UNION INTERNATIONALE DES TÉLÉCOMMUNICATIONS

UIT-T

SECTEUR DE LA NORMALISATION
DES TÉLÉCOMMUNICATIONS
DE L'UIT

X.920

(12/97)

SÉRIE X: RÉSEAUX POUR DONNÉES ET COMMUNICATION ENTRE SYSTÈMES OUVERTS

Traitement réparti ouvert

Technologies de l'information – Traitement réparti ouvert – Langage de définition d'interface

Recommandation UIT-T X.920

(Antérieurement Recommandation du CCITT)

RECOMMANDATIONS UIT-T DE LA SÉRIE X

RÉSEAUX POUR DONNÉES ET COMMUNICATION ENTRE SYSTÈMES OUVERTS

RÉSEAUX PUBLICS POUR DONNÉES	
Services et fonctionnalités	X.1–X.19
Interfaces	X.20–X.49
Transmission, signalisation et commutation	X.50–X.89
Aspects réseau	X.90–X.149
Maintenance	X.150–X.179
Dispositions administratives	X.180–X.199
INTERCONNEXION DES SYSTÈMES OUVERTS	
Modèle et notation	X.200–X.209
Définitions des services	X.210–X.219
Spécifications des protocoles en mode connexion	X.220–X.229
Spécifications des protocoles en mode sans connexion	X.230–X.239
Formulaires PICS	X.240–X.259
Identification des protocoles	X.260–X.269
Protocoles de sécurité	X.270–X.279
Objets gérés des couches	X.280–X.289
Tests de conformité	X.290–X.299
INTERFONCTIONNEMENT DES RÉSEAUX	
Généralités	X.300–X.349
Systèmes de transmission de données par satellite	X.350–X.399
SYSTÈMES DE MESSAGERIE	X.400–X.499
ANNUAIRE	X.500–X.599
RÉSEAUTAGE OSI ET ASPECTS SYSTÈMES	
Réseautage	X.600–X.629
Efficacité	X.630–X.639
Qualité de service	X.640–X.649
Dénomination, adressage et enregistrement	X.650–X.679
Notation de syntaxe abstraite numéro un (ASN.1)	X.680–X.699
GESTION OSI	
Cadre général et architecture de la gestion-systèmes	X.700–X.709
Service et protocole de communication de gestion	X.710–X.719
Structure de l'information de gestion	X.720–X.729
Fonctions de gestion et fonctions ODMA	X.730–X.799
SÉCURITÉ	X.800–X.849
APPLICATIONS OSI	
Engagement, concomitance et rétablissement	X.850–X.859
Traitement transactionnel	X.860–X.879
Opérations distantes	X.880–X.899
TRAITEMENT RÉPARTI OUVERT	**X.900–X.999**

Pour plus de détails, voir la Liste des Recommandations de l'UIT-T.

NORME INTERNATIONALE 14750

RECOMMANDATION UIT-T X.920

TECHNOLOGIES DE L'INFORMATION – TRAITEMENT RÉPARTI OUVERT – LANGAGE DE DÉFINITION D'INTERFACE

Résumé

La présente Recommandation | Norme internationale spécifie un langage de définition d'interface (IDL, *interface definition language*) pour les spécifications qui sont conformes au langage de calcul défini dans l'architecture du modèle de référence pour le traitement réparti ouvert (ODP, *open distributed processing*) (voir la Rec. UIT-T X.903 | ISO/CEI 10746-3). Le langage IDL permet de décrire des interfaces entre objets ainsi que leurs opérations et leurs paramètres associés. Il est totalement aligné sur le langage IDL – architecture de gestionnaire de requêtes pour objets communs (CORBA, *common object request broker architecture*) qui a été mis au point par le groupe de gestion d'objets (OMG, *object management group*).

Source

La Recommandation X.920 de l'UIT-T a été approuvée le 12 décembre 1997. Un texte identique est publié comme Norme internationale ISO/CEI 14750.

La Recommandation UIT-T X.920 résulte de l'adoption du texte des spécifications du langage de définition d'interface (IDL) du Groupe de gestion d'objets (OMG), pour lesquelles les droits de distribution internationaux et les droits sur les travaux dérivés restent la propriété du groupe OMG.

AVANT-PROPOS

L'UIT (Union internationale des télécommunications) est une institution spécialisée des Nations Unies dans le domaine des télécommunications. L'UIT-T (Secteur de la normalisation des télécommunications) est un organe permanent de l'UIT. Il est chargé de l'étude des questions techniques, d'exploitation et de tarification, et émet à ce sujet des Recommandations en vue de la normalisation des télécommunications à l'échelle mondiale.

La Conférence mondiale de normalisation des télécommunications (CMNT), qui se réunit tous les quatre ans, détermine les thèmes d'études à traiter par les Commissions d'études de l'UIT-T, lesquelles élaborent en retour des Recommandations sur ces thèmes.

L'approbation des Recommandations par les Membres de l'UIT-T s'effectue selon la procédure définie dans la Résolution n° 1 de la CMNT.

Dans certains secteurs des technologies de l'information qui correspondent à la sphère de compétence de l'UIT-T, les normes nécessaires se préparent en collaboration avec l'ISO et la CEI.

NOTE

Dans la présente Recommandation, l'expression "Administration" est utilisée pour désigner de façon abrégée aussi bien une administration de télécommunications qu'une exploitation reconnue.

DROITS DE PROPRIÉTÉ INTELLECTUELLE

L'UIT attire l'attention sur la possibilité que l'application ou la mise en œuvre de la présente Recommandation puisse donner lieu à l'utilisation d'un droit de propriété intellectuelle. L'UIT ne prend pas position en ce qui concerne l'existence, la validité ou l'applicabilité des droits de propriété intellectuelle, qu'ils soient revendiqués par un Membre de l'UIT ou par une tierce partie étrangère à la procédure d'élaboration des Recommandations.

A la date d'approbation de la présente Recommandation, l'UIT n'avait pas été avisée de l'existence d'une propriété intellectuelle protégée par des brevets à acquérir pour mettre en œuvre la présente Recommandation. Toutefois, comme il ne s'agit peut-être pas de renseignements les plus récents, il est vivement recommandé aux responsables de la mise en œuvre de consulter la base de données des brevets du TSB.

© UIT 1999

Droits de reproduction réservés. Aucune partie de cette publication ne peut être reproduite ni utilisée sous quelque forme que ce soit et par aucun procédé, électronique ou mécanique, y compris la photocopie et les microfilms, sans l'accord écrit de l'UIT.

TABLE DES MATIÈRES

		Page
1	Domaine d'application	1
2	Références normatives	1
2.1	Recommandations | Normes internationales identiques	1
2.2	Autres références	1
3	Définitions	1
4	Syntaxe et sémantique du langage IDL ODP	2
4.1	Conventions lexicales	2
4.2	Prétraitement	7
4.3	Grammaire du langage IDL ODP	8
4.4	Spécification IDL ODP	12
4.5	Héritage	14
4.6	Déclaration de constante	15
4.7	Déclaration d'un type	17
4.8	Types "Typecode" et "Principal"	23
4.9	Déclaration d'exception	23
4.10	Déclaration d'opération	24
4.11	Déclaration d'attribut	25
4.12	Module de spécification CORBA	26
4.13	Noms et domaines de visibilité	26
4.14	Différences par rapport au langage C++	28
Annexe A – Exceptions normalisées et réservées		29
A.1	Non-existence d'objet	30
A.2	Exceptions de transaction	30
Annexe B – Codage des types dans la spécification CORBA		31

Introduction

Le développement rapide du traitement réparti a fait naître le besoin d'un cadre de coordination pour la normalisation du traitement réparti ouvert (ODP). Le modèle de référence du traitement réparti ouvert (RM-ODP, *reference model of open distributed processing*) fournit un tel cadre. Il définit une architecture au sein de laquelle la mise en œuvre de la répartition, de l'interopérabilité et de la portabilité peut être intégrée.

Un des composants de l'architecture (décrit dans la partie 3 du modèle de référence RM-ODP: Architecture) (Rec. UIT-T X.903 | ISO/CEI 10746-3) est un langage adapté à la description de la signature des interfaces d'opération de calcul. La présente Recommandation | Norme internationale contient un tel langage de définition d'interface, appelé IDL ODP.

NOTE – La présente Recommandation | Norme internationale est techniquement alignée sur la spécification du langage de définition d'interface CORBA.

L'Annexe A est normative et fournit un ensemble normalisé d'exceptions pour une infrastructure de répartition ODP particulière.

L'Annexe B est informative et fournit le codage CORBA d'un type, appelé TypeCode, représentant des descriptions de type.

NORME INTERNATIONALE

RECOMMANDATION UIT-T

TECHNOLOGIES DE L'INFORMATION – TRAITEMENT RÉPARTI OUVERT – LANGAGE DE DÉFINITION D'INTERFACE

1 Domaine d'application

La présente Recommandation | Norme internationale est destinée à donner au modèle de référence du traitement ODP (voir la Rec. UIT-T X.902 | ISO/CEI 10746-2 et la Rec. UIT-T X.903 | ISO/CEI 10746-3) un langage et une notation d'environnement neutre, afin de décrire les signatures des interfaces d'opérations de calcul. L'utilisation de cette notation ne suppose pas l'utilisation de mécanismes supports ou de protocoles particuliers.

2 Références normatives

Les Recommandations et les Normes internationales suivantes contiennent des dispositions qui, par suite de la référence qui y est faite, constituent des dispositions valables pour la présente Recommandation | Norme internationale. Au moment de la publication, les éditions indiquées étaient en vigueur. Toutes Recommandations et Normes sont sujettes à révision et les parties prenantes aux accords fondés sur la présente Recommandation | Norme internationale sont invitées à rechercher la possibilité d'appliquer les éditions les plus récentes des Recommandations et Normes indiquées ci-après. Les membres de la CEI et de l'ISO possèdent le registre des Normes internationales en vigueur. Le Bureau de la normalisation des télécommunications de l'UIT tient à jour une liste des Recommandations UIT-T en vigueur.

2.1 Recommandations | Normes internationales identiques

– Recommandation UIT-T X.902 (1995) | ISO/CEI 10746-2:1996, *Technologies de l'information – Traitement réparti ouvert – Modèle de référence: Fondements.*

– Recommandation UIT-T X.903 (1995) | ISO/CEI 10746-3:1996, *Technologies de l'information – Traitement réparti ouvert – Modèle de référence: Architecture.*

2.2 Autres références

– ISO/CEI 646:1991, *Technologies de l'information – Jeu ISO de caractères codés à 7 éléments pour l'échange d'informations.*

– ISO/CEI 8859-1:1998, *Technologies de l'information – Jeux de caractères graphiques codés sur un seul octet – Partie 1: Alphabet latin nº 1.*

3 Définitions

Pour les besoins de la présente Recommandation | Norme internationale, les définitions suivantes s'appliquent.

La présente Recommandation | Norme internationale utilise les termes suivants, qui sont définis dans la Rec. UIT-T X.902 | ISO/CEI 10746-2:

– objet;

– interface;

– signature d'interface.

La présente Recommandation | Norme internationale utilise le terme suivant, qui est défini dans la Rec. UIT-T X.903 | ISO/CEI 10746-3:

– opération.

4 Syntaxe et sémantique du langage IDL ODP

Le langage de définition d'interface (*interface definition language*) IDL ODP sert à décrire les signatures des interfaces que les objets clients appellent et que les mises en œuvre d'objet fournissent. Une définition d'interface écrite en langage IDL ODP définit complètement la signature de cette interface et spécifie complètement les paramètres de chaque opération.

Une spécification IDL ODP se compose logiquement d'un ou de plusieurs fichiers. Un fichier est théoriquement traduit en plusieurs phases, dont la première est le prétraitement, qui effectue l'inclusion du fichier et la substitution des macros. Le prétraitement est régi par des directives qui sont introduites par des lignes ayant le dièse (#) comme premier caractère autre qu'un espace. Le résultat du prétraitement est une séquence de jetons qui est appelée "unité de traduction" et qui constitue un fichier après le prétraitement.

Le langage IDL ODP obéit aux mêmes règles lexicales que le langage C++[1)], bien que de nouveaux mots clés soient introduits pour prendre en charge les concepts de répartition. Il assure également la prise en charge complète des caractéristiques de prétraitement en langage C++ normal. La spécification IDL ODP est appelée à suivre les modifications concernant le langage C++, introduites par le groupe ISO/CEI chargé de la normalisation.

La description des conventions lexicales du langage IDL ODP est présentée au 4.1. Une description du prétraitement en langage IDL ODP est présentée au 4.2. Les règles de visibilité des identificateurs dans une spécification de langage IDL ODP sont décrites à partir du 4.13.

