XcodeML/C++ 仕様書

**V1.1J (Mar 30, 2016)**

**XcalableMP/Omni Compiler Project**

改版履歴

XcodeML/C Version 0.91J

* 配列要素の参照のXML要素を変更。
* subArrayRef要素を変更。
* indexRange要素を追加。

XcodeML/C++ draft 0.1J

* C++対応ドラフト初版

XcodeML/C++ 1.0J

* C++対応初版。2015/10/15

XcodeML/C++ 1.1J

* 二章を修正（正規化を加筆、fullName属性について）。2016/3/25

目次

1 はじめに 1

2 翻訳単位とXcodeProgram要素 2

2.1 name要素 2

2.2 value要素 4

3 typeTable要素とデータ型定義要素 5

3.1 データ型識別名 5

3.2 typeName要素 6

3.3 データ型定義要素属性 7

3.4 basicType要素 7

3.5 pointerType要素 8

3.6 functionType要素 9

3.7 arrayType要素 9

3.8 unionType要素 10

3.9 structType要素とclass要素（C++） 10

3.9.1 inheritedFrom要素（C++） 12

3.10 enumType要素 12

3.11 parameterPack要素（C++） 13

4 シンボルリスト 15

4.1 id要素 15

4.2 globalSymbols要素 16

4.3 symbols要素 17

5 globalDeclarations要素とdeclarations要素 18

5.1 globalDeclarations要素 18

5.2 declarations要素 18

5.3 functionDefinition要素 19

5.3.1 operator要素（C++） 20

5.3.2 constructor要素（C++） 20

5.3.3 destructor要素（C++） 21

5.3.4 params要素 21

5.4 varDecl要素 21

5.5 functionDecl要素 22

5.6 usingDecl要素（C++） 22

6 文の要素 24

6.1 exprStatement要素 24

6.2 compoundStatement要素 24

6.3 ifStatement要素 24

6.4 whileStatment要素 25

6.5 doStatement要素 25

6.6 forStatement要素 25

6.7 rangeForStatement要素（C++） 26

6.8 breakStatement要素 26

6.9 continueStatement要素 27

6.10 returnStatment要素 27

6.11 gotoStatement要素 27

6.12 tryStatement要素（C++） 27

6.13 throwStatement要素（C++） 28

6.14 catchStatement要素（C++） 28

6.15 statementLabel要素 28

6.16 switchStatement要素 28

6.17 caseLabel要素 29

6.18 gccRangedCaseLabel要素 29

6.19 defaultLabel要素 29

6.20 pragma要素 30

6.21 text要素 30

7 式の要素 31

7.1 定数の要素 31

7.2 変数参照の要素（Var要素、varAddr要素、arrayAddr要素） 32

7.3 pointerRef要素 33

7.4 arrayRef要素 33

7.5 メンバの参照の要素（C++拡張） 34

7.6 メンバポインタの参照の要素（C++） 35

7.7 複合リテラルの要素（新規） 36

7.8 thisExpr要素（C++） 38

7.9 assignExpr 要素 38

7.10 2項演算式の要素 38

7.11 単項演算式の要素 39

7.12 functionCall要素 40

7.12.1 arguments要素 40

7.13 commaExpr要素 40

7.14 インクリメント・デクリメント要素（postIncrExpr, postDecrExpr, preIncrExpr, preDecrExpr） 41

7.15 castExpr要素（廃止予定） 41

7.16 キャスト要素（staticCast, dynamicCast, constCast, reinterpretCast）（C++） 41

7.17 condExpr要素 42

7.18 gccCompoundExpr要素 42

7.19 newExpr要素とnewExprArray要素 42

7.20 deleteExpr要素とdeleteArrayExpr要素 43

7.21 lambdaExpr要素 43

7.21.1 captures要素 43

8 テンプレート定義要素（C++） 45

8.1 typeParams要素 45

8.2 structTemplate要素とclassTemplate要素 45

8.3 functionTemplate要素 47

8.4 aliasTemplate要素 48

9 テンプレートインスタンス要素（C++） 50

9.1 typeArguments要素 50

9.2 typeInstance要素 50

9.3 functionInstance要素 51

10 XcalableMP固有の要素 53

10.1 coArrayType要素 53

10.2 coArrayRef要素 53

10.3 subArrayRef要素 54

10.4 indexRange要素 54

11 その他の要素・属性 55

11.1 is\_gccExtension属性 55

11.2 gccAsm要素、gccAsmDefinition要素、gccAsmStatement要素 55

11.3 gccAttributes要素 57

11.4 builtin\_op要素 59

11.5 is\_gccSyntax属性 59

11.6 is\_modified属性 59

12 未検討項目 61

13 コード例 62

# はじめに

この仕様書は、プログラミング言語CおよびC++の中間表現であるXcodeMLを記述する。XcodeMLは、以下の特徴を持つ。

* CまたはC++の一つの翻訳単位（Translation Unit）のプログラムを入力にとり、各種の情報をXML形式で表現する。これにより、各種のプログラム変換処理を行いやすくするとともに、人間にとっても可読（human-readable）なフォーマットを持つ。
* ソースコードを意味的（semantic）に表現した抽象構文木構造を保持する。これにより、XcodeMLをソースコードに再度変換することができる。ここで、「意味的」としている理由は、プログラム変換処理が取り扱う必要のないソースコードの書き方の違いは、XcodeML上では表現しないからである。
* ソースコード上のの型情報を、抽象構文木構造から独立した形で表現するとともに、抽象構文木内の式それぞれについて、その式が何型として扱われているかを付加する
* ソースコード上の各種シンボルのスコープ範囲が明確となる構造を取る。

# 翻訳単位とXcodeProgram要素

ソースファイルに、＃include指定されたファイルを再帰的にすべて展開したものを、翻訳単位と呼ぶ。翻訳単位はXcodeProgram要素で表現される。

<XcodeProgram>

typeTable要素（3章）

globalSymbols要素（4.2節）

globalDeclarations要素（5.1節）

</XcodeProgram>

属性（optional）: compiler-info, version, time, language, source

XcodeMLファイルのトップレベルのXML要素は、XcodeProgram 要素である。XcodeProgram 要素は以下の子要素を含む。

* typeTable要素　– プログラムで利用されているデータ型の情報
* globalSymbols要素 – プログラムで利用されている大域変数の情報
* globalDeclarations 要素 – 関数、変数宣言などの情報

XcodeProgram要素は、属性として以下の情報を持つことができる

* compiler-info　－　CtoC コンパイラの情報
* version　－　CtoC コンパイラのバージョン情報
* time　－　コンパイルされた日時
* language　－　ソース言語情報
* source　－　ソース情報

## ソースコードの正規化

XcodeMLの設計方針は、XcodeMLで表現されたプログラムを入力にとって各種の解析処理をおこなった上で処理結果をXcodeMLとして出力するという処理（以下、この種の処理を「XcodeML解析処理」と呼ぶことにする）に適した構造を持つ、ということを主眼においている。このため、プログラムの意味が同一となるような記述方法が複数存在する場合、そのいずれかに「正規化」して扱うという設計方針をおいている。具体的には次のとおりである。

* XcodeMLの木構造が演算子の結合性を反映するため、ソースコード上に明示的に書かれた丸カッコはXcodeMLでは明示的には表現しない。つまり、ソースコード上でxという式を表現するXcodeMLとその式をカッコでかこんだ (x) という式を表現するXcodeMLは同一のものとなる。
* 組み込み型に対する単項プラス演算子はXcodeMLでは表現しない。（C++の演算子オーバーロードを用いた式は演算子の一種とはみなされず関数呼び出しの一種として表現されるため、これについては単項プラス演算子も明示的に表現される）
* 宣言はcompoundStatement要素の先頭でのみ取り扱う。すなわち、ソースコード上で複文の途中に新たな変数宣言が出現した場合には、その位置から複文末尾までの範囲を囲うcompoundTextを生成し、その先頭に配置する。
* データ型の宣言のうち、下記に述べる「複雑な型」に該当しないものについては、その宣言内容はtypeTableで表現し、globalDeclarations要素の中には持ち込まない。これにより、XcodeML解析処理部が新たな型をXcodeMLに追加する際、そのソースコード上での配置を考える必要がない。  
  下記に、typeTableのみでは表現不可能な「複雑な型」を述べる。
  + C++のローカルクラス
  + 入れ子のクラス（クラス内に別のクラス宣言を持つ場合、その両方ともが複雑な型とみなされる）
  + 無名クラス
* C++のクラス（構造体、共用体を含む）のメソッドの定義は、クラス宣言内で書く方法と、クラス宣言の外で書く方法の二種類の定義方法がある。これについては、前述の「複雑な型」でない限りは、すべて「クラス宣言の外に書く方法」に相当する形のXcodeMLとして表現する。
* クラスのメンバや親クラスについてのアクセス指定子は、個々のメンバ・親クラスに対してそれぞれ指定されている形で扱う。すなわち、ソースコード上に実際にどこにアクセス指定子が書かれていたかはXcodeMLでは表現せず、各メンバにどのようなアクセス指定がかかっていたかのみを表現する。
* 名前空間を用いた名前については、次の節で解説するfullName属性によって完全修飾名を表現する形で扱う。
  + 名前の定義については、構文的なnamespace｛｝の入れ子構造はXcodeMLでは表現せず、個々の名前がどのようなnamespaceに所属すべきかのみを表現していることになる。
  + 名前の使用については、using namespaceによる名前空間のインポートをXcodeMLでは表現せず、全ての名前を完全修飾名で扱う。
* 演算子オーバーロードを用いた式については、XcodeMLでは演算子としては扱わず、 operator キーワードを明示指定した関数呼び出しに相当する形で扱う。同じ演算子に対するオーバーロード方法にはクラスメソッドの形のものとグローバル関数の形のものがあり、このどちらの呼び出しになっているかが明示される形になる。また、それぞれfullName属性がつくので、ADRによる名前空間検索の解決結果も反映される。

備考： 7.3節に出現する「pointerRefとvarAddrをまとめてVarとして表現」する話なども正規化の一種として考えることができそうである。

これは「XcodeMLとしてはどちらの表現も可能」なので、ここで解説している正規化とは少し位置づけが違うようにも見えるが、上記で述べた「複雑な型」を表現するためにはglobalDeclarationsだけで任意の型を表現できる必要があるので、結局「正規化せずに全てglobalDeclarationsだけで扱う」表現と「正規化できるものは正規化して表現する」のどちらの表現も可能ということになる。

