinguishable from a local (language-independent) call.

a. The absence of a truly shared execution context in the RPC case will necessarily create limitations on the handling of locally sharable objects such as files and pointer spaces.

b. In the callback situation, there may be a detectable difference in the "call context" which could cause a number of anomalous results. Consider the case where a procedure 'Sue' calls (remotely) a procedure 'Mary', which in turns makes a callback to a procedure 'Sam' (using a reference supplied explicitly or implicitly by Sue). The callback rules require Sam to behave as if it were called by 'Sue' in terms of its procedure context! It does not behave as if it were called by a local 'Mary', if the intermediate Mary context affects the execution of Sam.

Note that these limitations do not affect the wide applicability of RPC, but they do affect the degree to which it is "transparent". Note also that some programming languages may be sufficiently restrictive as to prevent these cases from occurring, and some LIPC implementations may have the same effects. The difference is that in the RPC case, the situation requires this behaviour, while the language and LIPC implementations can, but need not, behave differently.

Item 1.18:	Use "implementation-defined" and "implementation-dependent"
Type:	m
Ref:	5.

The following terms have been in use in ISO language standards for the last 10 years and make an important distinction in requirements:

Implementation-defined describes a capability, such as default character-set, or an event, such as data conversion error, which must be supported/provided/detected by all implementations, but whose details will vary from one implementation to another. Each implementation is required to document the details of its handling of implementation-defined features.

Implementation-dependent describes a capability, such as Cancel, or event, such as an invalid combination of parameters, which may be supported/provided/detected, but not necessarily in all cases, and is very much dependent on the details of the implementation. Such capabilities or events need not be documented in detail - they are considered to be "defined" as whatever the compiler or library does.

These terms are also used in the LIPC. We believe that the RPC conformance clauses would benefit from their use where appropriate. The SC22 definitions, with "provider" substituted for "language processor" are:

implementation-defined: Possibly differing between providers, but defined for any particular provider.

implementation-dependent: Possibly differing between providers, and not necessarily defined for any particular provider.

Comments on Part 2

Item 2.1:	Form of the Grammar in Part 2.
Type:	m
Ref:	7.

An agreement was reached among the experts at the joint meeting on meta-syntax for the IDN grammars in LID, LIPC and RPC. We agreed the following:

- use some end-of-production marker in all productions.
- put non-terminals in italics in text references.
- put all terminals in double-quotes (") in text references.
- within non-terminal identifiers, use hyphen and not underscore to separate multiple words.

It was agreed that LIPC will use the meta-notation conventions currently in LID. We strongly recommend that RPC use the same conventions. (We jointly considered and rejected use of BS 6143, because it is not in common use anywhere and is cumbersome.)

Item 2.2:	Reserved Words.
Type:	m
Ref:	8.1.

The joint discussions included the question of whether all terminal keywords should be "reserved", i.e. disallowed as identifiers. WG11 does not object to this, but the LI standards would reserve many more words immediately if all currently incorporated datatype names are reserved, and we have every reason to expect that additional standard datatypes will evolve in the future.

We suggest that only the keywords which MUST be distinguishable from identifiers should be reserved. That is, it is not necessary to reserve words which appear where an identifier is permitted, unless either the semantics of the keyword are entirely different from those of any identifier or the syntax which follows the keyword could not follow an identifier.

In particular, type-specifiers of the form:

<type-identifier> [(<parameters>)]

do not require reserved words, since the syntax does not depend on whether the type-identifier is predefined or user-defined. The pre-defined type-identifiers (e.g. integer, character, etc.) should be considered to be in the name space of type references, and the uniqueness rules of Part2, subclause 10.1.1 applied. Note that the datatypes of the parameter values, if any, can be determined (by lookup of the type-identifier) before they are scanned, although that should not be necessary to achieve a correct syntactic parse.

The predefined datatype generators (pointer, procedure, choice, record, array, and table in LIPC), on the other hand, introduce special syntax and therefore should be reserved words, in order to simplify the construction of parsers.

The LIPC generators set, sequence and bag, and user-defined generators should not be reserved words, since they can be treated (with a minor syntax change) as part of the type-reference name space. This avoids alignment problems resulting from a larger reserved-word list.

With the current grammar, the reserved words then should be: array, begin, choice, client?, default, end, excluding*, from, import, in, inout, interface, new*, of, out, plus*, pointer, procedure, raises, range*, record, restricted?, returns, selecting*, server?, size, subtype*, table*, termination, to, type, unaliased?, value.

where * indicates a keyword which appears only in the LIPC, and
? indicates a keyword which may only appear in the RPC.