La grammaire du langage IDL ODP est un sous-ensemble du langage ISO/CEI C++ avec des créations syntaxiques additionnelles pour prendre en charge le mécanisme d'invocation d'opération. Le langage IDL ODP est un langage descriptif qui prend en charge la syntaxe C++ pour des déclarations de constantes, de types et d'opérations; il ne comporte aucune structure ou variable algorithmique. Sa grammaire est présentée au 4.3.

Le présent article décrit la sémantique du langage IDL ODP et donne la syntaxe des créations grammaticales IDL ODP. La description de la grammaire IDL ODP fait appel à une notation syntaxique qui est similaire à l'extension du formalisme de Backus-Naur (EBNF, *extended Backus-Naur format*). Le Tableau 1 énumère les symboles utilisés dans ce formalisme et leur signification.

Tableau 1 – Formalisme EBNF du langage IDL ODP

Symbole	Signification
::=	défini par:
|	ou, en variante:
<text>	Non-terminal
"text"	Littéral
*	L'unité syntaxique précédente peut être répétée zéro, une ou plusieurs fois
+	L'unité syntaxique précédente peut être répétée une ou plusieurs fois
{ }	Les unités syntaxiques englobées sont groupées en une seule unité syntaxique
[]	L'unité syntaxique précédente est facultative et peut apparaître zéro, une ou plusieurs fois

4.1 Conventions lexicales

Le présent paragraphe[2)] présente les conventions lexicales du langage IDL ODP. Il définit les jetons contenus dans une spécification IDL ODP et décrit les commentaires, les identificateurs, les mots clés et les littéraux sous forme d'entiers, de caractères, de constantes à virgule flottante, ainsi que de chaînes.

---

1) Ellis, Margaret A. et Bjarne Stroustrop, *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1990, ISBN 0-201-51459-1.

2) Le présent paragraphe est une adaptation du chapitre 2 du livre intitulé *The Annotated C++ Reference Manual*; il en diffère quant à la liste des mots clés autorisés et quant à la ponctuation.

Le langage IDL ODP fait appel au jeu de caractères ISO/CEI Latin-1 (ISO/CEI 8859-1), qui se subdivise en caractères alphabétiques (lettres), en chiffres, en caractères graphiques et en caractères de formatage, avec le caractère d'espacement. Le Tableau 2 montre les caractères alphabétiques du langage IDL ODP; les majuscules/minuscules équivalentes sont présentées en paires. Le Tableau 3 montre les chiffres et le Tableau 4 montre les caractères graphiques.

Les caractères de formatage sont reproduits dans le Tableau 5.

Tableau 2 – Les 114 caractères alphabétiques (lettres)

Caractère	Description	Caractère	Description
Aa	A majuscule/minuscule	Àà	A accent grave majuscule/minuscule
Bb	B majuscule/minuscule	Áá	A accent aigu majuscule/minuscule
Cc	C majuscule/minuscule	Ââ	A accent circonflexe majuscule/minuscule
Dd	D majuscule/minuscule	Ãã	A tilde majuscule/minuscule
Ee	E majuscule/minuscule	Ää	A tréma majuscule/minuscule
Ff	F majuscule/minuscule	Åå	A majuscule/minuscule avec rond
Gg	G majuscule/minuscule	Ææ	A E entrelacés majuscule/minuscule
Hh	H majuscule/minuscule	Çç	C cédille majuscule/minuscule
Ii	I majuscule/minuscule	Èè	E accent grave majuscule/minuscule
Jj	J majuscule/minuscule	Éé	E accent aigu majuscule/minuscule
Kk	K majuscule/minuscule	Êê	E accent circonflexe majuscule/minuscule
Ll	L majuscule/minuscule	Ëë	E tréma majuscule/minuscule
Mm	M majuscule/minuscule	Ìì	I accent grave majuscule/minuscule
Nn	N majuscule/minuscule	Íí	I accent aigu majuscule/minuscule
Oo	O majuscule/minuscule	Îî	I accent circonflexe majuscule/minuscule
Pp	P majuscule/minuscule	Ïï	I tréma majuscule/minuscule
Qq	Q majuscule/minuscule	Ðð	D barré majuscule/minuscule
Rr	R majuscule/minuscule	Ññ	N tilde majuscule/minuscule
Ss	S majuscule/minuscule	Òò	O accent grave majuscule/minuscule
Tt	T majuscule/minuscule	Óó	O accent aigu majuscule/minuscule
Uu	U majuscule/minuscule	Ôô	O accent circonflexe majuscule/minuscule
Vv	V majuscule/minuscule	Õõ	O tilde majuscule/minuscule
Ww	W majuscule/minuscule	Öö	O tréma majuscule/minuscule
Xx	X majuscule/minuscule	Øø	O barre oblique majuscule/minuscule
Yy	Y majuscule/minuscule	Ùù	U accent grave majuscule/minuscule
Zz	Z majuscule/minuscule	Úú	U accent aigu majuscule/minuscule
		Ûû	U accent circonflexe majuscule/minuscule
		Üü	U tréma majuscule/minuscule
		Ýý	Y accent aigu majuscule/minuscule
		Þþ	th islandais majuscule/minuscule
		ß	Eszett allemand
		ÿ	Y tréma minuscule

Tableau 3 – Chiffres décimaux

0 1 2 3 4 5 6 7 8 9

Tableau 4 – Les 65 caractères graphiques

Caractère	Description	Caractère	Description
!	point d'exclamation	¡	point d'exclamation inversé
"	guillemet droit	¢	signe cent
#	signe dièse	£	signe livre
$	signe dollar	¤	signe monétaire
%	signe pour cent	¥	signe yen
&	perluète	¦	barre verticale interrompue
'	apostrophe	§	signe section ou paragraphe
(	parenthèse gauche	¨	tréma
)	parenthèse droite	©	signe droits d'auteurs
*	astérisque	ª	indicateur ordinal, féminin
+	signe plus	«	guillemet anguleux gauche
,	virgule	¬	signe de négation
-	tiret, signe moins	–	tiret automatique
.	point	®	signe marque déposée
/	barre oblique	¯	signe de voyelle longue
:	deux-points	°	rond, signe du degré
;	point-virgule	±	signe plus ou moins
<	signe inférieur à	^{2}	exposant deux
=	signe égal	^{3}	exposant trois
>	signe supérieur à	´	accent aigu
?	point d'interrogation	µ	symbole micro
@	signe arroba	¶	signe alinéa
[	crochet gauche	·	point médian
\	barre oblique renversée	¸	cédille
]	crochet droit	^{1}	exposant un
^	accent circonflexe	º	indicateur ordinal masculin
_	tiret inférieur ou de soulignement	»	guillemet anguleux droit
`	accent grave	¼	fraction un quart
{	accolade gauche	½	fraction un demi
|	barre verticale	¾	fraction trois quarts
}	accolade droite	¿	point d'interrogation inversé
~	tilde	×	signe de multiplication
		÷	signe de division

Tableau 5 – Caractères de formatage

Description	Abréviation	Valeur octale ISO/CEI 646
sonnette	BEL	007
retour arrière	BS	010
tabulateur horizontal	HT	011
alinéa	NL, LF	012
tabulateur vertical	VT	013
saut de page	FF	014
retour chariot	CR	015

4.1.1 Jetons

Il existe cinq sortes de jetons: les identificateurs, les mots clés, les littéraux, les opérateurs et les séparateurs. Les espaces blancs (réserves), les tabulations horizontales et verticales, les retours à la ligne, les sauts de page et les commentaires (collectifs ou en "réserve"), décrits ci-dessous, ne sont pas pris en considération sauf s'ils servent à séparer des jetons. Une certaine réserve est nécessaire pour séparer des identificateurs, des mots clés et des constantes, qui autrement seraient contigus.

Si le flux entrant a été analysé sémantiquement pour en extraire les jetons jusqu'à un caractère donné, le jeton suivant est considéré comme étant formé de la plus longue chaîne de caractères qui peut éventuellement constituer un jeton.

4.1.2 Commentaires

Les caractères barre oblique et astérisque /* marquent le début d'un commentaire, qui se termine par un astérisque et par une barre oblique */. Ces commentaires ne s'imbriquent pas. Les caractères // marquent le début d'un commentaire qui se termine à la fin de la ligne où ils apparaissent. Les caractères de commentaire //, /* et */ n'ont pas de signification spéciale à l'intérieur d'un commentaire introduit par les caractères // et sont traités comme n'importe quels autres caractères. De même, les caractères de commentaire // et /* n'ont pas de signification spéciale à l'intérieur d'un commentaire introduit par les caractères /*. Les commentaires peuvent contenir des caractères alphabétiques, numériques et graphiques, des espaces, des tabulations horizontales et verticales, des sauts de page et des caractères de retour à la ligne.

4.1.3 Identificateurs

Un identificateur est une séquence de longueur arbitraire, composée de caractères alphabétiques, numériques et graphiques ("_"). Le premier caractère est toujours alphabétique. Tous les caractères sont significatifs.

Les identificateurs qui ne diffèrent que par leur hauteur de casse entrent en collision et provoquent une erreur de compilation. Un identificateur de définition doit toujours être orthographié de la même manière (en termes de hauteur de casse) d'un bout à l'autre d'une spécification.

Lorsque l'on compare deux identificateurs pour voir s'ils sont en collision:

- les lettres majuscules et minuscules sont traitées comme étant la même lettre. Le Tableau 2 définit le mappage des lettres majuscules et minuscules équivalentes;
- la comparaison ne tient *pas* compte des équivalences entre digrammes et paires de lettres (par exemple, les caractères "æ" et "ae" ne sont pas considérés comme équivalents); elle ne tient pas compte non plus des équivalences entre lettres accentuées et non accentuées (par exemple, les caractères "à" et "a" ne sont pas considérés comme équivalents).
- tous les caractères sont significatifs.

En langage IDL ODP, chaque identificateur ne possède qu'un seul espace de dénomination. Par exemple, l'utilisation du même identificateur pour une constante et pour une interface produit une erreur de compilation.

4.1.4 Mots clés

Les identificateurs énumérés dans le Tableau 6 sont réservés pour utilisation comme mots clés et ne peuvent avoir d'autre usage. Le mot clé `Object` est utilisé en langage IDL ODP pour représenter un type d'interface, tandis que le mot clé `interface` sert à indiquer le début d'une déclaration d'interface dans un modèle de signature d'interface. Le mot clé `Object` peut être utilisé comme un spécificateur de type. Le mot clé `attribut` définit une méthode donnant accès à une portion de l'état d'un objet. Une définition d'attribut est logiquement équivalente à la déclaration d'une paire de méthodes d'accès: l'une pour récupérer la valeur de l'attribut, l'autre pour fixer cette valeur.

Le mot clé Exception est utilisé pour représenter le concept ODP de terminaison nommée sans succès, le nom de l'exception étant le nom de la terminaison.

Les mots clés obéissent aux règles relatives aux identificateurs (voir 1.3) et doivent être écrits exactement comme indiqué dans la liste ci-dessus. Par exemple, le mot "`boolean`" est correct, tandis que le mot "`Boolean`" ne l'est pas. Les spécifications du langage IDL ODP utilisent les caractères de ponctuation indiqués dans le Tableau 7.

En outre, les jetons énumérés dans le Tableau 8 sont utilisés par le préprocesseur.

Tableau 6 – Mots clés

any	default	in	oneway	struc	wchar
attribute	double	inout	out	switch	wstring
boolean	enum	interface	raises	TRUE	
case	exception	long	readonly	typedef	
char	FALSE	module	sequence	unsigned	
const	fixed	Object	short	union	
context	float	octet	string	void	

Tableau 7 – Jetons de ponctuation

;	{	}	:	::	,	=	+	-	(	)	<	>	[	]
’	"	\	|	^	&	*	/	%	~	<<	>>			

Tableau 8 – Jetons utilisés par le préprocesseur

#	##	!	||	&&	include	pragma	define

4.1.5 Littéraux

Les littéraux caractère large et chaîne large sont spécifiés exactement comme les littéraux caractère et chaîne. Tous les littéraux caractère et chaîne, qu'ils soient larges ou non, peuvent (pour être portables) être spécifiés en utilisant les caractères du jeu de caractères ISO/CEI 8859-1. Il convient de noter que ces extensions pour les caractères internationaux affectent seulement la spécification des littéraux dans le langage IDL ODP et pas le reste des fichiers source IDL ODP. C'est-à-dire que les noms d'interfaces, les noms d'opérations, les noms de types, etc., continueront d'être restreints au jeu de caractères ISO/CEI 8859-1.

Les littéraux des nouveaux types entier et à virgule flottante sont définis comme cela est décrit dans le présent paragraphe (*littéraux sous forme d'entier* et *littéraux à virgule flottante*).