つまり、「まずは構文の木構造をそのまま反映しただけのXcodeML（正規化されていない）」と、「XcodeML解析処理に適した正規化がほどこされたあとのXcodeML」を考えて、前者の仕様と後者への正規化処理の仕様に分けて記述した方が厳密な議論ができるように思われる。

ただし、下記のfullNameの話やoperator呼び出しの種類、decltypeによる型などは、ソースコードには書かれていないがClangAST的には解決結果が保持されているので、これらは「正規化されていないXcodeML」の時点で情報を付与しておくべきである。また、解決結果だけがあれば情報は足りているので、「正規化されていないXcodeML」がもともとのソースコード上の構文的な構造を完全に反映する必要があるわけではない。このように考えると、「フェーズ１の正規化」と「フェーズ2の正規化」があり、フェーズ1の正規化は必ずおこなう（XcodeMLとして二種類の表現をするコースがそもそも準備されない）、という風に考える必要がある。

## fullName属性

C++のスコープ解決演算子による修飾をおこなった形の「フルネーム」を指定するためのXML属性である。

次章以降で解説する各種の要素のうち、ソースコード上での「名前」を表現する要素について、適宜挿入される共通の構造である。

fullName=”[::][スコープ名::]…(名前|operator 演算子名)”

* 下記の各属性に適宜挿入される。typeTable要素に含まれるもの：
  + name要素
* globalSymbols要素および（symbols要素に含まれるもの：
  + name要素
* globalDeclarations 要素およびdeclarations要素に含まれるもの：
  + name要素
  + operator要素
  + Var要素
  + varAddr要素
  + function要素
  + funcAddr要素
  + arrayRef要素
  + arrayAddr要素
  + memberRef要素
  + memberAddr要素
  + memberArrayRef要素
  + memberArrayAddr要素

fullName属性は、0個以上のスコープ名と「::」の組に続く名前（演算子名のときにはoperatorキーワードを明示）を文字列として持つ。スコープ名と「::」は、そのオブジェクトのスコープ名を省略せずすべて書き下した形でなければならない。また、スコープ名にnamespaceの別名定義があるとき、元のスコープ名に正規化した形で表現される。using文があるとき、スコープ名は省略しない形に正規化される。

例:

以下のプログラムで、

namespace NS {

int a; // (1)

}

NS::a = 10; // (2)

(1)におけるaは、以下のように表現される。

<name type=”int”>a</name>

(2)におけるaは、以下のように表現される。

<name type=”int” fullName=”NS::a”>a</name>

例:

　以下のプログラムで、

struct S {

int data;

int foo(int n) { return n + 1; }

};

int S :: \*d = &S :: data; // (1)

int (S :: \*f)(int) = &S :: foo; // (2)

struct S s1;

int \*p = &s1.data; // (3)

(1),(2)のdとfの名前は、それぞれ以下のように表現される。P１はintへのポインタ型である。

<name type="P1" fullName=”S::d”>d</name>

<name type="P1" fullName=”S::f”>f</name>

(1)と(2)の右辺式は、それぞれ以下のように表現される。Sは変数でないのでmemberAddr要素は用いられず、data変数のスコープと解釈する。

<varAddr type="P0" scope="global" fullName=”S::data”>data</varAddr>

<varAddr type="P0" scope="global" fullName=”S::Foo”>foo</varAddr>

(3)の右辺式は、以下のように表現される。s1は変数名なので、s1.dataはmemberAddr要素で表現される。

<memberAddr type="P5" member=”data” fullName=”S::data”>

<varAddr type="P4" scope="global" fullName=”s1”>s1</varAddr>

</memberAddr>

## value要素

globalSymbols要素、symbols要素およびglobalDeclarations要素、declarations要素中で、初期化式を持つ変数宣言を表現する際の初期値の式を表現する。

<value>

[ 式の要素（7章） *or*　value要素

… ]

</value>

属性: なし

備考：C++においては初期化式に副作用を持つ式を書くことが可能であり、特にグローバルスコープを持つ変数についての初期化式の実行順序は翻訳単位内での出現順を守ることになっているため、これらの情報をglobalSymbols部に持たせるのではなくglobalDeclarations部のみに持たせるのが適切な分担になる（局所スコープ内のsymbols部とdeclarations部についてもこれに合わせた方がよい）と考えられる。そうすると、この節は2章に置く必要がなく、5章以降に配置するのが適切である。

{ } で囲まれた式の並びは、value要素のネストで表現する。

例:

int型の初期値 1 に対応する表現は次のとおりになる。

<value>

<intConstant type="int">1</intConstant>

</value>

int型配列の初期値 { 1, 2 } に対応する表現は次のとおりになる。

<value>

<value>

<intConstant type="int">1</intConstant>

<intConstant type="int">2</intConstant>

</value>

</value>

# typeTable要素とデータ型定義要素

typeTable要素は、翻訳単位（2章）に対して一つだけ存在し、翻訳単位で使われているすべてのデータ型についての情報を定義する。

<typeTable>

[ データ型定義要素

… ]

</typeTable>

属性（optional）: なし

typeTable要素は、翻訳単位を表現するXcodeProgram要素（2章）の直接の子要素であり、データ型を定義するデータ型定義要素の列からなる。データ型定義要素には以下の要素がある。

* basicType要素(3.4節)
* pointerType要素(3.5節)
* functionType要素（3.6節）
* arrayType要素（3.7節）
* structType要素とunionType要素（3.8節）（廃止予定）
* class要素（3.9節）
* enumType要素（3.10節）
* typeInstance要素（9.2節）
* classTemplate要素（8.2節）
* aliasTemplate要素（8.4節）

すべてのデータ型定義要素は、型識別名（3.1節）を表すtype要素をもつ。

データ型定義要素は、データ型定義要素属性（1.1節）をもつことができる。

要検討：decltype対応

decltype(式) は式の型を表すが、式はスコープをもつのでtypeTableの中に移動することができない。

* 案1： 式の中のすべての名前に、スコープ名を付ける。decltype(main:x + main:y) など。🡪とても煩雑。scopenameを持たない { } の中に出現した場合は？
* 案2： typeTableを翻訳単位に一つにするのではなく、スコープ毎にもつようにする。
* 案3： decltypeが出現したスコープに限り、typeTableをもつ。

　Clang ASTでは「型推論の結果の型」をAST上に保持しているので、その仕組みに合わせて考えるのであれば、すべて「解決結果の型」を扱えばよいことになる。これも「正規化」の一種と考えて扱うのがよいかもしれない。

## データ型識別名

プログラム内において、データ型はデータ型識別名で区別される。その名前は、次のいずれかである。

* 基本データ型（3.4節）
* C, C++の基本データ型（C++拡張）

'void', 'char', 'short', 'int' , 'long', 'long\_long', 'unsigned\_char', 'unsigned\_short', 'unsigned', 'unsigned\_long', 'unsigned\_long\_long', 'float', 'double', 'long\_double', 'wchar\_t', ‘char16\_t’, ‘char32\_t’, 'bool' (\_Bool型)

* \_Complex、\_Imaginaryに対応する型

'float\_complex', 'double\_complex', 'long\_double\_complex', 'float\_imaginary', 'double\_imaginary', 'long\_double\_imaginary'

* GCCの組み込み型

'\_\_builtin\_va\_arg'

* 型推論の対象（C++）

'auto'

* 名前空間の名前（C++）

'namespace'

* 型の抽象（C++） —　テンプレートの型仮引数の型の名前

'any\_class', 'any\_typename'

* 派生データ型とクラス

他のデータ型識別名とは異なる、翻訳単位内でユニークな英数字の並び。

## typeName要素

typeName要素はデータ型識別名を表す。

<typeName/>

属性（必須）: ref

属性（optional）: access

以下の属性をもつことができる。

* ref属性　—　データ型識別名を示す。
* access属性　—　inheritedFrom要素の子要素のときだけ使用する。public, privateまたはprotecdedのいずれかの値をとる。

typeName要素は以下のように使用される。

* 型を引数とする関数の呼出しで
  + sizeOfExpr（7.11節）、gccAlignOfExpr（7.11節）、builtin\_op（11.5節）
* テンプレートの定義の型仮引数として（8章）
* テンプレートのインスタンスの型実引数として（9章）
* 構造体とクラスの継承元（3.9.1項）

例： 式　sizeof(int) は以下のように表現される。

<sizeOfExpr>

　　<typeName ref="int"/>

</sizeOfExpr>

## データ型定義要素属性

データ型定義要素は共通に以下の属性を持つことができる。これらをデータ型定義要素属性と呼ぶ。

* is\_const　－　そのデータ型がconst修飾子をもつかどうか
* is\_volatile　－　そのデータ型がvolatile修飾子をもつかどうか
* is\_restrict　－　そのデータ型がrestrict修飾子をもつかどうか
* is\_static　－　そのデータ型がstatic属性をもつかどうか
* reference(C++)　—　属性値がlvalueのとき左辺値参照、rvalueのとき右辺値参照を意味する。属性値がdefaultまたは属性が省略されているとき文脈依存であることを意味するが、lvalueまたはrvalueの値をもつことが望ましい。引数の値渡し（通常の場合）に対しては、defalutとする。
* access（C++）　－　アクセス指定子に対応。"private", "protected"または"public"
* is\_virtual（C++）　—　そのメンバ関数がvirtual属性をもつかどうか。
* is\_userDefined（C++）　—　その演算がユーザ定義によりオーバーロードされているかどうか

”is\_”　で始まる属性の値には、真を意味する1とtrue、および、偽を意味する0とfalseが許される。属性が省略されたとき、偽を意味する。

例: 左辺値参照と右辺値参照

以下の参照（左辺値参照）の宣言があるとき、

int& n\_alias = n\_org;

変数n\_aliasのデータ型識別要素は以下のようになる。

<basicType type="B0" name="int" reference=”lvalue”/>

以下のコンストラクタ（ムーブコンストラクタ）の定義の引数に現れた右辺値参照について、

struct Array {

int \*p, len;

Array( Array&& obj ) : p(obj.p), len(obj.len) {

obj.p = nullptr; obj.len = 0;