(This list will undoubtedly change as comments on the three standards are resolved.)

Item 2.3:	Extensions aka Annotations.
Type:	m
Ref:	8.7

The differences between the RPC extensions and LIPC annotations were discussed. The following was agreed, and is suggested:

a. The square-brackets ([,]) are in the IRV of ISO 646, although they are reserved for national use. They should (continue to) be used for annotation delimiters, instead of the clumsy %(and)% which suddenly appeared in this draft of the RPC. If it is desired to avoid national use characters, the "pointy brackets" (<,>) could be used, but at the same time the "curly brackets" ({,}) (also national-use characters) used in OSI object-identifier values should be changed as well.

b. RPC could use the text from LID, subclause 7.4, last paragraph to address the editor's note at the end of RPC's subclause 8.7.

c. The meaning of the "extension" syntax is to annotate a particular grammatical element with additional information for a particular purpose, so the construct should be called an "annotation". It is possible that any of RPC, LIPC, LID or language-mapping standards may need (now or in the future) to define some such "annotations" (e.g. for marshalling instructions), and thus the term "extension", which implies "non-standard", is inappropriate.

d. We agree that object-identifier, with suitable explanation about abbreviations, is sufficient as a "source". It is desirable to omit the braces from the value syntax (ref. LID 7.4).

Item 2.4:	Interface-identifiers.
Type:	Note.
Ref:	9.1

LIPC allows interface identifiers to be object identifiers or simple identifiers to accomodate both RPC and local mechanisms. This means that, as in much of the LI Datatypes syntax, the RPC syntax for interface-identifier is a restriction of the "global" IDN.

Item 2.5:	Type declarations.
Type:	Note
Ref:	9.5

In 2nd CD LI Datatypes, type and generator declarations have been modified so that the RPC type-declaration is a proper subset.

Item 2.6:	Character datatypes.
Type:	M
Ref:	9.5.1.8

WG11 agrees with the spirit of the RPC changes to the character datatype, but disagrees with several details.

First, a character (or character string) can only come from one reference repertoire, if any sense is going to made of either its encoding or its abstract value. Therefore, the RPC notion of repertoire list is inaccurate. The list must be a list of subsets of a single reference repertoire, so that, among other things, overlaps can be handled (how many different SPACE characters are there?) and a consistent encoding can be determined.

Second, hard-coding a few values of the "repertoire-identifier" datatype in the grammar is a bad precedent, because implementations will inevitably define their own extensions in the same way. The object-identifier mechanism used for other such name spaces is a much better method of resolving the unambiguous extensible naming problem. In fact, the question of exactly which character sets will be supported by ALL implementations is more of an ISP issue than an application-layer standard issue, and use of object-identifier allows that decision to be deferred.

Third, special syntax for the ISO 10646 case is undesirable, because it makes the character-type syntax unusual, which in turn affects type-declaration and reserved words. We suggest that this case can be handled by the following assumption:

```
{ iso standard 10646 collection(0) <collection-name> }
```

is considered an object-identifier "inadvertently registered" by ISO 10646, which is itself the register. To satisfy purists, the syntax could read:

```
object-identifier-value = "{" object-id-components-list "}"
      | collection-identifier .
collection-identifier =
      "{ iso standard 10646 collection" collection-name "}".
```

We suggest incorporating the following syntax from LID (2cd) and using or paraphrasing the accompanying text:

```
character-type = "character" [ "(" repertoire-list ")" ] .
repertoire-list = repertoire-identifier { "," repertoire-identifier } .
repertoire-identifier = value-expression .
```

"The value-expression for a repertoire-identifier shall designate a value of the object-identifier datatype, and that value shall refer to a character-set. All repertoire-identifiers in the repertoire-list shall designate subsets of a single reference character-set. When repertoire-list is not specified, it shall have a default value. The means for specification of the default is outside the scope of this draft International Standard.

"The value space of a character datatype comprises exactly the members of the character-sets identified by the repertoire-list. In cases where the character-sets identified by the individual repertoire-identifiers have members in common, the value space of the character datatype contains only the distinct members."

To satisfy ease-of-use concerns for RPC/LIPC, the "latin", "greek", "cyrillic", "japanese" identifiers can be (implicitly or explicitly) declared in value-declarations, such as:

```
value latin: object_identifier = { iso standard 8859 1 };
```

This also serves to provide a "good example" for extensions.