4.1.5.1 Littéraux sous forme d'entier

Un littéral-entier, composé d'une séquence de chiffres, est considéré comme étant décimal (en base dix) s'il ne commence pas par un 0 (chiffre zéro). Une séquence de chiffres commençant par 0 est considérée comme étant un entier octal (en base 8). Les chiffres 8 et 9 ne sont pas des chiffres octaux. Une séquence de chiffres précédée de 0x ou 0X est considérée comme étant un entier hexadécimal (en base seize). Les chiffres hexadécimaux vont de a ou A (qui a la valeur dix) jusqu'à f ou F (qui a la valeur quinze). Par exemple, le nombre douze peut s'écrire 12, 014 ou 0XC.

4.1.5.2 Littéraux sous forme de caractères

Un littéral-caractère se présente sous la forme d'un ou de plusieurs caractères encadrés par des apostrophes, comme dans 'x'. Les littéraux de caractère sont du type `char`.

Un caractère est une grandeur de 8 bits dont la valeur numérique est comprise entre 0 et 255 (en base dix). La valeur d'un littéral-caractère alphabétique, numérique, graphique ou d'espace est la valeur numérique du caractère correspondant, telle que définie dans le jeu de caractères normalisé ISO/CEI Latin-1 (ISO/CEI 8859-1). Voir les Tableaux 2, 3 et 4. La valeur d'élément nul est 0. La valeur d'un littéral-caractère de formatage est la valeur numérique du caractère correspondant, telle que définie dans l'ISO/CEI 646 (voir le Tableau 5). La signification de tous les autres caractères dépend de la mise en œuvre.

Les caractères non graphiques doivent toujours être représentés au moyen de séquences d'échappement, comme défini dans le Tableau 9 ci-dessous. On notera que les séquences d'échappement doivent être utilisées pour représenter, sous forme de littéraux en mode caractère, les caractères d'apostrophe et de barre de fraction inversée.

Si le caractère qui suit une barre de fraction inversée ne fait pas partie de ceux qui sont spécifiés, le comportement est indéterminé. Chaque séquence d'échappement ne spécifie qu'un seul caractère.

La séquence d'échappement \000 se compose d'une barre de fraction inversée, suivie d'un, deux ou trois chiffres octaux qui sont considérés comme spécifiant la valeur du caractère recherché. Une séquence de chiffres octaux ou hexadécimaux se termine par le premier caractère qui n'est pas, selon le cas, un chiffre octal ou un chiffre hexadécimal. La valeur d'une constante de caractère dépend de la mise en œuvre si elle dépasse la plus grande valeur de caractère.

Tableau 9 – Séquences d'échappement

Description	Séquence d'échappement
nouvelle ligne	\n
tabulation horizontale	\t
tabulation verticale	\v
retour arrière	\b
retour de chariot	\r
saut de page	\f
sonnette	\a
barre de fraction inversée	\\
point d'interrogation	\?
apostrophe	\'
guillemet droit	\"
nombre octal	\ooo
nombre hexadécimal	\xhh

4.1.5.3 Littéraux à virgule flottante

Un nombre à virgule flottante se compose d'un entier, d'une virgule décimale, d'une partie fractionnaire, d'un e ou d'un E et, facultativement, d'un exposant sous forme d'un entier signé. Les parties entière et fractionnaire se composent chacune d'une séquence de chiffres décimaux (en base dix). On peut omettre soit la partie entière ou la partie fractionnaire (mais pas les deux), soit la virgule décimale ou la lettre e (ou E) et l'exposant (mais pas les deux).

4.1.5.4 Littéraux à virgule fixe

Un littéral décimal à virgule fixe se compose d'un entier, d'une virgule décimale, d'une partie fractionnaire, d'un d ou d'un D. Les parties entière et fractionnaire se composent chacune d'une séquence de chiffres décimaux (en base dix). On peut omettre soit la partie entière ou la partie fractionnaire (mais pas les deux), soit la virgule décimale [mais pas la lettre d ou (D)].

4.1.5.5 Littéraux sous forme de chaîne

Un littéral-chaîne est une séquence de caractères (définie au 4.1.5.2) encadrée par des guillemets droits, comme dans "...".

Les littéraux de chaîne contigus sont concaténés. Les caractères contenus dans les chaînes concaténées sont tenus distincts. Par exemple, la chaîne

"\xA" "B" contient, après concaténation, les deux caractères '\xA' et 'B' (et non pas le seul caractère hexadécimal '\xAB').

La longueur d'un littéral-chaîne est le nombre de littéraux-caractères encadrés par les guillemets, après concaténation. La longueur du littéral est associée à celui-ci.

4.2 Prétraitement

On peut utiliser une notation de prétraitement pour écrire les modules, afin d'organiser les spécifications et de permettre de faire référence aux parties d'une spécification donnée. Les jetons d'inclusion de fichier source (#include) doivent toujours être interprétés comme constituant un moyen générique d'inclure un module de spécification donné, sans être liés à un quelconque système d'archivage particulier.

Le prétraitement en langage IDL ODP qui est spécifié dans la *norme ANSI/ISO C++* assure la substitution de macros, la compilation conditionnelle et l'inclusion de fichiers sources. En outre, des directives sont fournies afin de commander le numérotage des lignes dans les fichiers de diagnostic et pour le dépannage symbolique, afin de créer un message de diagnostic avec une séquence de jetons déterminée, et afin d'exécuter des actions dépendantes de la mise en œuvre (la directive # pragma). Certains noms prédéfinis sont disponibles. Ces capacités sont traitées logiquement par un préprocesseur, qui peut être concrètement mis en œuvre sous la forme d'un processus distinct ou non.

Les lignes qui commencent par un # (également appelées "directives"), communiquent avec ce préprocesseur. Une réserve peut apparaître avant le caractère #. Ces lignes ont une syntaxe indépendante du reste du langage IDL ODP; elles peuvent apparaître n'importe où et ont des effets durables (indépendamment des règles de visibilité du langage IDL ODP) jusqu'à la fin du module de traduction. L'emplacement dans le texte des jetons d'instruction de compilation (pragmas) propres au langage IDL ODP peut être soumis à des contraintes d'ordre sémantique.

Une directive de prétraitement (ou une ligne quelconque) peut se prolonger sur la ligne suivante d'un fichier source à condition de placer un caractère de barre de fraction inversée ("\") immédiatement avant le caractère de nouvelle ligne qui fait suite à la fin de la ligne qui doit être prolongée. Le préprocesseur effectue le prolongement en supprimant la barre de fraction inversée ainsi que le caractère de nouvelle ligne avant que la séquence d'entrée soit subdivisée en jetons. Un caractère de barre de fraction inversée peut ou non être le dernier caractère d'un fichier source.

Un jeton de prétraitement est: un jeton de langage IDL ODP (voir 4.1.1), un nom de fichier comme dans une directive de type `#include`, ou un caractère isolé quelconque, autre qu'une réserve, qui ne correspond à aucun autre jeton de prétraitement.

La fonction principale des capacités de prétraitement est de reprendre des définitions dans d'autres spécifications IDL ODP. Les textes contenus dans les fichiers repris par une directive `#include` sont traités comme s'ils apparaissaient dans le fichier d'accueil. Une description complète des capacités de prétraitement se trouve dans la *norme ANSI/ISO C++*.

4.3 Grammaire du langage IDL ODP

```
(1)   <specification>        ::=  <definition>+
(2)   <definition>           ::=  <type_dcl>";"
                              |   <const_dcl>";"
                              |   <except_dcl>";"
                              |   <interface>";"
                              |   <module>";"
(3)   <module>               ::=  "module"  <identifier> "{"  <definition>+ "}"
(4)   <interface>            ::=  <interface_dcl>
                              |   <forward_dcl>
(5)   <interface_dcl>        ::=  <interface_header> "{" <interface_body> "}"
(6)   <forward_dcl>          ::=  "interface" <identifier>
(7)   <interface_header>     ::=  "interface" <identifier> [<inheritance_spec> ]
(8)   <interface_body>       ::=  <export>*
(9)   <export>               ::=  <type_dcl> ";"
                              |   <const_dcl> ";"
                              |   <except_dcl> ";"
                              |   <attr_dcl> ";"
                              |   <op_dcl> ";"
(10)  <inheritance_spec>     ::=  ":" <scoped_name> { "," <scoped_name> }*
(11)  <scoped_name>          ::=  <identifier>
                              |   "::" <identifier>
                              |   <scoped_name> "::" <identifier>
(12)  <const_dcl>            ::=  "const" <const_type> <identifier> "=" <const_exp>
```

```
(13)    <const_type>           ::=  <integer_type>
                               |    <char_type>
                               |    <wide_char_type>
                               |    <boolean_type>
                               |    <floating_pt_type>
                               |    <string_type>
                               |    <wide_string_type>
                               |    <fixed_pt_const_type>
                               |    <scoped_name>
(14)    <const_exp>            ::=  <or_exp>
(15)    <or_exp>               ::=  <xor_expr>
                               |    <or_expr> "|" <xor_expr>
(16)    <xor_expr>             ::=  <and_expr>
                               |    <xor_expr> "^" <and_expr>
(17)    <and_expr>             ::=  <shift_expr>
                               |    <and_expr> "&" <shift_expr>
(18)    <shift_expr>           ::=  <add_expr>
                               |    <shift_expr> ">>" <add_expr>
                               |    <shift_expr> "<<" <add_expr>
(19)    <add_expr>             ::=  <mult_expr>
                               |    <add_expr> "+" <mult_expr>
                               |    <add_expr> "-" <mult_expr>
(20)    <mult_expr>            ::=  <unary_expr>
                               |    <mult_expr> "*" <unary-expr>
                               |    <mult_expr> "/" <unary_expr>
                               |    <mult_expr> "%" <unary_expr>
(21)    <unary_expr>           ::=  <unary_operator> <primary_expr>
                               |    <primary_expr>
(22)    <unary_operator>       ::=  "-"
                               |    "+"
                               |    "~"
(23)    <primary_expr>         ::=  <scoped_name>
                               |    <literal>
                               |    "(" <const_exp> ")"
(24)    <literal>              ::=  <integer_literal>
                               |    <string_literal>
                               |    <wide_string_literal>
                               |    <character_literal>
                               |    <wide_character_literal>
                               |    <fixed_pt_literal>
                               |    <floating_pt_literal>
                               |    <boolean_literal>
(25)    <boolean_literal>      ::=  "TRUE"
                               |    "FALSE"
(26)    <positive_int_const>   ::=  <const_exp>
```

```
(27)   <type_dcl>               ::=   "typedef" <type_declarator>
                                |     <struct_type>
                                |     <union_type>
                                |     <enum_type>
(28)   <type_declarator>        ::=   <type_spec> <declarators>
(29)   <type_spec>              ::=   <simple_type_spec>
                                |     <constr_type_spec>
(30)   <simple_type_spec>       ::=   <base_type_spec>
                                |     <template_type_spec>
                                |     <scoped_name>
(31)   <base_type_spec>         ::=   <floating_pt_type>
                                |     <integer_type>
                                |     <char_type>
                                |     <wide_char_type>
                                |     <boolean_type>
                                |     <octet_type>
                                |     <any_type>
                                |     <object_type>
(31a)  <object_type>            ::=   "Object"
(32)   <template_type_spec>     ::=   <sequence_type>
                                |     <string_type>
                                |     <wide_string_type>
                                |     <fixed_pt_type>
(33)   <constr_type_spec>       ::=   <struct_type>
                                |     <union_type>
                                |     <enum_type>
(34)   <declarators>            ::=   <declarator> { "," <declarator> } *
(35)   <declarator>             ::=   <simple_declarator>
                                |     <complex_declarator>
(36)   <simple_declarator>      ::=   <identifier>
(37)   <complex_declarator>     ::=   <array_declarator>
(38)   <floating_pt_type>       ::=   "float"
                                |     "double"
                                |     "long" "double"
(39)   <integer_type>           ::=   <signed_int>
                                |     <unsigned_int>
(40)   <signed_int>             ::=   <signed_long_int>
                                |     <signed_short_int>
                                |     <signed_longlong_int>
(40a)  <signed_longlong_int>    ::=   "long" "long"
(41)   <signed_long_int>        ::=   "long"
(42)   <signed_short_int>       ::=   "short"
(43)   <unsigned_int>           ::=   <unsigned_long_int>
                                |     <unsigned_short_int>
                                |     <unsigned_longlong_int>
```

```
(43a)  <unsigned_longlong_int> ::= "unsigned" "long" "long"

(44)   <unsigned_long_int>     ::=  "unsigned" "long"

(45)   <unsigned_short_int>    ::=  "unsigned" "short"

(46)   <char_type>             ::=  "char"

(46a)  <wide_char_type>        ::=  "wchar"

(47)   <boolean_type>          ::=  "boolean"

(48)   <octet_type>            ::=  "octet"

(49)   <any_type>              ::=  "any"

(50)   <struct_type>           ::=  "struct" <identifier> "{" <member_list> "}"

(51)   <member_list>           ::=  <member>+

(52)   <member>                ::=  <type_spec> <declarators> ";"

(53)   <union_type>            ::=  "union"<identifier> "switch"
                                         "(" <switch_type_spec> ")" "{" <switch_body> "}"

(54)   <switch_type_spec>      ::=  <integer_type>

                                |   <char_type>

                                |   <boolean_type>

                                |   <enum_type>

                                |   <scoped_name>

(55)   <switch_body>           ::=  <case>+

(56)   <case>                  ::=  <case_label>+ <element_spec> ";"

(57)   <case_label>            ::=  "case" <const_exp> ":"

                                |   "default" ":"

(58)   <element_spec>          ::=  <type_spec> <declarator>

(59)   <enum_type>             ::=  "enum" <identifier> "{" <enumerator>
                                         {"," <enumerator> } * "}"

(60    <enumerator>            ::=  <identifier>

(61)   <sequence_type>         ::=  "sequence" "<" <simple_type_spec> ","
                                         <positive_int_const> ">"

                                |   "sequence" "<" <simple_type_spec> ">"

(62)   <string_type>           ::=  "string" "<" <positive_int_const> ">

                                |   "string"

(62a)  <wide_string_type>      ::=  "wstring" "<" <positive_int_const>
                                         ">" | "wstring"

(63)   <array_declarator>      ::=  <identifier> <fixed_array_size>+

(64)   <fixed_array_size>      ::=  "[" <positive_int_const> "]"

(65)   <attr_dcl>              ::=  [ "readonly" ] "attribute"<param_type_spec>
                                         <simple_declarator>
                                         { "," <simple_declarator> }*

(66)   <except_dcl>            ::=  "exception" <identifier> "{" <member>*"}"

(67)   <op_dcl>                ::=  [ <op_attribute> ] <op_type_spec>
                                         <identifier> <parameter_dcls>
                                         [ <raises_expr> ] [ <context_expr> ]

(68)   <op_attribute>          ::=  "oneway"

(69)   <op_type_spec>          ::=  <param_type_spec>

                                |   "void"

(70)   <parameter_dcls>        ::=  "(" <param_dcl> { "," <param_dcl> }* ")"

                                |   "(" ")"

(71)   <param_dcl>             ::=  <param_attribute> <param_type_spec>
                                         <simple_declarator>
```

```
(72)    <param_attribute>        ::=   "in"
                                  |    "out"
                                  |    "inout"

(73)    <raises_expr>            ::=   "raises" "(" <scoped_name> { ","<scoped_name> }* ")"

(74)    <context_expr>           ::=   "context "(" <string_literal>
                                            { ","<string_literal> } * ")"

(75)    <param_type_spec>        ::=   <base_type_spec>
                                  |    <string_type>
                                  |    <fixed_pt_type>
                                  |    <wide_string_type>
                                  |    <scoped_name>

(76)    <fixed_pt_type>          ::=   "fixed" "<" <positive_int_const>
                                            ","<integer_literal> ">"

(77)    <fixed_pt_const_type>    ::=   "fixed"
```