}

}

仮引数objのデータ型識別要素は以下のようになる。

<basicType="B1" name="B2" reference=”rvalue”/>

## basicType要素

basicType 要素は、他のデータ型識別要素にデータ型定義要素属性を加えた、新しいデータ型定義要素を定義する。

<basicType/>

属性（必須）: type, name

属性（optional）: alignas, データ型定義要素属性

以下の属性を持つ。

* type　－　この型に与えられたデータ型識別名
* name　－　この型の元になる型のデータ型識別名

備考: 旧仕様と実装の違い

本仕様は、旧仕様とは異なり、実装に合わせた。旧仕様では以下のように定義されていた。

basicType要素は、C,C99の基本データ型を定義する。

実装では、データ型定義要素属性（constなど）を持たない基本データ型に対応するデータ型識別要素は定義されず、type="int" のようにデータ型識別名だけで表現されている。また、基本データ型以外の型（構造型など）に属性を付ける場合に、basicType要素を使用している。

例:

struct {int x; int y;} s;

struct s const \* volatile p;

は次のXcodeMLに変換される。 basicType要素によって、”struct s const”を意味するデータ型識別名B0 が定義されている。

<structType type="S0">

<symbols>

<id type=”int”><name=”x”><./name></id>

<id type=”int”><name=”y”><./name></id>

</symbols>

</structType>

<basicType type="B0" is\_const="1" name="S0"/>

<pointerType type="P0" is\_volatile="1" ref="B0"/>

## pointerType要素

pointerType要素はポインタのデータ型を定義する。

<pointerType/>

属性（必須）: type, ref

属性（optional）: データ型定義要素属性

以下の属性を持つ。

* type　－　この型に与えられたデータ型識別名
* ref　－　このポインタが指すデータのデータ型識別名

pointerType要素は、子要素を持たない。

例:

"int \*" に対応するデータ型定義は以下のようになる。

<pointerType type=”P0123” ref=”int”/>

## functionType要素

funtionType要素は、関数型を定義する。

<functionType>

　　[ params要素（5.3.4節） ]

</functionType>

属性（必須）: type, return\_type

属性（optional）: is\_inline

* type　－　この関数型に与えられたデータ型識別名
* return\_type　－　この関数型が返すデータのデータ型識別名
* is\_inline　－　この関数型がinline型であるかどうかの情報、0 または 1、false または true  
  　　　　　　　省略時は0, falseを意味する。

プロトタイプ宣言がある場合には、引数のXML要素に対応するparams要素を含む。

例:

"double foo(int a,int b)" のfooに対するデータ型は以下のようになる。

<functionType type="F0457" return\_type="double">

<params>

<name type="int">a</name>

<name type="int">b</name>

</params>

</fucntionType>

## arrayType要素

arrayType要素は、配列データ型を定義する。

<arrayType>

[ arraySize要素

… ]

</arrayType>

属性（必須）: type, element\_type

属性（optional）: array\_size, データ型定義要素属性

arrayType要素は以下の属性を持つ。

* type　－　この配列型に与えられたデータ型識別名
* element\_type　－　配列要素のデータ型識別名
* array\_size　－　配列のサイズ（要素数）。array\_sizeと子要素のarraySizeを省略した場合は、サイズ未指定を意味する。array\_size属性は子要素のarraySizeと同時に指定することはできない。

以下の子要素を持つ。

* arraySize　－　配列のサイズ（要素数）を表す式。式要素ひとつを子要素に持つ。   
  サイズを数値で表現できない場合や、可変長配列の場合に指定する。arrayType要素がarraySize要素を持つ場合、array\_size属性の値は"\*"とする。

例:

"int a[10]"のaに対するtype\_entryは以下のようになる。

<arrayType type="A011" element\_type="int" array\_size="10"/>

## unionType要素

union(共用体)データ型は、unionType要素で定義する。

<unionType>

symbols要素

</unionType>

属性（必須）: type

属性（optional）: データ型定義要素属性

unionType要素は以下の属性を持つ。

* type　－　この共用体型のデータ型識別名

unionType要素は、メンバに対する識別子の情報であるsymbols要素を持つ。 構造体・共用体のタグ名がある場合には、スコープに対応するシンボルテーブルに定義されている。

メンバのビットフィールドは、id要素の bit\_field 属性または id要素の子要素 であるbitField 要素に記述する（4.1節）。

## structType要素とclass要素（C++）

構造体およびクラスを表現する。

<structType> *or* <class>

　 [ inheritedFrom要素（3.9.1） ]

symbols要素（4.3節）

[ declarations要素（5.2節） ]

</structType> *or* </class>

属性（必須）: type

属性（optional）: lineno, file, inherited, データ型定義要素属性

以下の子要素をもつ。

* symbols要素　－　メンバ変数名とメンバ関数名のリスト
* declarations要素　－　メンバ変数とメンバ関数の宣言、メンバ関数の定義、ディレクティブ文字列を含む

以下の属性をもつ。

* type（必須）　－　この構造体、共用体またはクラスに与えられたデータ型識別名
* inherited　—　継承元のクラスの名前。name要素の名前と同様に、スコープ名付きの名前にもなれる。

メンバのビットフィールドは、id要素の bit\_field 属性または id要素の子要素 であるbitField要素に記述する（4.1節）。

構造体またはメンバの名前は、同じtype属性をもつid要素で指定する。typedef文またはusing文で指定された別名もまた、同じtype属性をもつid要素で指定する。

要検討：

friend関数の宣言。friend関数はそのクラスのメンバ関数ではない。

例:

以下の構造体宣言

struct {

int x;

int y : 8;

int z : sizeof(int);

};

に対するstructType要素は以下のようになる。この構造体のデータ型識別名はS0と定義された。

<stuctType type="S0">

<symbols>

<id type="int">

<name>x</name>

</id>

<id type="int" bit\_field="8">

<name>y</name>

</id>

<id type="int" bit\_field="\*">

<name>z</name>

<bitField>

<sizeOfExpr>

<typeName ref="int"/>

</sizeOfExpr>

</bitField>

</id>

</symbols>

</structType>

### inheritedFrom要素（C++）

structType要素とclass要素だけの子要素であり、継承元の構造体またはクラスの並びを表現する。

<inheritedFrom>

[ typeName要素（3.2節）

... ]

</inheritedFrom>

属性なし

以下の子要素をもつ。

* typeName要素　－　継承する構造体またはクラスのデータ型識別名を示す。access属性により、public, privateまたはprotectedの区別を指定できる。

## enumType要素

enum型は、enumType要素で定義する。type要素で、メンバの識別子を指定する。

<enumType>

[ name要素 ]

symbols要素

</enumType>

属性（必須）: type

属性（optional）: データ型定義要素属性

次の子要素を持つ。

* symbols要素　－　メンバの識別子を定義する。メンバの値はid子要素のvalue子要素で表す。
* name要素（C++、オプショナル）　—　スコープ付き列挙型のときのスコープ名を定義する。

メンバの識別子は、スコープに対応するシンボルテーブルにクラスmoeとして定義されている。 enumのタグ名がある場合には、スコープに対応するシンボルテーブルに定義されている。

例:

"enum { e1, e2, e3 = 10 } ee; "のeeに対するenumType要素は以下のようになる。

<enumType name="E0">

<symbols>

<id>

<name>e1</name>

</id>

<id>

<name>e2</name>

</id>

<id>

<name>e3</name>

<value><intConstant>10</intConstant></value>

</id>

</symbols>

</enumType>

## parameterPack要素（C++）

可変長引数を表現するための、仮引数の並びに対応する。

<parameterPack/>

属性（必須）: type, element\_type

属性（optional）: データ型定義要素属性

以下の属性を持つ。

* type　－　パックされた型に与えられたデータ型識別名
* elem\_type　－　パックされる個々の型のデータ型識別名

parameterPack要素は、子要素を持たない。

例:

以下の関数テンプレートの定義において、

template<typename T1, typename ... Types>

T1 product(T1 val1, Types ... tail) {

return val1 \* product(tail...);

}

"typename ... Types" に対応するデータ型定義は以下のようになる。

<parameterPack type="K0" ref="typename"/>

# シンボルリスト

## id要素

id要素は、変数名や配列名、関数名、struct/unionのメンバ名、 関数の引数、compound statementの局所変数名を定義する。

<id>

name要素（2.1節）

[ value要素（2.2節） ]

[ bitField要素 ]

[ alignAs要素 ]

</id>

属性（optional）: sclass, fspec, type, bit\_field, align\_as, is\_gccThread, is\_gccExtension

id要素は次の属性を持つことができる。

* sclass属性　－　storage class をあらわし、 'auto', 'param', 'extern', 'extern\_def', 'static', 'register', 'label', 'tagname', 'moe', 'typedef\_name', 'template\_param'（C++、テンプレートの型仮引数名）, 'namespace\_name'（C++）, 'thread\_local'（C++）, 'alias\_name'（C++、using文による別名）のいずれか。
* fspec属性　－　function specifierをあらわし、’inline’, ‘vertual’, ‘explicit’のいずれか。
* 【要検討】storage class specifier以外のdecl-specifierである ‘friend’, ‘constexpr’もここで表現するか？
* type属性　－　識別子のデータ型（）
* bit\_field属性　－　structType、unionTypeとclass要素においてメンバのビットフィールドを数値で指定する。
* align\_as属性　－　structType、unionTypeとclass要素において、メンバのalignmentを数値またはデータ型識別名で指定する。
* is\_gccThread属性　－　GCCの\_\_threadキーワードが指定されているかどうかの情報、0または1、falseまたはtrue。
* is\_gccExtension属性

以下の子要素を持つことができる。

* name要素　－　識別子の名前はname要素で指定する。
* value要素　－　識別子に対応した値はvalue要素で指定する。const変数の初期化（const int a=式）、参照の初期化（int& a=初期化子）、namespaceの別名定義（namespace new = scope1::old）などをここで表現する。  
  要検討： 実装時に再検討。何もかもvalue要素にするのがよいか？
* bitField要素　－　unionTypeとclass要素においてメンバのビットフィールドの値をbit\_field属性の数値として指定できないとき使用する。bitField要素は式を子要素に持つ。bitField要素を使用するとき、bit\_field 属性の値は、"\*" とする。
* alignAs要素　—　structType、unionTypeとclass要素においてメンバのalignmentをalign\_as属性の数値として指定できないとき、alignAs要素の子要素として式の要素で指定する。

識別子が変数などの場合、そのアドレスをvalue要素として持つ。コンパイラで生成される変数などの場合は、なくてもよい。

要確認：

　この１行は実態に合っているのか？　value要素として持つのは自然か？

例:

"int xyz;"の変数xyzに対するシンボルテーブルエントリは以下のようになる。なお、P6e7e0は "int \*"に対するtype\_id。

<id sclass=”extern\_def” type=”int”>

<name>xyz</name>

<value>

<VarAddr type=”P6e7e0”>xyz</varAddr>

</value>

</id>

"int foo()"の関数fooに対するシンボルテーブルエントリは以下のようになる。なお、F6f168は、fooのデータ型に対するtype\_id。P6f1a8は、F6f168 へのポインタのtype\_id。識別子fooは関数へのポインタになることに注意。

<id sclass=”extern\_def” type=”0x6f168”>

<name>foo</name>

<value>

<funcAddr type=”0xfla8”>foo</funcAddr>

</value>

</id>

## globalSymbols要素

大域のスコープを持つ識別子を定義する。

<globalSymbols>

[ id要素（4.1節）

… ]

</globalSymbols>

属性なし

子要素として、大域のスコープを持つ識別子のid要素の並びを持つ。

## symbols要素

局所スコープを持つ識別子を定義する。

<symbols>

[ id要素（4.1節）

… ]

</symbols>

属性なし

子要素として、定義する識別子に対するid要素を持つ。

# globalDeclarations要素とdeclarations要素

## globalDeclarations要素

大域的な（翻訳単位全体をスコープとする）変数、関数などの宣言と定義を行う。

<globalDeclarations>

[ { varDecl要素（5.4節）　*or*

functionDecl要素（5.5節）　*or*

usingDecl要素（5.6節） *or*

functionDefinition要素（5.3節） *or*

functionTemplate要素（8.3節） *or*

text要素（6.21節） }

… ]

</globalDeclarations>

属性なし

以下の子要素を持つ。

* functionDefinition要素　－　関数の定義
* varDecl要素　－　変数の定義
* functionDecl要素　－　関数の宣言
* text要素　－　ディレクティブなど任意のテキストを表す

## declarations要素

compoundStatement（6.2節）、class（3.9節）などをスコープとする変数、関数などの宣言と定義を行う。

<declarations>

[ { varDecl要素（5.4節）　*or*

functionDecl要素（5.5節）　*or*

usingDecl要素（5.6節） *or*

functionDefinition要素（5.3節） *or*

text要素（6.21節） }

… ]

</declarations>

属性なし

以下の子要素を持つ。

* functionDefinition要素　－　関数の定義
* varDecl要素　－　変数の定義
* functionDecl要素　－　関数の宣言
* text要素　－　ディレクティブなど任意のテキストを表す

## functionDefinition要素

関数定義、メンバ関数の定義、コンストラクターの定義、デストラクターの定義、および、演算子オーバーロードの定義を行う。以下のいずれか一つの子要素を持つ。

<functionDefinition>

name要素（2.1節） *or* operator要素（5.3.1） *or* constructor要素（5.3.2） *or* descructor要素（5.3.3）

symbols要素（4.3節）

params要素（5.3.4）

body要素

</functionDefinition>

属性(optional): is\_gccExtension

以下のいずれか一つの子要素を持つ。

* name要素　－　関数またはメンバ関数のときの、関数の名前
* operator要素　—　演算子オーバーロードのときの、演算子の名前
* constructor要素　—　構造体またはクラスのコンストラクタのとき
* destructor要素　—　構造体またはクラスのデストラクタのとき

加えて、以下の子要素をもつ。

* symbols要素　－　パラメータ（仮引数）のシンボルリスト。子要素はid要素の並び。   
  params要素　－　パラメータ（仮引数）の並び
* body要素　－　関数本体。子要素として文（通常はcompoundStatement）を含む。関数に局所的な変数などの宣言は、body要素の中に記述される。body要素内にGCCのネストされた関数を表すfunctionDefinitionを含む場合がある。

以下の属性を持つ

* is\_gccExtension属性

例：

関数の定義

struct sss \*foo(struct sss \*arg1, int nnn)

{

・・・（略）・・・

}

に対し、以下の表現が対応する。

<functionDefinition>

<name>foo</name>

<symbols>

<id type=”P1” sclass=”param”>

<name>arg1</name>

</id>

<id type=”int” sclass=”param”>

<name>nnn</name>

</id>

</symbols>

<params>

<name type=”P1”>arg1</name>

<name type=”int”>nnn</name>

</params>

<body>

<compoundStatement>

…（略）…

</compoundStatement>

</body>

</functionDefinition>

### operator要素（C++）

functionDefinition要素の子要素。演算子オーバーロードを定義するとき、name要素の代わりに指定する。

<operator>演算子名</operator>

属性なし

演算子名には、単項演算要素名（7.11節）、二項演算要素名（7.10節）などの7章で定義される演算子のXML要素の名前、または、ユーザ定義リテラルのアンダースコアで始まる名前を記述する。以下に例示する。

<operator>plusExpr</operator>  
 <operator>\_my\_op</operator>

### constructor要素（C++）

functionDefinition要素の子要素。そのメンバ関数がコンストラクタのとき、name要素の代わりに指定する。

<constructor>

[ { name要素（2.1節）

value要素（2.2節） }

… ]

</constructor>

属性(optional): is\_explicit

name要素とvalue要素の組は初期化構文に対応する。

要検討： コンストラクタのバリエーションに対応し切れていない。

### destructor要素（C++）

functionDefinition要素の子要素。そのメンバ関数がデストラクタであるとき、name要素の代わりに指定する。

<destructor/>

### params要素

関数の引数の並びを指定する。

<params>

[ { name要素（2.1節）

[ value要素（2.2節） ] }

… ]

[ ellipsis ]

</params>

属性なし

　以下の子要素をもつことができる。

* name要素　－　引数の名前に対応するname要素を持つ。引数のデータ型の情報は、name要素のtype属性名と同じtype属性名をもつデータ型定義要素（3章）で表現される。
* value要素　—　paramsが関数またはラムダ関数の仮引数並びで、直前のname要素に対応する仮引数がデフォルト実引数をもつとき、それを表現する。
* ellipsis　－　可変長引数を表す。paramsの最後の子要素に指定可能。

params要素内のname要素は、引数の順序で並んでいなくてはならない。

## varDecl要素

変数の宣言を行う。

<varDecl>

name要素（2.1節）

[ value要素（2.2節） ]

</varDecl>

属性なし

変数宣言を行う識別子の名前をname要素で指定する。 以下の子要素を持つ。

* name要素　－　宣言する変数に対するname要素を持つ。
* value要素　－　初期値を持つ場合、value要素で指定する。配列・構造体の初期値の場合、value要素に複数の式を指定する。

例:

int a[] = { 1, 2 };

<varDecl>

<name>a</name>

<value>

<intConstant type="int">1</intConstant>

<intConstant type="int">2</intConstant>

</value>

</varDecl>

## functionDecl要素

関数宣言を行う。

<functionDecl>

name要素（2.1節）

</functionDecl>

属性なし

以下の子要素を持つ

* name要素　－　関数名を指定する

## usingDecl要素（C++）

C++のusing宣言（using declaration）とusing指示（using directive）に対応する。

<usingDecl>

name要素（2.1節）

</usingDecl>

属性(optional): lineno, file, namespace

以下のようにusing文に対応する。

* using指示 ”using namespace 名前空間名” の形のとき
  + namespace属性の値を1またはtrueとする。
  + 名前空間名をname要素とする。名前空間名にはスコープ名と「::」が含まれることがある。
* using宣言 "using 名前” の形のとき
  + namespace属性を持たないか、値を0またはfalseとする。
  + 名前をname要素とする。名前にはスコープ名と「::」が含まれることがある。
* 別名宣言 "using 別名 = 型”　の形のとき、usingDecl要素では表現されない。typedefと同様、データ型定義要素（3章）で表現される。

# 文の要素

Cの文の構文要素に対応するXML要素である。それぞれのXML要素には、文の元の行番号とファイル名を属性として付加することができる。

* lineno　－　文番号を値として持つ
* file　－　この文が含まれているファイル名

## exprStatement要素

式で表現される文を表す。式の要素（7章）を持つ。

<exprStatement>

式の要素（0章）

</exprStatement>

属性(optional): lineno, file

## compoundStatement要素

複文を表現する。

<compoundStatement>

symbols要素（4.3節）

declarations要素（5.2節）

body要素

</compoundStatement>

属性(optional): lineno, file

以下の子要素を持つ。

* symbols要素　－　このスコープの中で定義されているシンボルリスト
* declarations要素　－　このスコープの中で定義される宣言
* body 要素　－　複文本体。文の要素の並び。

## ifStatement要素

if文を表現する。

<ifStatement>

condition要素

then要素

else要素

</ifStatement>

属性(optional): lineno, file

以下の子要素を持つ。

* condition 要素　－　条件式を子要素として含む
* then要素　－　then部の文を子要素として含む
* else要素　－　else部の文を子要素として含む

## whileStatment要素

while文を表現する。

<whileStatement>

condition要素

body要素

</whileStatement>

属性(optional): lineno, file

以下の子要素を持つ

* condition 要素　－　条件式を子要素として含む
* body 要素　－　本体の文を子要素として含む

## doStatement要素

do文を表現する。

<doStatement>

body要素

condition要素

</doStatement>

属性(optional): lineno, file

以下の子要素を持つ。

* body要素　－　本体を表す、文の要素の並びを含む
* condition要素　－　条件式を表す式の要素を含む

## forStatement要素

for文（従来仕様）を表現する。

<forStatement>

[ init要素 ]

[ condition要素 ]

[ iter要素 ]

body要素

</forStatement>

属性(optional): lineno, file

以下の要素を持つ。

* init要素　－　初期化式または宣言文を要素として含む
* condition要素　－　条件式として式の要素を含む
* iter要素　－　繰り返し式として式の要素を含む
* body要素　－　for文の本体を表す、文の要素の並びを含む。

init要素は、for文の中の初期化式または宣言文を表現する。

<init>

式の要素 or symbols要素

</init>

属性なし

　init要素は、forStatement要素の中だけに現れる。初期化式を意味する式の要素を含むか、または、0個以上の局所変数の宣言を意味するsymbols要素を含む。

## rangeForStatement要素（C++）

C++仕様のfor文

for ( *for-range-declaration* : *expression* ) *statement*

を表現する。

<rangeForStatement>

id要素

range要素

body要素

</rangeForStatement>

属性(optional): lineno, file

以下の子要素を持つ。

* id要素（4.1節）
* range要素　－　配列やコンテナを表す式の要素（7章）を含む。
* body要素　－　for文の本体を表す、文の要素（6章）の並びを含む。