Item 2.7:	Syntax bug for char_type.
Type:	m
Ref:	9.5.1.3

The production for <char_type> should be:

```
<char_type> ::= character of ( <repertoire-list> )
```

Omission of the parentheses causes <repertoire-list> to be ambiguous whenever there is more than one <repertoire> in it.

We also recommend deletion of the keyword "of".

Item 2.8:	Interface-reference.
Type:	Note.
Ref:	9.5.1.8

It is our understanding that "interface-reference" is an RPC-specific datatype, which does not appear to generalize to the LIPC context (or any other). "interface-reference" therefore will not appear in LI Datatypes nor LIPC.

Item 2.9:	Choice-type and Select-type alignment.
Type:	M
Ref:	9.5.2.2

In spite of agreement in Arles on what in general was wanted for the choice-type, there are now significant differences between the RPC syntax and model and the LID/LIPC syntax and model. The model of the choice-type is still a subject of debate in both RPC and WG11, so further discussion is needed to achieve alignment.

a. RPC changed the name of the datatype to select-type, for reasons which are unclear. (The term choice-type has been in use in both standards up until now.) The keyword "SELECT" would be a minor problem for users of either, since they also both contain the keyword "SELECTING". Moreover, the keyword CHOICE is that used in ASN.1 to name what is conceptually the same datatype.

b. RPC describes the "discriminant" field as a value-reference, intending to obtain the tag-type via the object referred to. This syntax permits (whether intentionally or not) the discriminant to be a constant value, since a value reference could be a value-identifier or a value of an enumerated type or a formal datatype parameter. While this is technically not a problem, we assume that the intent was that the discriminant be what LID calls a "dependent-value", i.e. a reference to another argument or field. LID/LIPC require the type-specifier for the tag-type here, not a reference to a particular value thereof, because the tag-type is a component of the datatype, while a particular value of the tag-type is at best a component of a particular value of the choice-type.

c. RPC requires the alternatives to have both selection values (<subtype-spec>) and <field-names>, except for void. The LID/LIPC does not provide for field-names. WG11 does not object to this syntax, if the motivation for the names could be described.

d. LID/LIPC modified the syntax to force the default-alternative to be last, in order to satisfy a liaison request from SC22 WG17. It is believed that this simplifies mapping to some languages. We request that RPC do the same.

Item 2.10:	Require alternatives to cover the tag-type.
Type:	m
Ref:	9.5.2.2

It should be stated that if a default alternative is not present, that the supplied alternative subtypes must completely cover the value space of the discriminant. Otherwise, there are situations in which the choice value is undefined.

Item 2.11:	Termination Parameters
Type:	m
Ref:	9.7 or 10.1

Any formal relationship (based on names, etc.) between termination parameters and "normal" arguments of a procedure (-type) makes it impossible to define standard terminations without risk of accidental association with user-defined parameters. The following requirement should be added to 9.7 or 10.1:

Parameters in terminations are distinct semantically from each other and from ALL OTHER output or input/output parameters in the interaction model. That is, there shall not be any formal relationship between arguments in a termination-list and arguments in a procedure argument-list. (There may be implicit relationships for specific terminations, i.e. the definition of a termination may specify that the 3rd parameter of the termination will have the same value as the second input argument or as would have been supplied to the first output argument on a normal termination.)

Item 2.12:	Unaliased Pointers.
Type:	m
Ref:	9.5.2.4

The notion "unaliased" does not describe the pointer datatype itself, nor any generic property of or limitation on values of the type. Rather, it describes a state of the execution context at the time of a particular call on a procedure.

Declaring a pointer "unaliased" means that the actual pointer value passed for this formal parameter is guaranteed at the time of the call not to name a "box" which is accessible by any other means available to the called procedure, including (but not restricted to!) other values of pointer type passed as, or through, other arguments to the same procedure. It is neither necessary nor sufficient to require that there be no other instance of that particular pointer "label" anywhere within the caller's execution context, if that is what "not statically or dynamically aliased with any other pointer" means. There is no problem with the caller having copies of the pointer value in multiple places; it is only necessary that the called procedure not have access to any of them. On the other hand, it is not sufficient that the label isn't duplicated anywhere. For example, if the pointer value refers to a component of an array or record and the entire aggregate is referenced by (or through) another pointer argument, the "labels" may be different, but simultaneous access is possible. Even worse is the possibility that the called procedure can callback a client procedure that uses name space access to the same "box" in the client execution context. In RPC, the context shared between the two open systems only includes the arguments; but in LIPC, and therefore in RPC callback cases, the context shared between two procedures on the same open system may include some form of "global variable" as well.