4.4 Spécification IDL ODP

Une spécification IDL ODP se compose d'une ou de plusieurs définitions de type, de constante, d'exception ou de module. La syntaxe est la suivante:

```
        <specification>::=       <defintion> +

        <definition>             ::=   <type_dcl> ";"
                                  |    <const_dcl> ";"
                                  |    <except_dcl> ";"
                                  |    <interface_dcl> ";"
                                  |    <module_dcl> ";"
```

Voir 4.6, 4.7 et 4.9, respectivement, pour les spécifications des structures <const_dcl>, <type_dcl>, et <except_dcl>.

4.4.1 Déclaration de module

Une définition de module suit la syntaxe suivante:

```
        <module>                 ::=   "module" <identifier> "{ <definition>+"}"
```

La structure modulaire sert à détecter des identificateurs IDL ODP; voir 4.12 pour plus de détails.

4.4.2 Déclaration d'interface

Une déclaration d'interface suit la syntaxe suivante:

```
        <interface>              ::=   <interface_dcl>
                                  |    <forward_dcl>

        <interface_dcl>          ::=   <interface_header> "{" <interface_body> "}"

        <forward_dcl>            ::=   "interface" <identifier>

        <interface_header>       ::=   "interface: <identifier> [ <inheritance_spec> ]

        <interface_body>         ::=   <export>*

        <export>                 ::=   <type_dcl> ";"
                                  |    <const_dcl> ";"
                                  |    <except_dcl> ";"
                                  |    <attr_dcl>";"
                                  |    <op_dcl>";"
```

4.4.2.1 En-tête d'interface

L'en-tête d'interface se compose de deux éléments:

- Le nom de l'interface – Ce nom doit être précédé du mot clé `interface` et se compose d'un identificateur qui désigne l'interface.
- Une spécification facultative d'héritage. La spécification d'héritage est décrite au 4.4.2.2.

Le non-terminal `<identifier>` qui désigne une interface définit un nom de type autorisé. Un tel nom de type peut être utilisé chaque fois qu'un non-terminal `<identifier>` est autorisé par la grammaire, sous réserve des contraintes sémantiques décrites dans les paragraphes suivants. Le nom `Object` est un nom valide qui permet de transmettre toute référence d'interface. Etant donné que l'on ne peut conserver que des références à une interface, la signification d'un paramètre ou d'un élément structurel qui est un type d'interface sert de *référence* à une instance de ce type d'interface. Chaque lien linguistique décrit la façon dont le programmeur doit représenter de telles références d'interface. On peut en particulier utiliser ce lien comme paramètre dans une description d'opération, ce qui permet de transmettre des références d'interface sous la forme de paramètres.

4.4.2.2 Spécification d'héritage

La syntaxe d'héritage est la suivante:

```
<inheritance_spec>      ::=  ":" <scoped_name> { "," <scoped_name> }*

<scoped_name>           ::=  <identifier>

                         |     "::" <identifier>

                         |     <scoped_name>"::" <identifier>
```

Chaque nom `<scoped_name>` contenu dans une spécification d'héritage `<inheritance_spec>` doit désigner une interface préalablement définie. Voir 4.5 pour la description de l'héritage.

4.4.2.3 Corps d'interface

Le corps d'interface contient les sortes de déclarations suivantes:

- les déclarations de constantes, qui spécifient les constantes exportées par l'interface; la syntaxe de déclaration des constantes est décrite au 4.6;
- les déclarations de types, qui spécifient les définitions de type exportées par l'interface; la syntaxe de déclaration des types est décrite au 4.7;
- les déclarations d'exceptions, qui spécifient les structures d'exception exportées par l'interface; la syntaxe de déclaration des exceptions est décrite au 4.9;
- les déclarations d'attributs, qui spécifient les attributs associés qui sont exportés par l'interface; la syntaxe de déclaration des attributs est décrite au 4.11;
- les déclarations d'opérations qui spécifient les opérations exportées par l'interface et le format de chaque opération, y compris son nom, le type de données renvoyées, les types de tous les paramètres d'une opération, les exceptions autorisées qui peuvent être renvoyées à la suite d'une invocation, et des informations sur le contexte qui peuvent affecter la méthode de transmission; la syntaxe de déclaration des opérations est décrite au 4.10.

Les interfaces vides (c'est-à-dire celles qui ne contiennent aucune déclaration) sont autorisées.

4.4.2.4 Déclaration anticipée

Une déclaration anticipée annonce le nom d'une interface sans définir celle-ci. Cela permet la définition d'interfaces qui se font mutuellement référence. La syntaxe consiste simplement à faire suivre le mot clé `interface` d'un identificateur `<identifier>` qui désigne l'interface. La définition proprement dite doit suivre ultérieurement dans la spécification.

De multiples déclarations anticipées du même nom d'interface sont autorisées.

4.5 Héritage

Une interface peut être dérivée d'une autre interface qui est alors appelée interface de *base* de l'interface dérivée. Celle-ci, comme toutes les interfaces, peut déclarer de nouveaux éléments (constantes, types, attributs, exceptions et opérations). En outre, les éléments d'une interface de base, s'ils n'ont pas été redéfinis dans l'interface dérivée, peuvent faire l'objet d'une référence comme s'ils étaient des éléments de l'interface dérivée. L'opérateur de résolution du nom ("::") peut être utilisé pour se référer explicitement à un élément de base; cela permet de faire référence à un nom qui a été redéfini dans l'interface dérivée.

Une interface dérivée peut redéfinir un nom quelconque de type, de constante ou d'exception qui a été hérité; les règles de visibilité de tels noms sont décrites au 4.12.

Une interface est dite directe si elle est mentionnée dans la spécification d'héritage `<inheritance_spec>`. Elle est dite indirecte si elle n'est pas directe mais sert de base à l'une des interfaces mentionnées dans la spécification `<inheritance_spec>`.

Une interface peut être dérivée d'un nombre quelconque d'interfaces de base. Cette utilisation de plusieurs interfaces de base directes est souvent appelée *héritage multiple*. L'ordre de déduction n'est pas significatif.

Une interface ne peut pas être spécifiée plus d'une seule fois comme interface de base directe d'une interface dérivée; mais elle peut être spécifiée plus d'une fois comme interface de base indirecte. Soit l'exemple suivant:

```
interface A {...}

interface B:A {...}

interface C:A {...}

interface D:B,C {...}
```

Les relations entre ces interfaces sont représentées sur la Figure 1. Cette forme "en diamant" est autorisée.

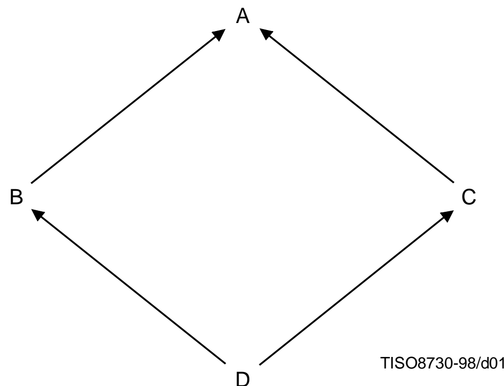

Figure 1 – Exemple d'héritage multiple autorisé

La référence aux éléments de l'interface de base ne doit jamais être ambiguë. La référence à un élément de l'interface de base est ambiguë si l'expression utilisée se rapporte à une constante, à un type ou à une exception dans plusieurs interfaces de base. (Il est présentement interdit d'hériter de deux interfaces avec le même nom d'opération ou d'attribut, ou de redéfinir un nom d'opération ou d'attribut dans l'interface dérivée.) Les ambiguïtés peuvent être résolues par association du nom à celui de son interface (c'est-à-dire au moyen d'un nom inscrit dans un domaine de visibilité `<scoped name>`).

Les références aux constantes, aux types et aux exceptions sont liées à une interface lorsque celle-ci est définie, c'est-à-dire que ces références sont remplacées par les noms universels équivalents de type `<scoped name>`. Ce remplacement garantit que la syntaxe et la sémantique d'une interface ne seront pas modifiées lorsque cette interface sera la base d'une interface dérivée. Soit l'exemple suivant:

```
const long L=3;

interface A {

        typedef float coord[L]);

      void f (in coord s); // s possède 3 nombres à virgule flottante en précision
      simple };
```

```
interface B{
      const long L=4;
};
interface C: B, A {}                 //quelle est la signature de f()?
```

Le rattachement préalable des constantes, des types et des exceptions à la définition d'une interface garantit que la signature de l'opération `f` dans l'interface `C` est la suivante:

```
typedef float coord[3];
void f (in coord s);
```

ce qui correspond à la signature contenue dans l'interface `A`. Cette règle empêche également que la redéfinition d'une constante, d'un type ou d'une exception dans l'interface dérivée affecte les opérations et les attributs hérités d'une interface de base.

L'héritage d'interface provoque l'importation, dans le domaine actuel de visibilité des noms, de tous les identificateurs contenus dans l'enveloppe de l'arbre d'héritage. Un nom de type, de constante, de valeur d'énumération ou d'exception, extrait d'un domaine de visibilité englobant, peut être redéfini dans le domaine de visibilité actuel.

Toutes les opérations qui peuvent s'appliquer à un objet particulier doivent avoir des noms uniques. Cette prescription interdit la redéfinition d'un nom d'opération dans une interface dérivée, ainsi que l'héritage de deux opérations portant le même nom.

NOTE – On prévoit que de futures révisions du langage IDL ODP pourront alléger cette règle d'une certaine manière, par exemple en autorisant la surcharge ou en offrant certains moyens pour opérer une distinction entre opérations homonymes.

4.6 Déclaration de constante

4.6.1 Syntaxe

Le présent paragraphe décrit la syntaxe des déclarations de constantes.