## breakStatement要素

break文を表現する。

<breakStatement/>

属性(optional): lineno, file

## continueStatement要素

continue文を表現する。

<continueStatement/>

属性(optional): lineno, file

## returnStatment要素

return文を表現する。

<returnStatement>

[ 式の要素 ]

</returnStatement>

属性(optional): lineno, file

returnする式を、子要素として持つことができる。

## gotoStatement要素

goto文を表現する。

<gotoStatement>

name要素 or 式の要素

</gotoStatement>

属性(optional): lineno, file

子要素にname要素か式のいずれかを持つ。式はGCCにおいてジャンプ先として指定可能なアドレスの式を表す。

* name要素　－　ラベル名の名前を指定する。
* 式の要素　－　ジャンプ先のアドレス値を指定する。

## tryStatement要素（C++）

try構文を表現する。

<tryStatement>

body要素

</tryStatement>

属性(optional): lineno, file

以下の子要素を持つ。

* body要素　－　本体を表す、文の要素（6章）の並びを含む

## throwStatement要素（C++）

throw文を表現する。

<throwStatement/>

属性(optional): lineno, file

子要素を持たない。

## catchStatement要素（C++）

catch構文を表現する。

<catchStatement>

params要素（5.3.4節）

body要素

</catchStatement>

属性(optional): lineno, file

以下の子要素を持つ。

* params要素　—　内容は1つのname要素または1つのellipsisでなければならない。補足する例外の型を示す。
* body要素　－　本体を表す、文の要素（6章）の並びを含む

## statementLabel要素

goto文のターゲットのラベルを表す。

<statementLabel>

name要素

</statementLabel>

属性なし

ラベル名をname要素として持つ。

* name要素　－　ラベル名の名前を指定する。

## switchStatement要素

switch文を表現する。

<statementLabel>

value要素

body要素

</statementLabel>

属性(optional): lineno, file

以下の子要素を持つ。

* value要素　－　switchする値を表す式の要素（0章）
* body要素　－　switch文の本体を表す文の要素（0章）であり、多くの場合compoundStatement要素（6.2節）となる。caseLabel要素（6.17節）とgccRangedCaseLabel要素（6.18節）とdefaultLabel要素（6.19節）を含むことができる。

## caseLabel要素

switch文のcase文を表す。switch要素の中のbody要素の中のcompoundStatementの中だけに現れることができる。

<caseLabel>

value要素

</caseLabel>

属性(optional): lineno, file

caseの値を子要素としてもつ。

* value要素　－　caseの値を指定する。

## gccRangedCaseLabel要素

gcc拡張のcase文での範囲指定を表す。switch要素の中のbody要素の中のcompoundStatementの中だけに現れることができる。

<gccRangedCaseLabel>

value要素

value要素

</gccRangedCaseLabel>

属性(optional): lineno, file

caseの値を要素としてもつ。

* value要素　－　caseの値の下限値を指定する。
* value要素　－　caseの値の上限値を指定する。

## defaultLabel要素

switch文のdefaultラベルを表す。switch要素の中のbody要素の中のcompoundStatementの中だけに現れることができる。

<defaultLabel/>

属性(optional): lineno, file

## pragma要素

pragma要素は#pragma文を表す。

<pragma>文字列</pragma>

属性(optional): lineno, file

#pragmaに指定する文字列を持つ。

## text要素

text要素は任意のテキストを表し、コンパイラに依存したディレクティブなどの情報を要素として持つために使用する。

<text>文字列</text>

属性(optional): lineno, file

内容に任意の文字列を持つ。この要素は globalDeclarasions にも出現する。

# 式の要素

式の構文要素に対応するXML要素である。式の要素には、本章に記述されたXML要素以外に、以下のものがある。

* functionInstance要素（9.3節）

式の要素には、共通して以下の属性を付加できる。

* type属性　―　式のデータ型情報を取り出すことができる。
* （廃止予定）lvalue属性　―　式が左辺値であることを示す。

**要検討：**

lvalue属性は、式の要素の属性からテータ型定義要素の属性に移動したい。

## 定数の要素

定数は以下のXML要素によって表現する。

<intConstant>10進数または16進数</intConstant>

<longlongConstant>16進数 16進数</longlongConstant>

<floatConstant>浮動小数点数</floatConstant>

<stringConstant>文字列</stringConstant>

<moeConstant>列挙型メンバ名</moeConstant>

<booleanConstant>真偽値</booleanConatant>

<funcAddr>関数名</funcAddr>

属性(必須): type

* intConstant要素　－　整数の値を持つ定数を表す。数値として、十進数もしくは、16進数（0xから始まる）を記述する。type属性には”int”, ”long”, ”unsigned”, ”unsigned\_long”, ”char”と”wchar\_t”が許される。C++ではこれらに加えて、”char16\_t”と”char32\_t”が許される。  
  備考：char16\_tは必ず16ビット、char32\_tは必ず32ビットだが、wchar\_tは環境によって16ビットまたは32ビットであると定義されている。
* longlongConstant要素　－　32ビット16進数(0xから始まる)の２つの数字を空白で区切って記述する。type属性には”long\_long”と”unsigned\_long\_long”が許される。
* floatConstant要素　－　floatまたはdoubleまたはlong doubleの値を持つ定数を表す。浮動小数点数のリテラルを記述する。type属性には”float”, ”double”と”long\_double”が許される。
* stringConstant要素　－　内容にダブルクォーテーションで囲まない文字列を記述する。文字列中の特殊文字はXML（HTML）のルールに従ってクォートされる（’<’は＆lt;に置換されるなど）。type属性には、”char”と”wchar\_t”が許される。C++ではこれらに加えて、"char16\_t”と”char32\_t”が許される。  
  仕様変更：旧仕様ではtype属性を持たず、代わりに以下のように定義されている。
  + 属性に is\_wide="[1|0|true|false]" (省略時0)を持ち、1またはtrueのときwchar\_t型の文字列を表す。
* moeConstant要素　－　enum型の定数を表す。内容にenum定数（列挙型のメンバの名前）を記述する。type属性は列挙型のタイプ名を記述する。
* booleanConstant要素　－　真理値リテラル。falseまたはtrue。type属性は”bool”のみ許される。
* funcAddr要素　－　関数のアドレスを表す。内容に関数名を記述する。type属性は、原則としてその関数のインスタンスの型とするが、翻訳時に不明な場合には別の表現とする。（詳細は実装時に検討する。）

## 変数参照の要素（Var要素、varAddr要素、arrayAddr要素）

変数名への参照を表現する。それぞれ、v, &v, aに対応する（vは配列以外の変数、aは配列変数）。

<Var>変数名</Var>

<varAddr>変数名</varAddr>

<arrayAddr>配列変数名</arrayAddr>

属性(必須): type, scope

* Var要素　－　配列以外の変数を参照する式。内容に変数名を指定する。
* varAddr要素　－　配列以外の変数のアドレスを参照する式。内容に変数名を指定する。
* arrayAddr要素　－　配列の先頭アドレスを参照する式。内容に配列変数名を指定する。

scope属性をつかって、局所変数を区別する。

* scope属性　－　"local", "global", "param"のいずれか

例：

nがint型のとき、nのアドレスの参照 &n は、

<varAddr type=”P0” scope=”local”>n</varAddr>

と表現される。ここでP0は、typeTableの中で

<pointerType type=”P0” ref=”int”/>

などと宣言されている。

例：

aがint型の配列のとき、aの参照、すなわちa[0]のアドレスの参照は、

<arrayAddr type=”A5” scope=”local”>a</varAddr>

と表現される。ここでA5は、typeTableの中で

<arrayType type=”A5” element\_type=”int” array\_size=”3”/>

などと宣言されている。

備考：

aが配列のとき、2015年10月現在のF\_Frontでは &a の参照をaの参照と同様arrayAddrで表現している。これに関連してOmni XMPでは型の不一致によるエラーが出ている（バグレポート439）。

## pointerRef要素

式（ポインタ型）の指示先を表現する。

<pointerRef>

式の参照

</pointerRef>

属性(必須): type

例：

　式 \*var1 （var1はint型へのポインタ）は以下のように表現される。

<pointerRef type=”int”>

<Var type=”P0” scope=”local”>var1</Var>

</pointerRef>

要確認：

　現状（2015年10月）のC\_Frontでは、\*(&var\_name) というパターンのとき

<PointerRef><varAddr>var\_name</varAddr></PointerRef>

でなく

<Var>var\_name</Var>

と表現している。なぜこのパターンに限って簡単化しているのか不明。

## arrayRef要素

配列要素a[i]への参照を表現する。

<arrayRef>

arrayAddr要素

式の要素

</arrayRef>

属性(必須): type

例：

　int a[3]; と宣言されているとき、配列要素 a[i] の参照は、

<arrayRef type="int">

<arrayAddr type="A5”scope="local">a</arrayAddr>

<Var type="int" scope="local">i</Var>

</arrayRef>

のように表現される。配列要素のアドレス &a[i] の参照は、

<addrOfExpr type="P232">

<arrayRef type="int">

<arrayAddr type="A5”scope="local">a</arrayAddr>

<Var type="int" scope="local">i</Var>

</arrayRef>

</addrOfExpr>

のように表現される。ここでP232はint型へのポインタと宣言されている。後者はarrayAddr要素でないことに注意されたい。

## メンバの参照の要素（C++拡張）

構造型、クラス、または共用型のオブジェクトをsとするとき、sのメンバmへの参照s.m、sのメンバmのアドレスの参照&s.m、sのメンバ配列aの要素への参照s.a[i]、および、sのメンバ配列aの要素のアドレスの参照&s.a[i]を、それぞれ以下のように表現する。

<memberRef> *or* <memberAddr> *or* <memberArrayRef> *or* <memberArrayAddr>

　　式の要素

</memberRef> *or* </memberAddr> *or* </memberArrayRef> *or* </memberArrayAddr>

属性(必須): type, member

* memberRef　－　配列以外のメンバを参照する。member属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクトsのint型メンバnへの参照 s.n について、以下のように表現する。

<memberRef type=”int” member=”n”>

<varAddr type=”P0” scope=”local”>s</varAddr>

</memberRef>

* memberAddr　－　配列名以外のメンバのアドレスを参照する。member属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクトsのint型メンバnのアドレス &s.n について、以下のように表現する。

<memberAddr type=”int” member=”n”>

<varAddr type=”P6” scope=”local”>s</varAddr>

</memberAddr>

* memberArrayRef　－　オブジェクトの配列メンバを参照する。member属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクトsのint型配列メンバaへの参照 s.a について、以下のように表現する。