Thus, like "in" and "out", "unaliased" is a property attached to a procedure parameter that goes beyond its datatype to refer to a state of the shared context. (See WG11 Nxxx - the Box and Context model.) For this reason, LI Datatypes will not include "unaliased", but LIPC will. The syntax is acceptable, but the definition should be revised to something like:

"in" and "unaliased" means that any actual pointer value passed in this position is guaranteed at the time of the call not to name a "box" which is accessible by any other means to the called procedure, including any procedure the called procedure may reference.

"out" and "unaliased" means that any actual pointer value passed in this position is guaranteed at the time of the return not to name a "box" which is accessible by any other means to the caller.

Item 2.13:	Restricted Pointers.
Type:	m
Ref:	9.5.2.4

WG11 is uncertain what "restricted" is intended to mean. The limitation on the value space of restricted pointer values is handled by the LI Datatypes syntax "excluding(null)". Does "restricted" mean exactly "unaliased, excluding(null)"? If so, why is it stated that restricted pointers "can be supported efficiently"? It appears that they should be neither more nor less efficient than unaliased pointers.

We suspect that almost all uses of "restricted pointer" are artifacts of implementation.

Some programming languages, such as C, do not provide for the manipulation of values record-types or array-types as a unit, and do not permit them to be passed by value or returned as the principal result. As a consequence the programmer is required to manipulate "pointer to (R)" for what is conceptually just "R". The IDN user should declare such a parameter to be of type "R" and let the C mapping (and marshalling) handle the fact that the corresponding C datatype in the program will be R* (pointer to (R)).

By the same token, many compilers optimize the passing of an array or record value in an implementation which passes the address of the first cell and lets the compiled subprogram manipulate the value via the (implicit) pointer. Clearly this is an issue for the marshalling routines for RPC, but it should not enter into the IDN, except possibly as an annotation.

Finally, there is the case in which the user, for purposes of efficient storage management, builds a (tree) structure with pointers, even though each node logically "contains" all nodes subordinate to it. (This is again a language limitation: in LISP the programmer manipulates the subordinate nodes as "contained", while the compiler builds the pointer structure.) The contained datatype then logically has the form: Choice(T, Void) or Set of (T), but the implementation has the form: Pointer to (T). This is an interesting and debatable case, but it could NOT be a use of "restricted" because the efficiency and the success of the mechanism depend on allowing some values of the pointer to be "null".

Item 2.14:	Value-expressions.
Type:	m
Ref:	9.4, 10.3

In the three CDs, the language generated by the RPC syntax and the language generated by the LIPC/LID syntax with respect to "value-expression" (<value_expr>) are different in minor ways, and the formal syntaxes do not much resemble one another. The LIPC/LID syntax should be expanded to permit

interface-identifier :: value-identifier,

in order to allow the RPC language to be a proper subset, but this alone will not resolve the significant differences in syntactic approach. Some intermediate level of common syntax should be sought.

Item 2.15: Object-identifier
Type: m
Ref: 9.4, 9.5

For reasons which are not clear, RPC does not permit the datatype Object-Identifier to be the datatype of any parameter, nor does it permit a value-declaration to name a value of object-identifier-type. The modifications to character-type make the latter feature desirable, and type-declarations for character-types might use object-identifier values as parameters, necessitating the former. Moreover, it is reasonable to suppose that RPC applications may have other reasons for manipulating the object-identifier datatype, since it is a commonly occurring datatype in ASN.1.

We suggest:

a. add the alternative "| <object-identifier-type>" to the production for primitive-type. And add a section describing the type to 9.5.1, with the production:

   object-identifier-type = "object_identifier" .

b. add the alternative "| <object-identifier-value>" to the production for value-expression, and replace all occurrences of <object-identifier> with <object-identifier-value>. (The reason for the last is to avoid ambiguity of the term "object-identifier" and to be consistent with other LIPC/RPC naming. Every other xxx-identifier produces <identifier>.)

c. move the syntax and definition of object-identifier from 9.9 to 9.4.

d. correct the production for object-id-component. The second alternative should be "<digit> ..." and not just "<digit>". That is, ASN.1 permits a "digit-string" or "number" here.