La syntaxe de déclaration des constantes est la suivante:

```
<const_dcl>          ::=  "const" <const_type> <identifier> "=" <const_exp>
<const_type>         ::=  <integer_type>
                     |    <char_type>
                     |    <wide_char_type>
                     |    <boolean_type>
                     |    <floating_pt_type>
                     |    <string_type>
                     |    <wide_string_type>
                     |    <fixed_pt_const_type>
                     |    <scoped_name>
<const_exp>          ::=  <or_exp>
<or_exp>             ::=  <xor_expr>
                     |    <or_expr> "|" <xor_expr>
<xor_expr>           ::=  <and_expr>
                     |    <xor_expr> "^" <and_expr>
<and_expr>           ::=  <shift_expr>
                     |    <and_expr> "&" <shift_expr>
<shift_expr>         ::=  <add_expr>
                     |    <shift_expr> ">>" <add_expr>
                     |    <shift_expr> "<<" <add_expr>
<add_expr>           ::=  <mult_expr>
                     |    <add_expr> "+" <mult_expr>
                     |    <add_expr> "-" <mult_expr>
```

```
<mult_expr>               ::=  <unary_expr>
                           |    <mult_expr> "*" <unary-expr>
                           |    <mult_expr> "/" <unary_expr>
                           |    <mult_expr> "%" <unrary_expr>
<unary_expr>              ::=  <unary_operator> <primary_expr>
                           |    <primary_expr>
<unary_operator>          ::=  "-"
                           |    "+"
                           |    "~"
<primary_expr>            ::=  <scoped_name>
                           |    <literal>
                           |    "(" <const_exp> ")"
<literal>                 ::=  <integer_literal>
                           |    <string_literal>
                           |    <character_literal>
                           |    <wide_character_literal>
                           |    <fixed_pt_literal>
                           |    <floating_pt_literal>
                           |    <boolean_literal>
<boolean_literal>         ::=  "TRUE"
                           |    "FALSE"
<positive_int_const>::= <const_exp>
```

4.6.2 Sémantique

Le nom <scoped_name>, visible dans la production <const_type>, doit être le nom préalablement défini d'une constante de type <integer_type>, <char_type>, <wide_char_type>, <boolean_type>, <floating_pt_type>, <fixed_pt_const_type>, <string_type> ou <wide_string_type>.

Un opérateur infixe peut combiner deux entiers, deux nombres à virgule flottante ou deux nombres à virgule fixe, mais aucun mélange de ceux-ci. Les opérateurs infixes sont seulement applicables à des types entiers, des nombres à virgule flottante et des nombres à virgule fixe.

Une expression de constante de type entier est assimilée à un type "long-double" non signé, à moins qu'elle ne contienne un littéral-entier inversé ou une constante-entier de valeur négative ou d'une autre sous-expression de valeur négative, auquel cas elle est évaluée comme type long-double signé. La valeur calculée du long-double est forcée à revenir au type "cible" dans les initialisateurs de constantes. Un état d'erreur est détecté si la valeur ainsi calculée dépasse l'étendue affectée au type cible ou si une valeur intermédiaire dépasse l'étendue du type évalué comme étant de type (long-double ou long-double non signé).

Tous les littéraux de type "nombre à virgule flottante" sont assimilés au type long-double; toutes les constantes numériques à virgule flottante sont forcées au type long-double et toutes les expressions à virgule flottante sont calculées comme des longs-doubles. La valeur calculée d'un long-double est forcée à revenir au type "cible" dans les initialisateurs de constantes. Un état d'erreur est détecté si ce forçage échoue ou si certaines valeurs intermédiaires dépassent (lors de l'évaluation de l'expression) l'étendue affectée au type long-double.

Les opérateurs unaires (+) et binaires (* / + –) sont applicables aux expressions à virgule flottante. Les opérateurs unaires (+ – ~) et binaires (* / % + – << >> & | ^) sont applicables aux expressions d'entiers.

L'opérateur unaire "~" indique qu'il faut créer le complément binaire de l'expression à laquelle cet opérateur s'applique. Dans le cadre de telles expressions, les valeurs sont des compléments à 2. Un tel complément peut être créé comme suit:

`long long` –(valeur+1)

`unsigned long long` (2**64–1) – valeur

L'opérateur binaire "%" fournit le reste de la division de la première expression par la seconde. Si le deuxième opérande de l'opérateur "%" est 0, le résultat est indéfini; sinon l'expression:

```
(a/b)*b +a%b
```

est égale à a. Si les deux opérandes ne sont pas négatifs, le reste n'est pas négatif; s'ils le sont, le signe du reste dépend de l'application.

L'opérateur binaire "<<" indique que la valeur de l'opérande de gauche doit être déplacée à gauche du nombre de bits spécifiés par l'opérande de droite, avec bourrage à zéro des positions binaires vides. L'opérande de droite doit toujours être compris dans l'étendue 0 ≤ opérande de droite < 32.

L'opérateur binaire "&" indique qu'il faut créer l'opérateur logique AND au niveau des bits des opérandes de gauche et de droite.

L'opérateur binaire "|" indique qu'il faut créer l'opérateur logique OR au niveau des bits des opérandes de gauche et de droite.

L'opérateur "^" indique qu'il faut créer l'opérateur logique EXCLUSIVE-OR au niveau des bits des opérandes de gauche et de droite.

L'expression <positive_int_const> doit être assimilée à une constante sous forme d'entier positif.

Les expressions de constante décimale à virgule fixe sont évaluées comme suit. Un littéral à virgule fixe possède le nombre apparent de chiffres total et fractionnel, à l'exception des zéros d'en-tête et de queue qui sont factorisés, comprenant des zéros non significatifs avant la virgule décimale. Par exemple, 0123.450d est considéré comme fixed<5,2> et 3000.00 comme fixed<1, -3>. Les opérateurs de préfixe n'attribuent pas la précision, un préfixe + est optionnel, et ne modifient pas le résultat. Les limites supérieures du nombre de chiffres et l'échelle du résultat d'une expression infixe, fixed<d1, s1> op fixed<2d, s2>, sont indiquées dans le tableau suivant:

Op	Result: fixed<d,s>
+	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
–	fixed<max(d1-s1,d2-s2) + max(s1,s2) + 1, max(s1,s2)>
*	fixed<d1+d2, s1+s2>
/	fixed<(d1-s1+s2) + s inf , s inf>

Un quotient peut avoir un nombre arbitraire de positions décimales, indiqué par une échelle de s inf. Le calcul s'effectue par paire, avec les règles usuelles d'association de gauche à droite, de priorité de l'opérateur et des parenthèses. Si un calcul individuel entre une paire de littéraux à virgule fixe génère effectivement plus de 31 chiffres significatifs, un résultat de 31 chiffres est alors conservé de la façon suivante:

```
fixed<d,s> => fixed<31, 31-d+s>
```

Les zéros d'en-tête et de queue ne sont pas considérés comme significatifs. Les chiffres oubliés sont abandonnés; l'arrondi n'est pas effectué. Le résultat du calcul individuel devient alors un opérande littéral de la prochaine paire de littéraux à virgule fixe à traiter. Les opérateurs unaires (+ -) et binaires (* / + -) sont applicables aux expressions à virgule flottante et à virgule fixe. Les opérateurs unaires (+ – ~) et binaires (* / % + – << >> & | ^) sont applicables aux expressions d'entiers.

4.7 Déclaration d'un type

Le langage IDL ODP fournit des structures syntaxiques pour nommer les types de données; c'est-à-dire qu'il offre des déclarations en langage de type C qui associent un identificateur à un type. Le langage IDL ODP fait appel au mot clé typedef afin d'associer un nom à un type de données; un nom est également associé à un type de données au moyen des déclarations struct, union, et enum; la syntaxe de déclaration est la suivante:

```
<type_dcl>              ::=  "typedef" <type_declarator>
                        |    <struct_type>
                        |    <union_type>
                        |    <enum_type
<type_declarator>       ::= <type_spec> <declarators>
```

Pour les déclarations de type, le langage IDL ODP définit un ensemble de spécificateurs de type indiquant des types de valeurs. La syntaxe de ces déclarations est la suivante:

```
<type_spec>             ::=  <simple_type_spec>
                        |    <constr_type_spec>
<simple_type_spec>      ::=  <base_type_spec>
                        |    <template_type_spec>
                        |    <scoped_name>
<base_type_spec>        ::=  <floating_pt_type>
                        |    <integer_type>
                        |    <char_type>
                        |    <wide_char_type>
                        |    <boolean_type>
                        |    <octet_type>
                        |    <any_type>
                        |    <object_type>
<template_type_spec>    ::=  <sequence_type>
                        |    <string_type>
<constr_type_spec>      ::=  <struct_type>
                        |    <union_type>
                        |    <enum_type>
<declarators>           ::=  <declarator> { "," <declarator> } *
<declarator>            ::=  <simple_declarator>
                        |    <complex_declarator>
<simple_declarator>     ::=  <identifier>
<complex_declarator>    ::=  <array_declarator>
```

Comme indiqué ci-dessus, les spécificateurs de type se composent, en langage IDL ODP, de types de données scalaires et de constructeurs de type. Les spécificateurs de type IDL ODP peuvent être utilisés dans des déclarations d'opération afin d'assigner des types de données aux paramètres de ces opérations.

4.7.1 Types de base

La syntaxe pour les types de base pris en charge est la suivante:

```
<floating_pt_type>      ::=  "float"
                        |    "double"
                        |    "long" "double"
<integer_type>          ::=  <signed_int>
                        |    <unsigned_int>
<signed_int>            ::=  <signed_long_int>
                        |    <signed_short_int>
                        |    <signed_longlong_int>
<signed_long_int>       ::=  "long"
<signed_short_int>      ::=  "short"
<signed_longlong_int>   ::=  "long" "long"
<unsigned_int>          ::=  <unsigned_long_int>
                        |    <unsigned_short_int>
                        |    <unsigned_longlong_int>
<unsigned_long_int>     ::=  "unsigned" "long"
```

```
<unsigned_short_int>     ::=  "unsigned" "short"

<unsigned_longlong_int>  ::=  "unsigned" "long" "long"

<char_type>              ::=  "char"

<wide_char_type>         ::=  "wchar"

<boolean_type>           ::=  "boolean"

<octet_type>             ::=  "octet"

<any_type>               ::=  "any"

<object_type>            ::=  "Object "
```

Chaque type de données IDL ODP est mappé sur un type de données natif au moyen du mappage de traduction approprié. Des erreurs de conversion entre les types de données IDL ODP et les types natifs auxquels ils sont mappés peuvent se produire au cours de l'exécution d'une invocation d'opération.

4.7.1.1 Types d’entiers

Le langage IDL ODP mappe les types de données d'entiers de précision étendue signés et non signés (`long`, `short`, `long long` et `unsigned`). Le type `long` représente l'étendue -2^{31} .. 2^{31} -1, le type `unsigned long` représente l'étendue 0 .. 2^{32} -1, le type `short` représente l'étendue -2^{15} .. 2^{15} -1, le type `unsigned short` représente l'étendue 0 .. 2^{16} -1, le type `long long` représente l'étendue -2^{63} -1, tandis que le type `unsigned long long` représente des valeurs dans l'étendue 0 .. 2^{64} -1

4.7.1.2 Type à virgule fixe

Le type de données `fixed` représente un nombre décimal à virgule fixe ayant jusqu'à 31 chiffres significatifs. Le facteur d'échelle est normalement un entier non négatif inférieur ou égal au nombre total de chiffres (il convient de noter que les constantes ayant une échelle effectivement négative, telle que 10 000, sont toujours autorisées). Cependant, certains langages et certains environnements peuvent prendre en compte des types ayant une échelle négative ou une échelle plus grande que le nombre de digits.

4.7.1.3 Types à virgule flottante

Les types IDL ODP à virgule flottante sont les types `float`, `double` et `long double`. Le type `float` représente des nombres à virgule flottante en précision simple selon l'IEEE. Le type `double` représente des nombres à virgule flottante en précision double selon l'IEEE; le type `long double` représente un nombre à virgule flottante selon l'IEEE avec double extension, qui peut avoir un exposant d'au moins 15 bits de longueur et une fraction signée d'au moins 64 bits. La norme de l'IEEE concernant la virgule flottante (*IEEE Standard for Binary Floating-Point Arithmetic,* ANSI/IEEE Std 754-1985) doit être consultée pour plus d'informations concernant la précision offerte par ces types.

Les mises en œuvre qui ne prennent pas totalement en charge l'ensemble des valeurs de la norme IEEE 754 concernant la virgule flottante doivent spécifier complètement leurs divergences par rapport à cette norme.