<memberArrayRef type=”A0” member=”a”>

<varAddr type=”P1” scope=”local”>s</varAddr>

</memberArrayRef>

* memberArrayAddr　－　オブジェクトの配列メンバのアドレスを参照する。member属性にメンバ名を指定し、子要素でオブジェクトのアドレスを表現する。例えば、オブジェクトsのint型配列メンバaのアドレス &s.a について、以下のように表現する。

<memberArrayAddr type=”P24” member=”a”>

<varAddr type=”P7” scope=”local”>s</varAddr>

</memberArrayAddr>

　メンバの参照が入れ子になるとき、子要素の表現も入れ子になる。

要検討：構造体まわりの現在のC\_Frontの変換仕様について

arrayRef要素（7.4節）とmemberArrayRef要素、arrayAddr要素（7.2節）とmemberArrayAddr要素は、それぞれ名前が似ているが意味の対称性がない。少なくとも名前を再考したい。他の点でも、今後構造体やクラスへの対応を考えると、整理しておきたいところ。

|  |  |  |  |
| --- | --- | --- | --- |
| C言語表現 | XcodeML表現 | C言語表現 | XcodeML表現 |
| v | Var v | s.v | memberRef v  varAddr s |
| &v | varAddr v | &s.v | memberAddr v  varAddr s |
| a | arrayAddr a | s.a | memberArrayRef a  varAddr s |
| &a | arrayAddr a | &s.a | memberArrayAddr a  varAddr s |
| a[i] | arrayRef  arrayAddr a  Var i | s.a[i] | pointerRef  plusExpr  memberArrayRef a  varAddr s  Var i |
| &a[i] | addrOfExpr  arrayRef  arrayAddr a  Var i | &s.a[i] | plusExpr  memverArrayRef a  varAddr s  Var i |

## メンバポインタの参照の要素（C++）

オブジェクトsのメンバへのポインタの参照s.\*pを表現する。

<memberPointer>

　　式の要素

</memberPointer>

属性(必須): type, name

name属性に変数名を指定し、子要素で構造体のアドレスを表現する。

備考:

メンバの参照（7.5節）ではmember属性にメンバ名を記述するのに対し、メンバポインタの参照（本節）ではname属性に変数名を記述する。この仕様は実装を反映した。

例:

以下のプログラムで、(1)はメンバ変数へのポインタの宣言、(2)はメンバ関数へのポインタの宣言であり、それぞれメンバ変数、メンバ関数をポイントするよう初期化されている（2.1節の例参照）。(3)の右辺によりs1.fooが引数3で呼び出され、左辺s1.dataに代入される。

struct S {

int data;

int foo(int n) { return n + 1; }

};

int S :: \*d = &S :: data; // (1)

int (S :: \*f)(int) = &S :: foo; // (2)

struct S s1;

s1.\*d = (s1.\*f)(3); // (3)

このとき、(3)の左辺は以下のように表現される。

<memberPointerRef type="P4" name=”d”>

<varAddr type="P3" scope="global">s1</varAddr>

</memberPointerRef>

(3)の右辺は以下のように表現される。

<functionCall type="int">

<function>

<memberPointerRef type="P4" name=”f”>

<varAddr type="P3" scope="global">s1</varAddr>

</memberPointerRef>

</function>

<arguments>

<intConstant type="int">3</intConstant>

</arguments>

</functionCall>

## 複合リテラルの要素（新規）

型Tの複合リテラル (T){ … } および型Tの複合リテラルのアドレス &(T){ … } を表現する。

<compoundValue> *or* <compoundValueAddr>

　　value要素

</compoundValue> *or* </compoundValueAddr>

属性(必須): type

指示付きの初期化子（(T){ [2]=1, .x=2 } のような記述）に対応する表現は持たず、常に展開された表現に変換される（例参照）。

備考：

複合リテラルは、旧仕様書ではcastExpr要素で表現すると書かれているが、C\_Frontの動作と食い違っている。本節はC\_Frontの動作に合わせて書き起こした。

例：

　以下のようなプログラムで、

typedef struct { int x, y; } two\_int\_t;

int n = 20;

foo(&(two\_int\_t){ 1, n });

goo((two\_int\_t){ .y=300 });

関数fooの引数は以下の表現となる。

<compoundValueAddr type="P6">

<value>

<value>

<intConstant type="int">1</intConstant>

<Var type="int" scope="local">n</Var>

</value>

</value>

</compoundValueAddr>

関数gooの引数は以下の表現となる。

<compoundValue type="P6">

<value>

<value>

<intConstant type="int">0</intConstant>

<intConstant type="int">300</intConstant>

</value>

</value>

</compoundValue>

## thisExpr要素（C++）

thisExpr 要素は、C++の this に対応する。

<thisExpr/>

属性なし

## assignExpr 要素

assignExpr 要素は、２つの式の要素をsub要素に持ち、代入を表す。

<assignExpr>

式の要素

式の要素

</assignExpr>

属性(必須): type

属性（optional）: is\_userDefined

第１の式を左辺、第２の式を右辺とする代入文を表現する。

## 2項演算式の要素

二項演算式を表現する。被演算子の２つのXML要素を内容に指定する。

<二項演算要素名>

式の要素

式の要素

</二項演算要素名>

属性(必須): type

属性（optional）: is\_userDefined

二項演算要素名には以下のものがある。

* 算術二項演算子
  + plusExpr　－　加算
  + minusExpr　－　減算
  + mulExpr　－　乗算
  + divExpr　－　除算
  + modExpr　－　剰余
  + LshiftExpr　－　左シフト
  + RshiftExpr　－　右シフト
  + bitAndExpr　－　ビット論理積
  + bitOrExpr　－　ビット論理和
  + bitXorExpr　－　ビット論理　排他和
* 代入演算子
  + asgPlusExpr　－　加算
  + asgMinusExpr　－　減算
  + asgMulExpr　－　乗算
  + asgDivExpr　－　除算
  + asgModExpr　－　剰余
  + asgLshiftExpr　－　左シフト
  + asgRshiftExpr　－　右シフト
  + asgBitAndExpr　－　ビット論理積
  + asgBitOrExpr　－　ビット論理和
  + asgBitXorExpr　－　ビット論理　排他和
* 論理二項演算子
  + ogEQExpr　－　等価
  + logNEQExpr　－　非等価
  + logGEExpr　－　大なり、または同値
  + logGTExpr　－　大なり
  + logLEExpr　－　小なり、または等価
  + logLTExpr　－　小なり
  + logAndExpr　－　論理積
  + logOrExpr　－　論理和

備考：

　Cでは代入演算の第１オペランドは必ずlvalue（左辺式）だったが、C++では演算子のオーバーロードがあるためその限りではなくなった。

## 単項演算式の要素

単項演算式を表現する。被演算子を内容に指定する。

<単項演算要素名>

式の要素

</単項演算要素名>

属性(必須): type

属性（optional）: is\_userDefined

単項演算要素名には以下のものがある。

* 算術単項演算子
  + unaryMinusExpr　－　符号反転
  + bitNotExpr　－　ビット反転
* 論理単項演算子
  + logNotExpr　－　論理否定
* sizeof演算子
  + sizeOfExpr　－　子要素として式の要素またはtypeName要素を指定する。
* alignof演算子（C++）
  + alignOfExpr　－　子要素として式の要素またはtypeName要素を指定する。
* typeid演算子（C++）
  + typeidExpr　－　子要素として式の要素またはtypeName要素を指定する。
* GCC拡張の演算子
  + gccAlignOfExpr　－　GCCの\_\_alignof\_\_演算子を表す。子要素に式またはtypeName要素を指定する。
  + gccLabelAddr　－　GCCの&&単項演算子を表す。内容にラベル名を指定する。

## functionCall要素

functionCall要素は関数呼び出しを表す。

<functionCall>

<function>

式の要素

</function>

arguments要素（7.12.1項）

</functionCall>

属性(必須): type

function要素には呼び出す関数のアドレスを指定する。

arguments要素には引数の並びを指定する。

### arguments要素

実引数（actual argument）の0個以上の並びを表現する。

<arguments>

[ 式の要素

... ]

</arguments>

## commaExpr要素

コンマ式（第１オペランドと第２オペランドを評価し、第２オペランドの値を返す式）を表す。

<commaExpr>

式の要素

式の要素

</commaExpr>

属性(必須): type

属性（optional）: is\_userDefined

## インクリメント・デクリメント要素（postIncrExpr, postDecrExpr, preIncrExpr, preDecrExpr）

postIncrExpr要素、postDecrExpr要素は、CおよびC++のポストインクリメント、デクリメント式を表す。preIncrExpr要素、preDecrExpr要素は、CおよびC++のプレインクリメント、デクリメント式を表す。

<postIncrExpr> *or* <postDeclExpr> *or* <preIncrExpr> *or* <preDecrExpr>

式の要素

</postIncrExpr> *or* </postDeclExpr> *or* </preIncrExpr> *or* </preDecrExpr >

属性(必須): type

属性（optional）: is\_userDefined

## castExpr要素（廃止予定）

castExpr要素は型変換の式（旧仕様）、または複合リテラルを表す。

<castExpr>

式の要素 *or* value要素

</castExpr>

属性(必須): type

属性（optional）: is\_gccExtension

以下の子要素を持つ。

* castされる式、または、複合リテラルのリテラル部

備考：

現在のC\_Frontでは、複合リテラルにこの表現は使われておらず、compoundValue要素またはcompoundValueAddr要素（7.7節）が使われている。キャストはC++仕様のstatic\_cast, const\_castまたはreinterpret\_castに変換して表現する方が、バリエーションの削減になるため望ましい。どちらの用途にも使われないのであれば、castExprは廃止すべきと考える。

## キャスト要素（staticCast, dynamicCast, constCast, reinterpretCast）（C++）

順に、C++のstatic\_cast, dynamic\_cast, const\_castおよびreinterpret\_castを表現する。

<staticCast> *or* <dynamicCast> *or* <constCast> *or* <reinterpretCast>

式の要素

</staticCast> *or* </dynamicCast> *or* </constCast> *or* </reinterpretCast>

属性(必須): type

Cのcastは、staticCastまたはconstCastまたはreinterpretCastで表現する。

## condExpr要素

三項演算 x ? y : z を表現する。

<condExpr>

式の要素

[ 式の要素 ]

式の要素

</condExpr>

属性(必須): type

第２オペランド（２番目の式）は省略されることがある（GNU拡張対応）。

要確認:

選択される式のtype属性が異なるとき、condExpr要素のtype属性はどうするべきか？

## gccCompoundExpr要素

gcc拡張の複文式に対応する。

<gccCompoundExpr>

compoundStatement要素

</gccCompoundExpr>

属性(必須): type

* compoundStatement要素　－　複文式の内容を指定する。

## newExpr要素とnewExprArray要素

new演算子またはnew[]演算子から成る式を表現する。

<newExpr>

arguments要素（7.12.1項）

</newExpr>

属性(必須): type

<newArrayExpr>

式の要素（7章）

</newArrrayExpr>

属性(必須): type

第１と第２の書式は、それぞれnew演算子とnew[]演算子による領域確保を表現する。確保されるデータは、type属性の型をもつ。第１の書式の子要素は、コンストラクタに渡されるパラメタを表す。第２の書式の子要素は、確保する配列の長さを表す。

## deleteExpr要素とdeleteArrayExpr要素

delete演算子またはdelete[]演算子から成る式を表現する。

<deleteExpr>

式の要素

</deleteExpr>

属性(必須): type

<deleteArrayExpr>

式の要素

</deleteArrayExpr>

属性(必須): type

第１と第２の書式は、それぞれdelete演算子とdelete[]演算子による領域解放を表現する。子要素は、解放する領域へのポインタである。

## lambdaExpr要素

C＋＋のラムダ式を表現する。

<lambdaExpr>

captures要素

symbols要素

params要素

body要素

</lambdaExpr>

属性(必須): type

symbols要素、params要素（5.3.4節）とbody要素は、functionDefinition要素（5.2節）の子要素と同様である。

### captures要素

　captures要素は以下の表現である。

<captures>

<byReference>

[ name要素

… ]

</byReference>

<byValue>

[ name要素

… ]

</byValue>

</captures>

属性(optional): default, is\_mutable

captures要素はオプショナルに以下の属性をもつ。

* default属性　－　“by\_reference” のとき、スコープデフォルトが参照キャプチャ “[&]” であることを意味し、”by\_value”のときデフォルトがコピーキャプチャ “[=]” であることを意味する。省略されたとき、キャプチャがないことを意味する。
* is\_mutable属性　－　1またはtrueのとき、mutable指定があることを意味する。0またはfalseまたは省略されたとき、mutable指定がないことを意味する。

　子要素のbyReference要素で指定された名前の変数は参照キャプチャされ、byValue要素で指定された名前の変数はコピーキャプチャされる。それ以外の変数は、default属性の指定に従う。

# テンプレート定義要素（C++）

テンプレート定義要素には、以下のものがある。

* structTemplate要素（8.2節）　—　構造型のテンプレートを定義する。
* classTemplate要素（8.2節）　—　クラステンプレートを定義する。
* funcitionTemplate要素（8.3節）　—　関数、メンバ関数、演算子オーバーロード、および、ユーザ定義リテラルのテンプレートを定義する。
* aliasTemplate要素（8.4節）　—　型のエイリアスのテンプレートを定義する。

これらのテンプレート定義要素は、共通して型仮引数を表現するtypeParams要素（8.1節）をもつ。

## typeParams要素

テンプレートの型仮引数の並びを指定する。

<typeParams>

[ { typeName要素（3.2節）

[ <value>

typeName要素（3.2節）

</value> ] }

… ]

</typeParams>

属性なし

　以下の子要素をもつことができる。

* typeName要素　－　型仮引数に対応するデータ型識別名を表現する。
* value要素　—　子要素としてtypeName要素をもつ。関数テンプレートまたはメンバ関数テンプレートにおいて、直前のtypeName要素に対応する仮引数がデフォルト実引数をもつとき、それを表現する。

typeName要素は、引数の順序で並んでいなければならない。

## structTemplate要素とclassTemplate要素

データ型定義要素（3章）の一つ。構造体とクラスのテンプレートをそれぞれ以下のように表現する。

<structTemplate>

symbols要素（4.3節）

typeParams要素（8.1節）

structType要素（3.9節）

</structTemplate>

属性(optional): lineno, file

<classTemplate>

symbols要素（4.3節）

typeParams要素（8.1節）

class要素（3.9節）

</classTemplate>

属性(optional): lineno, file

以下の子要素をもつ。

* symbols要素　—　型仮引数に関するid要素を子要素として持つ。
* typeParams要素　—　子要素としてtypeName要素の並びを持つ。
* structType要素またはclass要素　—　構造体またはクラスの定義

例:

　以下のプログラムは、

template <typename T>

struct pair { T val1, val2; };

以下のように表現される。

<structTemplate>

<symbols>

<id type=”S0” sclass=”template\_param”>

<name>T</name>

</id>

</symbols>

<typeParams>

<typeName ref=”S0”>

</typeParams>

<structType type="S1">

<symbols>

<id type="S0">

<name>val1</name>

</id>

<id type="S0">

<name>val2</name>

</id>

</symbols>

</structType>

</structTemplate>

ここで、データ型識別名S0はtypeTableの中で以下のように定義されている。

<basicType type="S0" name=”any\_typename”/>

## functionTemplate要素

関数、メンバ関数、演算子オーバーロード、および、ユーザ定義リテラルのテンプレートを表現する。globalDeclaration要素（5.1節）とdeclaration要素（5.2節）の子要素。

<functionTemplate>

symbols要素（4.3節）

typeParams要素（8.1節）

functionDefinition要素（5.3節）

</functionTemplate>

属性(optional): lineno, file

以下の子要素をもつ。

* symbols要素　—　型仮引数に関するid要素を子要素として持つ。
* typeParams要素　—　子要素としてtypeName要素の並びを持つ。
* functionDefinition要素　—　関数の定義

例：

以下の関数テンプレートについて、

template <class T>

T square(const T& x) { return x \* x; }

型仮引数Tに対するデータ型識別名X0と、仮引数xの型const T&に対するデータ型識別名X1は、typeTableの中で以下のように定義される。

<basicType type="X0" name=”any\_class”/>

<basicType type=”X1” is\_const=”1” is\_lvalue=”1” name=”X0”/>

そして、関数テンプレートは以下のように定義される。

<functionTemplate>

<symbols>

<id type=”X0” sclass=”template\_param”>

<name>T</name>

</id>

</symbols>

<typeParams>

<typeName ref=”X0”>

</typeParams>

<functionDefinition>

<name>square</name>

<symbols>

<id type=”X1” sclass=”param”>

<name>x</name>

</id>

</symbols>

<typeParams>

<name type=”X1”>x</name>

</typeParams>

<body>

・・・（略）・・・

</body>

</functionDefinition>

</functionTemplate>

## aliasTemplate要素

データ型定義要素（3章）の一つ。using文による型の別名定義に対するテンプレートを表現する。

<aliasTemplate>

symbols要素（4.3節）

typeParams要素（8.1節）

</aliasTemplate>

属性(必須): type, name

属性(optional): lineno, file

以下の属性を持つ。

* type　－　型の別名に与えられるデータ型識別名
* name　－　この型の元になる型のデータ型識別名

要検討：

属性名はrefがよいかnameがよいか。

以下の子要素を持つ。

* symbols要素　—　型仮引数に関するid要素を子要素として持つ。
* typeParams要素　—　子要素としてtypeName要素の並びだけを持つことができる。

定義される別名は、このtype属性値と同じtype属性値をもつid要素で表現され、そのスコープのシンボルテーブル（globalSymbolsまたはsymbols）に登録される。

例：

以下の別名テンプレートについて、

template <typename T>

using myMap = std::map<int, T&>;

型仮引数Tに対するデータ型識別名X0と、std::mapの第２型引数T&　に対するデータ型識別名X0は、typeTableの中で以下のように定義される。

<basicType type="X0" name=”any\_typename”/>

<basicType type=”X1” is\_lvalue=”1” name=”X0”/>

そして、別名テンプレートは以下のように定義される。ここで、T0は他で定義されているstd::mapのデータ型識別名であり、std::mapのid要素のtype属性と一致している。T1はこのaliasTemplateで定義されたデータ型識別名であり、myMapのid要素のtype属性と一致している。

<aliasTemplate type=”T1” name=”T0”>

<symbols>

<id type=”X0” sclass=”template\_param”>

<name>T</name>

</id>

</symbols>

<typeParams>

<typeName ref=”X0”>

</typeParams>

</aliasTemplate>

# テンプレートインスタンス要素（C++）

テンプレートインスタンス要素には、以下のものがある。

* typeInstance要素（9.2節）　—　構造型、クラス、および型の別名のテンプレートについて、型実引数を与えて具体化する。
* funcitionInstance要素（9.3節）　—　関数、メンバ関数、演算子オーバーロード、および、ユーザ定義リテラルのテンプレートについて、型実引数を与えて具体化する。

これらのテンプレートインスタンス要素は、共通して型実引数を表現するtypeArtuments要素（9.1節）をもつ。

## typeArguments要素

テンプレートのインスタンスの型実引数の並びを指定する。

<typeArguments>

[ typeName要素（3.2節）

… ]

</typeArguments>

属性なし

　以下の子要素をもつことができる。

* typeName要素　－　型実引数に対応するデータ型識別名を表現する。

typeName要素は、引数の順序で並んでいなければならない。

## typeInstance要素

データ型定義要素（3章）の一つ。型のテンプレートのインスタンスを表現する。

<typeInstance>

typeArguments要素（9.1節）

</typeInstance>

属性(optional): type, ref

以下の属性を持つ。

* type属性　—　typeInstance要素のデータ型識別名、すなわち表現されたインスタンスの型
* ref属性　—　対応するテンプレートのデータ型識別名

要検討：

属性名はrefがよいかnameがよいか。

例：

構造型のテンプレート

template <typename T1, typename T2, typename T3>

struct Triple {

・・・（略）・・・

};

のデータ型識別名をTRI0とするとき、そのインスタンス

triple<int, int\*, int\*\*>

のデータ型識別名TRI1は、以下のように表現される。

<typeInstance type=”TRI1” ref=”TRI0”>

<typeParameters>

<typeName ref=”int”>

<typeName ref=”P0”>

<typeName ref=”P1”>

</typeParameters>

</typeInstance>

ここで、P0とP1は、以下のように定義されているデータ型識別名である。

<pointerType type=”P0” ref=”int” />

<pointerType type=”P2” ref=”int” />

<pointerType type=”P1” ref=”P2” />

## functionInstance要素

式の要素（7章）の一つ。関数とメンバ関数のテンプレートのインスタンスを表現する。

<functionInstance>

typeArguments要素（9.1節）

functionCall要素（7.12節）

</functionInstance>

属性(必須): type, name

　型実引数の並びは、スペースで区切られたデータ型識別名で表現する。

例：

functionTemplate要素（8.3節）の例において、関数テンプレート

template <class T>

T square(const T& x) { return x \* x; }

のTに対するデータ型識別名は以下のX0、仮引数xの型const T&に対するデータ型識別名は以下のX1である。

<basicType type="X0" name=”any\_class”/>

<basicType type=”X1” is\_const=”1” is\_lvalue=”1” name=”X0”/>

ここで、このテンプレート関数の参照

square<int>(10)