4.7.1.4 Type caractère

Le langage IDL ODP définit un type de données `char` qui est une grandeur de 8 bits qui:

1) code un caractère d'un octet simple de tout jeu de codes orienté octet;

2) lorsqu'il est utilisé dans un tableau, code un caractère de plusieurs octets d'un jeu de codes multi-octet.

Autrement dit, une réalisation est libre d'utiliser en interne tout jeu de codes pour coder des données de type caractère, bien que la conversion vers une autre forme puisse être requise pour la transmission.

Par défaut, le jeu de caractères ISO/CEI Latin-1 (ISO/CEI 8859-1) définit la signification et la représentation de tous les caractères graphiques possibles (c'est-à-dire les caractères alphabétiques, numériques, graphiques et d'espace qui sont définis dans les Tableaux 2, 3 et 4). La signification et la représentation des caractères nuls et de formatage (voir le Tableau 5) sont la valeur numérique du caractère correspondant comme défini dans la norme ASCII (ISO/CEI 646). La signification de tous les autres caractères dépend de la mise en œuvre.

Au cours de la transmission, les caractères peuvent être convertis en d'autres formes appropriées, selon ce qui est requis par un lien linguistique particulier. De telles conversions peuvent modifier la représentation d'un caractère tout en conservant son sens. Par exemple, un caractère peut être converti à partir et à destination de certains jeux de caractères internationaux.

4.7.1.5 Type caractère large

Le langage IDL ODP définit un type de données `wchar` qui code des caractères larges, extraits d'un jeu de caractères quelconque. Comme pour les données de type caractère, une mise en œuvre a la possibilité d'utiliser, en interne, n'importe quel jeu d'éléments codés afin de représenter des caractères larges; une conversion vers une autre forme peut cependant être requise pour la transmission. La dimension des caractères de type `wchar` dépend de la mise en œuvre.

4.7.1.6 Type booléen

Le type booléen (`boolean`) est utilisé pour indiquer un item de données qui ne peut prendre qu'une seule des deux valeurs suivantes: TRUE ou FALSE.

4.7.1.7 Type "octet"

Le type `octet` est une grandeur de huit éléments binaires qui est assurée de ne pas subir de conversion au cours de sa transmission par le système de communication.

4.7.1.8 Type "any"

Le type `any` permet de spécifier des valeurs qui peuvent exprimer un type quelconque de langage IDL ODP.

4.7.2 Types "constructed" (structurés)

Les types structurés sont les suivants:

```
<constr_type_spec>      ::= <struct_type>
                        |       <union_type>
                        |       <enum_type>
```

Bien que la syntaxe IDL ODP permette la génération de spécifications de type structuré récurrent, la seule récurrence permise pour les types structurés l'est au moyen du modèle séquentiel de type `sequence`.

Par exemple, la spécification suivante est correcte:

```
struct foo {
        long value;
        sequence <foo> chain;
}
```

On trouvera au 4.7.3.1 des détails sur le modèle séquentiel de type `sequence`.

4.7.2.1 Structures

La syntaxe de la structure est la suivante:

```
<struct_type>           ::=  "struct" <identifier> "{" <member_list> "}"
<member_list>           ::=  <member>+
<member>                ::=  <type_spec> <declarators> ";"
```

Le terme `<identifier>` contenu dans le type `<struct_type>` définit un nouveau type autorisé. Les types de structures peuvent également être nommés au moyen d'une déclaration de type `typedef`.

Les règles de détection de nom prescrivent que les déclarateurs de membre, contenus dans une structure particulière, soient uniques. La valeur d'une structure `struct` est déterminée par celle de tous ses membres.

4.7.2.2 Réunion d'ensembles discriminés

La syntaxe d'une réunion mot clé `union` d'ensembles discriminés est la suivante:

```
<union_type>            ::=  "union"<identifier> "switch"
                                   "(" <switch_type_spec>"")" "{" <switch_body> "}"
<switch_type_spec>      ::=  <integer_type>
                        |    <char_type>
                        |    <boolean_type>
                        |    <enum_type>
                        |    <scoped_name>
```

```
<switch_body>           ::=  <case>+

<case>                  ::=  <case_label>+ <element_spec> ";"

<case_label>            ::=  "case" <const_exp> ":"
                        |    "default" ":"

<element_spec>          ::=  <type_spec> <declarator>
```

En langage IDL ODP, les réunions sont au croisement des déclarations de réunion (`union`) d'objets C et des déclarations de corps commutatif à valeur par défaut ou à constante de message `switch`. Les réunions IDL ODP doivent toujours porter sur des ensembles discriminés; c'est-à-dire que l'en-tête de la réunion doit toujours comporter un champ d'étiquette de type qui détermine le membre de la réunion qu'il convient d'utiliser pour l'instance actuelle d'une communication. L'identificateur `<identifier>` qui suit le mot clé `union` définit un nouveau type autorisé. Les types de réunion peuvent également être nommés au moyen d'une déclaration de définition de type `typedef`. Le non-terminal `<const_exp>` contenu dans une étiquette `<case_label>` doit toujours être compatible avec la spécification `<switch_type_spec>`. Un cas de valeur par défaut (`default`) ne peut apparaître qu'une seule fois, au plus. Le nom détecté `<scoped_name>` contenu dans la production `<switch_type_spec>` doit toujours être d'un des types prédéfinis suivants: `integer`, `char`, `boolean` ou `enum`.

Les étiquettes de cas doivent toujours correspondre au type défini du discriminateur ou lui être automatiquement attribuables. Le Tableau 10 montre l'ensemble complet des règles de correspondance.

Les règles de détection de nom prescrivent que les déclarateurs d'élément doivent être uniques dans une réunion donnée. Si la spécification `<switch_type_spec>` est du type des énumérés `<enum_type>`, l'identificateur de cette énumération se trouve dans le domaine de visibilité de la réunion; celle-ci doit donc être distincte des déclarateurs d'élément.

Il n'est pas prescrit que toutes les valeurs possibles du discriminateur de réunion soient énumérées dans le corps commutatif `<switch_body>`. La valeur d'une réunion est celle de son discriminateur, assortie de l'une des valeurs suivantes:

- si la valeur du discriminateur a été explicitement énumérée dans une déclaration de `cas`, la valeur de l'élément associé à cette déclaration de `cas`;
- si une étiquette de `cas` a été spécifiée, la valeur de l'élément associé à l'étiquette de `cas` par défaut;
- aucune valeur additionnelle.

L'accès au discriminateur et aux éléments associés dépend du mappage linguistique.

Tableau 10 – Mise en correspondance des étiquettes de cas

Discriminateur type	Mis en correspondance avec
long	toute valeur d'entier dans l'étendue des valeurs du type long
long long	toute valeur d'entier dans l'étendue du type long long
short	toute valeur d'entier dans l'étendue des valeurs du type court
unsigned long	toute valeur d'entier dans l'étendue des valeurs du type long non signé
unsigned long long	toute valeur d'entier dans l'étendue du type long long non signé
unsigned short	toute valeur d'entier dans l'étendue des valeurs du type court non signé
char	caractère
wchar	caractère large
boolean	TRUE ou FALSE
enum	tout énumérateur pouvant être énuméré par le type "énumération de discriminateurs"

4.7.2.3 Enumérations

Les types d'énumérés se composent de listes ordonnées d'identificateurs, dont la syntaxe est la suivante:

```
<enum_type>          ::=  "enum"<identifier>"{" <enumerator> { ","<enumerator> }*"}"

<enumerator>         ::=  <identifier>
```

Un maximum de 2^{32} identificateurs peut être spécifié dans une énumération donnée; en tant que tels, les noms des objets énumérés doivent toujours être mappés sur un type de données natif, capable de représenter une énumération de longueur maximale. L'ordre dans lequel les identificateurs sont désignés dans la spécification d'une énumération définit l'ordre relatif des identificateurs. Tout mappage linguistique qui permet à deux énumérateurs d'être comparés, ou qui définit des fonctions de successeur/prédécesseur pour des énumérateurs, doit être conforme à cette relation d'ordre. L'identificateur `<identifier>` qui suit le mot clé `enum` définit un nouveau type autorisé. Les types d'énumérés peuvent également être nommés au moyen d'une déclaration de définition de type (`typedef`).

4.7.3 Types de modèles

Les types de modèles sont les suivants:

```
<template_type_spec>     ::=  <sequence_type>
                          |   <string_type>
                          |   <wide_string_type>
                          |   <fixed_pt_type>
```

4.7.3.1 Séquences

Le langage IDL ODP définit le type "séquence" par le mot clé `sequence`. Une séquence est une table unidimensionnelle qui possède deux caractéristiques: une longueur maximale (fixée au moment de la compilation) et une longueur réelle (qui est déterminée au moment de l'exécution).

La syntaxe des séquences est la suivante:

```
<sequence_type>          ::=  "sequence" "<" <simple_type_spec>
                                    ","<positive_int_const> ">"

                          |   "sequence" "<" <simple_type_spec> ">"
```

Le deuxième paramètre d'une déclaration de séquence indique la longueur maximale de cette séquence. Si une constante à entier positif est spécifiée pour la longueur maximale, la séquence est dite "bornée". Avant de transmettre une séquence bornée sous forme d'argument de fonction (ou sous forme de champ dans une structure ou dans une réunion), la longueur de cette séquence doit être déterminée par mappage linguistique. Après réception du résultat d'une séquence exécutée à la suite d'une invocation d'opération, la longueur réelle de la séquence retournée sera déterminée; cette valeur peut être obtenue par mappage linguistique.

Si aucune longueur maximale n'est spécifiée, la longueur de la séquence n'est pas spécifiée (séquence non bornée). Avant de transmettre une telle séquence sous forme d'argument de fonction (ou sous forme de champ dans une structure ou dans une réunion), il faut déterminer, par mappage linguistique, la longueur réelle de la séquence, sa longueur maximale et l'adresse du tampon qui la contiendra. Après réception du résultat d'une séquence exécutée à la suite d'une invocation d'opération, la longueur réelle de la séquence retournée sera déterminée; cette valeur peut être obtenue par mappage linguistique.

Un type de séquence peut être utilisé comme paramètre de type pour un autre type de séquence. Par exemple, l'expression suivante:

```
typedef sequence <sequence <long> > Fred;
```

déclare que "Fred" est du type "séquence non bornée d'une séquence non bornée de nombres longs". On notera que, pour déclarer des séquences imbriquées, il faut utiliser des espaces vides afin de séparer les deux jetons ">" qui terminent la déclaration, afin qu'ils ne soient pas analysés comme représentant un unique jeton ">>".

4.7.3.2 Chaînes

Le langage IDL ODP définit le type "chaîne" par le mot clé `string`, qui se compose de toutes les grandeurs de 8 bits possibles, sauf la valeur nulle. Une chaîne est analogue à une séquence de caractères. Comme dans le cas des séquences du type "any", la longueur réelle d'une chaîne doit toujours être déterminée par mappage linguistique avant de transmettre cette chaîne sous la forme d'un argument de fonction (ou de l'intégrer à une structure ou à une réunion). La syntaxe de déclaration des chaînes est la suivante:

```
<string_type>            ::=  "string" "<" <positive_int_const> ">"

                          |   "string"
```

L'argument applicable à la déclaration de chaîne est la longueur maximale de la chaîne. Si cette longueur maximale est spécifiée sous la forme d'un entier positif, la chaîne est dite "bornée"; si aucune longueur maximale n'est spécifiée, la chaîne est dite "non bornée".

Les chaînes sont analysées comme étant d'un type particulier parce que de nombreux langages possèdent des fonctions internes spécialisées ou des fonctions de bibliothèque de liens pour la manipulation des chaînes. Un type de chaîne particulier peut autoriser une importante optimisation dans le traitement des chaînes, par rapport à ce qui peut être accompli au moyen de séquences de type général.

4.7.3.3 Chaînes de caractères larges

Le type de données `wstring` représente une séquence de caractères larges (`wchar`) terminée par un caractère nul (note: un caractère large nul). Le type `wstring` est analogue au type `string`, sauf que son type d'élément est `wchar` au lieu de `char`.

```
<wide_string_type>       ::=  "wstring" "<" <positive_int_const> ">"
                          |   "wstring"
```

4.7.4 Déclarateur complexe

4.7.4.1 Tables de valeurs

Le langage IDL ODP définit des tables à plusieurs dimensions fixes. Chaque dimension de table possède une étendue de valeurs explicites.

La syntaxe des tables de valeurs est la suivante:

```
<array_declarator>       ::= <identifier> <fixed_array_size>+

<fixed_array_size>       ::= "[" <positive_int_const> "]"
```

Les étendues de valeurs (dans chaque dimension) sont fixées au moment de la compilation. Lorsqu'une table est transmise sous la forme d'un paramètre pour une invocation d'opération, tous les éléments de cette table sont transmis.

La mise en œuvre des indices de table est propre au mode de mappage utilisé; la transmission d'un indice de table sous la forme d'un paramètre peut donner des résultats incorrects.

4.8 Types "Typecode" et "Principal"

Les types particuliers "Typecode" et "Principal" peuvent servir pour obtenir des valeurs de type qui pourront être utilisées en tant que paramètres pour des opérations. En tant que tels ils doivent être définis par un lien de langage pour une réalisation particulière. L'Annexe B présente un codage particulier pour les types Typecode relevant de la spécification CORBA.

4.9 Déclaration d'exception

Les déclarations d'exception permettent de déclarer des structures de données de type structuré qui peuvent être retournées pour indiquer qu'une condition d'exception est apparue au cours de l'exécution d'une requête. La syntaxe de déclaration des exceptions est la suivante:

```
<except_dcl>             ::="exception" <identifier> "{" <member>* "}"
```

Chaque exception est caractérisée par son identificateur IDL ODP, par un identificateur de type d'exception et par le type de la valeur de retour associée (comme spécifié par le terme <member> de la déclaration). Si une requête donne un résultat de type exception, la valeur de l'identificateur de cette exception est mise à la disposition du programmeur pour déterminer la nature de l'exception particulière qui est apparue.

NOTE – Il s'agit d'une règle neutre pour ce qui est de l'architecture. Certaines exceptions sont toutefois réservées. On pourra les trouver dans l'Annexe A.

Si la déclaration d'une exception comporte des membres, le programmeur aura accès aux valeurs de ces membres lorsqu'une exception apparaîtra. Si aucun membre n'est spécifié, aucune information additionnelle ne sera disponible lorsqu'une exception apparaîtra.

4.10 Déclaration d'opération

En langage IDL ODP, les déclarations d'opération sont analogues aux déclarations de fonctions en langage C. Leur syntaxe est la suivante:

```
<op_dcl>               ::= [<op_attribute> ] <op_type_spec>
                                     <identifier><parameter_dcls>

                           [<raises_expr>] [ <context_expr>]



<op_type_spec>         ::= <param_type_spec>

                           | "void"
```

Une déclaration d'opération se compose des éléments suivants:

- un attribut facultatif d'opération, qui spécifie la sémantique d'invocation que le système de communication doit offrir lorsque l'opération est invoquée. Les attributs d'opération sont décrits au 4.10.1;
- le type de résultat retourné par l'opération; ce type peut être un de ceux qui sont définis dans le langage IDL ODP. Les opérations qui ne renvoient pas de résultat doivent spécifier le type "void";
- un identificateur, qui désigne l'opération dans le domaine de visibilité de l'interface qui la définit;
- une liste de paramètres, qui spécifie zéro, une ou plusieurs déclarations de paramètres pour l'opération. La déclaration de paramètres est décrite au 4.10.2;
- une expression facultative de propagation d'exceptions, qui indique les exceptions qui peuvent être propagées à la suite d'une invocation d'opération. Les expressions de propagation d'exceptions sont décrites au 4.10.3;
- une expression contextuelle facultative, qui indique les éléments du contexte de requête qui peuvent être consultés par la méthode de mise en œuvre de l'opération. Les expressions contextuelles sont décrites au 4.10.4.

4.10.1 Attribut d'opération

L'attribut d'opération spécifie la sémantique d'invocation que le service de communication doit fournir pour l'invocation d'une opération particulière. Un attribut d'opération est facultatif et la syntaxe de sa spécification est la suivante:

```
<op_attribute>         ::= "oneway"
```

Lorsqu'un client invoque une opération assortie de l'attribut oneway (dans un seul sens), la sémantique d'invocation est de type "au mieux", ce qui ne garantit pas l'aboutissement de l'appel; l'exécution "au mieux" implique que l'opération sera invoquée une seule fois au plus. Une opération assortie de l'attribut oneway ne doit jamais contenir de paramètres de sortie et doit toujours spécifier un résultat en retour de type void. Une opération définie avec l'attribut oneway ne peut pas comporter d'expression de propagation d'exception; l'invocation d'une telle opération peut toutefois propager une exception normalisée.

Si aucun attribut d'opération <op_attribute> n'est spécifié, la sémantique d'opération ne pourra être invoquée qu'une seule fois au plus en cas de propagation d'exceptions et exactement une seule fois en cas de retour favorable de l'invocation d'opération.

4.10.2 Déclarations de paramètres

Les déclarations de paramètres contenues dans les déclarations d'opération IDL ODP ont la syntaxe suivante:

```
<parameter_dcls>       ::=  "(" <param_dcl> { "," <param_dcl> }* ")"

                       |    "(" ")"

<param_dcl>            ::=  <param_attribute>
                                  <param_type_spec><simple_declarator>

<param_attribute>      ::=  "in"

                       |    "out"

                       |    "inout"
```

```
<param_type_spec>          ::=  <base_type_spec>
                           |    <string_type>
                           |    <wide_string_type>
                           |    <wide_string_type>
                           |    <scoped_name>
```

Une déclaration de paramètre doit comporter un attribut directionnel qui informe le service de communication, chez le client comme chez le serveur, du sens dans lequel le paramètre doit être transmis. Les attributs directionnels sont les suivants:

- `in` – Le paramètre est transmis du client au serveur;
- `out` – Le paramètre est transmis du serveur au client;
- `inout` – Le paramètre est transmis dans les deux sens.

L'on escompte que la mise en œuvre ne tentera pas de modifier un paramètre de type `in`. La capacité d'effectuer quand même une telle tentative est propre au mode de traduction linguistique; l'effet d'une telle action n'est pas défini.

Si une exception est propagée à la suite d'une invocation, les valeurs du résultat retourné et tous les paramètres des types `out` et `inout` sont indéfinis.

Lorsqu'une `chaîne, chaîne large` ou `séquence` non bornée est transmise sous la forme d'un paramètre `inout`, la valeur retournée ne peut pas être plus longue que la valeur d'entrée.

4.10.3 Expressions de propagation d'exceptions

Une expression de propagation d'exceptions (type `raises`) spécifie les exceptions qui peuvent être propagées à la suite de l'invocation d'une opération. La syntaxe de spécification correspondante est la suivante:

```
<raises_expr>          ::=  "raises" "(" <scoped_name> { "," <scoped_name> }* ")"
```

Le nom détecté `<scoped_name>` contenu dans l'expression `raises` doit désigner des exceptions préalablement définies.

En plus des éventuelles exceptions propres aux opérations, spécifiées dans l'expression de propagation, il existe un ensemble d'exceptions normalisées qui peuvent être signalées.

L'absence d'expression de propagation d'exceptions au sujet d'une opération implique qu'il n'existe pas d'exceptions propres à cette opération. Les invocations d'une telle opération restent susceptibles de recevoir une des exceptions normalisées.

4.10.4 Expressions contextuelles

Une expression contextuelle (de type `context`) spécifie les éléments du contexte client qui peuvent avoir une incidence sur la suite donnée à la requête par l'objet. Sa syntaxe de spécification est la suivante:

```
<context_expr>         ::=  "context" "(" <string_literal>
                                  {"," <string_literal> }*")"
```

Le système d'exécution garantit que la valeur (éventuellement) associée à chaque littéral-chaîne `<string_literal >` dans le contexte du client sera mise à la disposition de l'implémentation par objets lorsque la requête sera acheminée. Le gestionnaire de requêtes pour objets répartis (ORB, *object request broker*) et/ou l'objet a la possibilité d'utiliser les informations contenues dans ce *contexte de requête* au cours de la résolution et de l'exécution de la requête.

L'absence d'une expression contextuelle indique qu'aucun contexte de requête n'est associé aux requêtes pour cette opération.

Chaque littéral-chaîne (`string_literal`) est une séquence de longueur arbitraire de caractères alphabétiques, numériques et formatants [point ("."), soulignement ("_") et astérisque ("*")]. Le premier caractère de la chaîne ne doit jamais être alphabétique. Un astérisque ne peut être utilisé que comme dernier caractère de la chaîne. Certaines implémentations peuvent utiliser le caractère de point pour partitionner le champ de nom.

4.11 Déclaration d'attribut

On peut affecter à une interface aussi bien des attributs que des opérations; en tant que tels, les attributs sont définis comme faisant partie d'une interface. Une définition d'attribut est logiquement équivalente à la déclaration d'une paire de fonctions de dispositif d'accès: l'une pour récupérer la valeur de l'attribut, l'autre pour déterminer la valeur de celui-ci.

La syntaxe des déclarations d'attribut est la suivante:

```
<attr_dcl>              ::= ["readonly"]"attribute"<param_type_spec>
                                  <simple_declarator>

                            {","<simple_declarator>}*
```

Le mot clé facultatif `readonly` (lecture seulement) indique qu'il n'existe qu'une seule fonction de dispositif d'accès: celle de récupération de valeur. Soit l'exemple suivant:

```
interface foo {
      enum material_t {rubber, glass};
      struct position_t {
      float x, y;
      };

      attribute float radius;
      attribute material_t material;
      readonly attribute position_t position;
      ...
};
```

Ces déclarations d'attribut sont équivalentes au fragment de pseudo-spécification suivant:

```
float           _get_radius  ();
void            _set_radius  (in float r);
material_t      _get_material();
void    _       set_material (in material_t m);
position_t      _get_position ();
```

Le nom d'attribut est soumis aux règles de visibilité/détection de nom du langage IDL ODP; ces règles garantissent que les noms des accès ou des fonctions n'entreront pas en collision avec de quelconques noms d'opération dont la spécification en langage IDL ODP est autorisée.

Les opérations faisant intervenir des attributs ne renvoient d'erreurs qu'au moyen d'exceptions normalisées.

Les attributs sont hérités. Un nom d'attribut ne peut pas être redéfini comme étant d'un type différent. Voir 4.12 pour de plus amples renseignements sur les contraintes de redéfinition et sur le traitement des ambiguïtés.

4.12 Module de spécification CORBA

Le domaine de visibilité des noms CORBA (appelé le module CORBA) est réservé dans le langage IDL ODP. Les définitions au sein du module CORBA devraient seulement être utilisées lors de l'écriture de définitions d'interfaces spécifiques de domaine, et ne devraient pas être utilisées dans les descriptions générales d'interface de calcul ODP.

4.13 Noms et domaines de visibilité

L'ensemble d'un fichier de langage IDL ODP forme un domaine de visibilité de noms. En outre, les sortes de définitions suivantes forment des domaines de visibilité imbriqués:

- module;
- interface;
- structure;
- union;
- opération;
- exception.

Les identificateurs des sortes de définitions suivantes sont inscrits dans un domaine de visibilité:

- types;
- constantes;
- valeurs d'énumération;
- exceptions;
- interfaces;
- attributs;
- opérations.

Un identificateur ne peut être défini qu'une seule fois dans un domaine de visibilité. On peut cependant redéfinir des identificateurs dans des domaines de visibilité imbriqués.

En raison d'éventuelles restrictions imposées par de futurs liens linguistiques, les identificateurs IDL ODP ne sont pas sensibles à l'inversion de casse – c'est-à-dire que deux identificateurs qui ne diffèrent que par la casse de leurs caractères sont considérés comme étant des redéfinitions réciproques. Toutes les références à une définition doivent cependant utiliser la même casse que l'instance de définition (ce qui permet d'effectuer des mappages naturels avec des langages sensibles à l'inversion de casse).

Les noms de type définis dans un domaine de visibilité sont disponibles pour usage immédiat dans le cadre de ce domaine. Voir en particulier 4.7.2 sur les cycles de définition des types.

Un nom peut être utilisé sous une forme non qualifiée à l'intérieur d'un domaine de visibilité particulier; ce nom sera résolu par des recherches centrifuges dans les domaines englobants successifs. Une fois qu'un nom non qualifié a été utilisé dans un domaine, il ne peut plus être redéfini – c'est-à-dire que si l'on a utilisé un nom dans un domaine qui englobe le domaine actuel, on ne peut plus redéfinir une version de ce nom dans ce domaine actuel. De telles redéfinitions produisent une erreur de compilation.

Un nom qualifié (qui est de la forme <scoped-name>:: <identifier>) est résolu comme suit: d'abord par association du qualificateur <scoped-name> avec un domaine de visibilité puis par localisation de la définition de l'identificateur <identifier> à l'intérieur du domaine S. Cet identificateur doit être défini directement dans S ou (si S est une interface) être hérité dans S. L'identificateur n'est pas recherché dans les domaines englobants.

Lorsqu'un nom qualifié commence par les caractères doubles "::", le processus de résolution commence par le plus petit domaine de visibilité de noms englobant et localise les identificateurs suivants dans le nom qualifié, par la règle décrite dans l'alinéa précédent.

Chaque définition IDL ODP contenue dans un fichier y possède un nom global qui est construit comme suit.

Avant de commencer l'exploration d'un fichier contenant une spécification IDL ODP, le nom de la racine actuelle est initialement vide ("") et le nom du domaine de visibilité actuel est initialement vide (""). Chaque fois qu'un mot clé module est rencontré, on ajoute la chaîne "::" et l'identificateur associé au nom de la racine actuelle; dès détection de la fin du mot module, le postambule "::" et l'identificateur sont supprimés du nom de la racine actuelle. Chaque fois qu'un mot clé de type interface, struct, union ou exception est rencontré, la chaîne "::" et l'identificateur associé sont ajoutés au nom du domaine actuel; dès détection de la fin du mot clé interface, struct, union ou exception, le postambule "::" et l'identificateur sont supprimés du nom du domaine actuel. Par ailleurs, un nouveau domaine sans nom est ouvert lorsque les paramètres d'une déclaration d'opération sont traités; cela permet aux noms des paramètres de copier d'autres identificateurs; lorsque le traitement du paramètre est effectué, le domaine sans nom est fermé.

Le nom global d'une définition IDL ODP est la concaténation de la racine actuelle, du domaine de visibilité actuel, d'une définition "::" et de l'<identificateur>, qui est le nom local de cette définition.

Les procédures d'héritage produisent des duplicatas (copies miroirs) des identificateurs hérités – c'est-à-dire qu'elles introduisent des noms dans l'interface dérivée, mais que ces noms sont considérés comme étant sémantiquement "conformes" à la définition initiale. Deux duplicatas du même original (comme cela ressort du diagramme en "diamant" de la Figure 1) introduisent un même nom dans l'interface dérivée et n'entrent pas en conflit l'un avec l'autre.

L'héritage apporte de multiples noms globaux IDL ODP aux identificateurs hérités. Soit l'exemple suivant:

```
interface A{
      exception E {
      long L;
      };
      void f() raises (E);
};


interface B:A {
      void g() raises(E);
};
```

Dans cet exemple, l'exception est désignée par les noms globaux `::A::E` et `::B::E`.

Une ambiguïté peut apparaître dans les spécifications en raison de l'imbrication des domaines de dénomination. Par exemple:

```
interface A{
      typedef string <128> string_t;
};


interface B{
      typedef string <256> string_t;
};


interface C:A,B {
      attribute string_t Title; /* AMBIGUOUS */
};
```

En langage C, la déclaration d'attribut est ambiguë, car le compilateur ne sait pas quelle chaîne `string_t` est désirée. Les déclarations ambiguës produisent des erreurs de compilation.

4.14 Différences par rapport au langage C++

La grammaire du langage IDL ODP, tout en s'efforçant de se conformer à la syntaxe du langage C++, est un peu plus restrictive. Les restrictions actuelles sont les suivantes:

- un type de retour de fonction est obligatoire;
- un nom doit toujours être fourni pour chaque paramètre formel applicable à une déclaration d'opération;
- une liste de paramètres composée du simple jeton `void` n'est pas autorisée comme synonyme d'une liste vide de paramètres;
- des étiquettes sont requises pour les structures, pour les réunions d'ensembles discriminés et pour les énumérations;
- les types "entier" ne peuvent pas être simplement qualifiés par "int" ou "unsigned"; ils doivent être déclarés explicitement comme étant courts (`short`) ou longs (`long`);
- un caractère (`char`) ne peut pas être qualifié par les mots clés `signed` et `unsigned`.

Annexe A

Exceptions normalisées et réservées

(Cette annexe fait partie intégrante de la présente Recommandation | Norme internationale)

La présente annexe présente un ensemble normalisé d'exceptions qui ont été définies pour une infrastructure ODP. Ces identificateurs d'exceptions peuvent être retournés en tant que résultat de toute invocation d'opération, quelle que soit la spécification d'interface. Les exceptions normalisées ne peuvent pas être énumérées dans des expressions de propagation (type `raises`).

De façon à limiter la complexité dans le traitement des exceptions normalisées, l'ensemble de ces exceptions doit être maintenu dans des dimensions gérables. Cette contrainte oblige à définir des classes d'équivalence au lieu d'énumérer de nombreuses exceptions similaires. Par exemple, une invocation d'opération peut échouer à différents points à cause de l'impossibilité d'allouer de la mémoire vive. Plutôt que d'énumérer plusieurs exceptions différentes, correspondant aux différentes façons dont un défaut d'allocation de mémoire provoque l'exception (au cours du classement, du déclassement, chez le client, dans la mise en œuvre d'un objet, lors de la mise en paquets pour le réseau, etc.), on définit une seule exception correspondant à un échec d'allocation de mémoire vive. Chaque exception normalisée est assortie d'un code mineur qui désigne la sous-catégorie de l'exception; l'attribution de valeurs aux codes mineurs relève de chaque mise en œuvre du gestionnaire ORB.

Chaque exception normalisée comporte également un code de statut d'exécution qui prend une des valeurs suivantes: {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE}. Ces valeurs ont les significations suivantes:

COMPLETED_YES	L'implémentation de l'objet a terminé le traitement avant la propagation de l'exception.
COMPLETED_NO	L'implémentation de l'objet n'a jamais été entreprise avant la propagation de l'exception.
COMPLETED_MAYBE	Le statut d'achèvement de l'implémentation est indéterminé.

Les exceptions normalisées sont définies ci-dessous. Les objets clients doivent être préparés à manipuler les exceptions système qui ne sont pas sur la liste, à la fois car de futures versions de cette spécification peuvent définir des exceptions normalisées additionnelles et car les réalisations d'infrastructure ODP peuvent provoquer des exceptions système non normalisées.

```
#define ex_body {unsigned long minor; completion_status completed}


enum completion_status {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE};

enum exception_type {NO_EXCEPTION, USER_EXCEPTION, SYSTEM_EXCEPTION};


exception UNKNOWN                    ex_body;    //exception inconnue
exception BAD_PARAM                  ex_body;    //transmission d'un paramètre invalide
exception NO_MEMORY                  ex_body;    //échec d'allocation de mémoire vive
exception IMP_LIMIT                  ex_body;    //violation de limite de mise en œuvre
exception COMM_FAILURE               ex_body;    //panne de communication
exception INV_OBJREF                 ex_body;    //invalidité de la référence d'objet
exception NO_PERMISSION              ex_body;    //tentative d'opération non autorisée
exception INTERNAL                   ex_body;    //erreur interne du gestionnaire ORB
exception MARSHAL                    ex_body;    //erreur de classement de paramètre/
                                                   résultat
exception INITIALIZE                 ex_body;    //échec d'initialisation du gestionnaire
                                                   ORB
exception NO_IMPLEMENT               ex_body;    //exécution d'opération indisponible
exception BAD_TYPECODE               ex_body;    //code de type erroné
exception BAD_OPERATION              ex_body;    //opération invalide
exception NO_RESOURCES               ex_body;    //ressources insuffisantes pour la
                                                   requête
exception NO_RESPONSE                ex_body;    //réponse à la requête pas encore
                                                   disponible
```

```
exception PERSIST_STORE            ex_body;    //panne de mémoire persistante

exception BAD_INV_ORDER            ex_body;    //invocations de programme déclassées

exception TRANSIENT                ex_body;    //panne transitoire – réémettre la
                                                 requête

exception FREE_MEM                 ex_body;    //libération de mémoire impossible

exception INV_IDENT                ex_body;    //syntaxe d'identificateur invalide

exception INV_FLAG                 ex_body;    //fanion spécifié invalide

exception INTF_REPOS               ex_body;    //erreur d'accès au répertoire
                                                 d'interface

exception BAD_CONTEXT              ex_body;    //erreur de traitement d'objet contextuel

exception OBJ_ADAPTER              ex_body;    //détection de panne par adaptateur
                                                 d'objet

exception DATA_CONVERSION          ex_body;    //erreur de conversion de données

exception OBJECT_NOT_EXIST         ex_body;    //objet non existant, détruire référence

exception TRANSACTION_REQUIRED     ex_body;

exception TRANSACTION_ROLLEDBACK   ex_body;

exception INVALID_TRANSACTION      ex_body;
```

A.1 Non-existence d'objet

L'exception OBJECT_NOT_EXIST est levée chaque fois qu'une invocation sur un objet détruit a été effectuée. C'est un rapport autoritaire de faute "grave". Toute personne le recevant est autorisée à (et est même supposée) détruire toutes les copies de cette référence d'objet et à lancer d'autres procédures appropriées du style "récupération finale".

A.2 Exceptions de transaction

L'exception TRANSACTION_REQUIRED indique que la demande transportait un contexte de transaction nul, mais qu'une transaction active est demandée. L'exception TRANSACTION_ROLLEDBACK indique que le retour en arrière de la transaction associée à la demande a déjà été effectué ou marqué. Ainsi, l'opération indiquée soit ne pouvait pas être effectuée, soit n'a pas été effectuée car tout nouveau calcul pour le compte de la transaction serait inutile.

L'exception INVALID_TRANSACTION indique que la demande transportait un contexte de transaction non valide. Par exemple, cette exception pourrait être levée si une erreur survenait lors d'une tentative d'enregistrement d'une ressource.

Annexe B

Codage des types dans la spécification CORBA

(Cette annexe ne fait pas partie intégrante de la présente Recommandation | Norme internationale)

Dans la spécification CORBA, l'interface IDL ODP pour les codes de type est la suivante:

```
module CORBA {
   enum TCKind {
      tk_null, tk_void,
      tk_short, tk_long, tk_ushort, tk_ulong,
      tk_float, tk_double, tk_boolean, tk_char,
      tk_octet, tk_any, tk_TypeCode, tk_Principal, tk_objref,
      tk_struct, tk_union, tk_enum, tk_string,
      tk_sequence, tk_array, tk_alias, tk_except,
      tk_longlong, tk_ulonglong, tk_longdouble,
      tk_wchar, tk_wstring, tk_fixed_pt
      };
   interface TypeCode {
      exception               Bounds{};
      exception               BadKind{};

      // for all TypeCode kinds
      boolean                 equal (in TypeCode tc);
      TCkind                  kind ();

      // for tk_objref, tk_struct, tk_union, tkenum, tk_alias, and tk_except
      RepositoryId            id () raises (BadKind) ;

      // for tk_objref, tk_struct, tk_union, tkenum, tk_alias, and tk_except
      Identifier              name () raises (BadKind) ;

      // for tk_struct, tk_union, tk_enum, and tk_except
      unsigned long           member_count () raises (BadKind);
      Identifier              member_name (in unsigned long index)
                                   raises (BadKind, Bounds);

      // for tk_struct, tk_union, and tk_except
      TypeCode                member_type (in unsigned long index)
                                   raises (BadKinds, Bounds);

      //for tk_union
      any                     member_label (in unsigned long index)
                                   raises (BadKind, Bounds);
      TypeCode                discriminator_type () raises (BadKind);
      long                    default_index () raises (BadKind);
      //for tk_string, tk_sequence, and tk_array
      unsigned long length () raises (BadKind);
```

```
        //for tk_sequence, tk_array, and tk_alias
        TypeCode                content_type () raises (BadKind);

        // deprecated interface
        long                    parameter_count ();
        any                     parameter (in long index) raises (Bounds) ;
    };
```

Grâce aux opérations ci-dessus, n'importe quel code de type peut être décomposé en ses éléments constituants.

SÉRIES DES RECOMMANDATIONS UIT-T

Série A	Organisation du travail de l'UIT-T
Série B	Moyens d'expression: définitions, symboles, classification
Série C	Statistiques générales des télécommunications
Série D	Principes généraux de tarification
Série E	Exploitation générale du réseau, service téléphonique, exploitation des services et facteurs humains
Série F	Services de télécommunication non téléphoniques
Série G	Systèmes et supports de transmission, systèmes et réseaux numériques
Série H	Systèmes audiovisuels et multimédias
Série I	Réseau numérique à intégration de services
Série J	Transmission des signaux radiophoniques, télévisuels et autres signaux multimédias
Série K	Protection contre les perturbations
Série L	Construction, installation et protection des câbles et autres éléments des installations extérieures
Série M	RGT et maintenance des réseaux: systèmes de transmission, de télégraphie, de télécopie, circuits téléphoniques et circuits loués internationaux
Série N	Maintenance: circuits internationaux de transmission radiophonique et télévisuelle
Série O	Spécifications des appareils de mesure
Série P	Qualité de transmission téléphonique, installations téléphoniques et réseaux locaux
Série Q	Commutation et signalisation
Série R	Transmission télégraphique
Série S	Equipements terminaux de télégraphie
Série T	Terminaux des services télématiques
Série U	Commutation télégraphique
Série V	Communications de données sur le réseau téléphonique
Série X	**Réseaux pour données et communication entre systèmes ouverts**
Série Y	Infrastructure mondiale de l'information
Série Z	Langages de programmation