は、以下のように表現される。

<functionInstance>

<typeArguments>

<typeName ref=”int”/>

</typeArguments>

<functionCall type=”X0”>

<function>

<funcAddr type=”P0”>square</funcAddr>

</function>

<arguments>

　　 <intConstant type=”init”>10</intConstant>

</arguments>

</functionCall>

<functionInstance>

ここで、funcAddr要素のtype属性P0は、関数squareへのポインタを意味するデータ型識別名である。

# XcalableMP固有の要素

備考：

C++対応版作成に当たって再検討していない。

## coArrayType要素

"#pragma xmp coarray" によって宣言された、Co-Array型を表す。 次の属性を持つ。

* type　－　派生データ型名。
* element\_type　－　Co-Arrayの要素のデータ型名。データ型名に対応する型がcoArrayTypeのときは、２次元以上のCo-Array型を表す。
* array\_size　－　Co-Array次元を表す。

次の子要素を持つ。

* arraySize　－　Co-Array次元を表す。arraySize要素を持つときの array\_size 属性の値は "\*" とする。

例:

int A[10];

#pragma xmp coarray [\*][2]::A

上記の変数Aの型を表すXML要素は、次のcoArrayType "C2"になる。

<arrayType type="A1" element\_type="int" array\_size="10"/>

<coArrayType type="C1" element\_type="A1"/>

<coArrayType type="C2" element\_type="C1" array\_size="2"/>

## coArrayRef要素

Co-Array型の変数への参照を表す。

次の子要素を持つ。

* 1番目の式　－　Co-Array変数を表す式。
* 2番目以降の式　－　Co-Array次元を表す式。複数の次元を持つ場合は、複数の式を指定する。

## subArrayRef要素

部分配列の参照を表す。

次の子要素を持つ。子要素を省略することはできない。

* 第一のXML要素として配列を表す式をもつ。
* 2番目以降の式　－　添字または添字3つ組を表す式。複数の次元を持つ場合は、複数の式を指定する。

## indexRange要素

3つ組(triplet)を表す。

次の子要素を持つ。子要素を省略することはできない。

* lowerBound　－　下限のインデックスを表す。子要素に式を持つ。
* upperBound　－　上限のインデックスを表す。子要素に式を持つ。
* step　－　インデックスの刻み幅を表す。子要素に式を持つ。

# その他の要素・属性

## 

## is\_gccExtension属性

is\_gccExtension属性は、GCCの \_\_extension\_\_ キーワードをXML要素の先頭に付加するかどうかを定義し、値は 0 または 1 (falseまたはtrue) である。is\_gccExtension属性は省略可能で、指定しないときは値 0を指定したときと同じ意味である。次のXML要素に is\_gccExtension 属性を持つことができる。

* id
* functionDefinition
* castExpr
* gccAsmDefinition

例:

"\_\_extension\_\_ typedef long long int64\_t" に対応する定義は次のようになる。

<id type="long\_long" sclass="typedef" is\_gccExtension="1">

<name>int64\_t</name>

</id>

## gccAsm要素、gccAsmDefinition要素、gccAsmStatement要素

gccAsm 要素・gccAsmDefinition要素・gccAsmStatement要素は、GCCの asm/\_\_asm\_\_ キーワードを定義する。子要素として asm の引数の文字列を持つ。

* gccAsm　－　asm式を表す。次の子要素を持つ。
* stringConstant (1個)　－　アセンブラコードを表す。
* gccAsmDefinition　－　asm定義を表す。子要素はgccAsmと同じ。
* gccAsmStatement　－　asm文を表す。

次の属性を持つ。

* is\_volatile　－　volatile が指定されているかどうかの情報、0または1、falseまたはtrue。

次の子要素を持つ。

* stringConstant (1個)　－　アセンブラコードを表す。
* gccAsmOperands (2個)　－　1番目が出力オペランド、2番目が入力オペランドを表す。オペランドを省略する場合は、子要素を持たないタグを記述する。子要素にgccAsmOperand(複数)を持つ。
* gccAsmClobbers (0-1個)　－　クロバーを表す。子要素に0個以上の stringConstant を持つ。
* gccAsmOperand　－　入出力オペランドを表す。

次の属性を持つ。

* match (省略可) 　－　matching constraintの代わりに指定する識別子を表す（"[識別子]" に対応）。
* constraint (省略不可) 　－　constraint/constraint modifierを表す。

次の子要素を持つ。

* 式 (1個)　－　入力または出力に指定する式を表す。

例:

asm volatile (

"661:\n"

"\tmovl %0, %1\n662:\n"

".section .altinstructions,\"a\"\n"

".byte %c[feat]\n"

".previous\n"

".section .altinstr\_replacement,\"ax\"\n"

"663:\n"

"\txchgl %0, %1\n"

: "=r" (v), "=m" (\*addr)

: [feat] "i" (115), "0" (v), "m" (\*addr));

<gccAsmStatement is\_volatile="1">

<stringConstant><![CDATA[661:\n\tmovl .. (省略) ..]]></stringConstant>

<gccAsmOperands>

<gccAsmOperand constraint="=r">

<Var>v</Var>

</gccAsmOperand>

<gccAsmOperand constraint="=m">

<pointerRef><Var>addr</Var></pointerRef>

</gccAsmOperand>

</gccAsmOperands>

<gccAsmOperands>

<gccAsmOperand match="feat" constraint="i">

<intConstant>115</intConstant>

</gccAsmOperand>

<gccAsmOperand constraint="m">

<pointerRef><Var>addr</Var></pointerRef>

</gccAsmOperand>

</gccAsmOperands>

</gccAsmStatement>

## gccAttributes要素

gccAttributes 要素はGCCの \_\_attribute\_\_ キーワードを定義する。子要素として、\_\_attribute\_\_の引数の文字列を持つ。gccAttributes 要素は、gccAttribute 要素を子要素に複数持つ。

* 型を表すXML要素全てが gccAttributes 要素を子要素に持つ（0～1個）。
* id 要素が gccAttributes 要素を子要素に持つ（0～1個）。
* functionDefinition 要素が gccAttributes 要素を子要素に持つ（0～1個）。

例:

型を表すXML要素の子要素に、gccAttributes を設定する例

typedef \_\_attribute\_\_((aligned(8))) int ia8\_t;

ia8\_t \_\_attribute\_\_((aligned(16)) n;

<typeTable>

<basicType type="B0" name="int" align="8" size="4"/>

<gccAttributes>

<attribute>aligned(8)</attribute>

</gccAttributes>

</basicType>

<basicType type="B1" name="int" align="16" size="4"/>

<gccAttributes>

<attribute>aligned(8)</attribute>

<attribute>aligned(16)</attribute>

</gccAttributes>

</basicType>

</typeTable>

<globalSymbols>

<id type="B0" sclass="typedef\_name">

<name>ia8\_t</name>

</id>

<id type="B1">

<name>n</name>

</id>

</globalSymbols>

<globalDeclarations>

<varDecl>

<name>n</name>

</varDecl>

</globalDeclarations>

id 要素、functionDefinition 要素の子要素に、gccAttributes を設定する例

void func(void);

void func2(void) \_\_attribute\_\_(alias("func"));

void \_\_attribute\_\_((noreturn)) func() {

...

}

<typeTable>

<functionType type="F0">

<params>

<name type="void"/>

</params>

</functionType>

<functionType type="F1">

<params>

<name type="void"/>

</params>

</functionType>

</typeTable>

<globalSymbols>

<id type="F0" sclass="extern\_def">

<name>func</name>

</id>

<id type="F1" sclass="extern\_def">

<name>func2</name>

<gccgccAttributes>

<gccAttribute>alias("func")</gccAttribute>

</gccgccAttributes>

</id>

</globalSymbols>

<globalDeclarations>

<functionDefinition>

<name>func</name>

<gccgccAttributes>

<gccAttribute>noreturn</gccAttribute>

</gccgccAttributes>

<body>...</body>

</functionDefinition>

</globalDeclarations>

## builtin\_op要素

builtin\_op要素はコンパイラ組み込みの関数呼び出しを表す。以下のXML要素をそれぞれ0～複数持つ。子要素の順番は関数引数の順番と一致していなければならない。

* 式　－　呼び出す関数の引数として、式を指定する。
* typeName　－　呼び出す関数の引数として、型名を指定する。
* gccMemberDesignator　－　呼び出す関数の引数として、構造体・共用体のメンバ指示子を指定する。属性に構造体・共用体の派生データ型名を示す ref、メンバ指示子の文字列を示す member を持つ。子要素に配列インデックスを表す式(0-1個)と、gccMemberDesignator要素(0-1個)を持つ。

## is\_gccSyntax属性

is\_gccSyntax属性はそのタグに対応する式、文、宣言がgcc拡張を使用しているかどうかを定義する。 値として0 または 1 (falseまたはtrue) を持つ。この属性は省略可能であり、省略された場合は値に0を指定した時と同じ意味になる。

## is\_modified属性

is\_modified属性はそのタグに対応する式、文、宣言がコンパイルの過程で変形されたかどうかを定義する。値として0 または 1 (falseまたはtrue) を持つ。この属性は省略可能であり、省略された場合は値に0を指定した時と同じ意味になる。

次のXML要素に is\_gccSyntax 属性、is\_modified 属性を持つことができる。

* varDecl
* 文の要素
* 式の要素

# 未検討項目

以下の項目については、本ドキュメントで触れていない。

* 宣言
  + asm ( … )
  + 結合指定 extern “C” double x;
* クラス
  + 部分特殊化
  + final
  + 純粋仮想関数、純粋指定子（＝０）
* 例外処理
  + noexceptキーワード
* 属性
  + [[ noreturn ]] [[ carries\_dependency ]]　[[ deprecated ]]

# コード例

例1:

int a[10];

int xyz;

struct { int x; int y;} S;

foo() {

int \*p;

p = &xyz; /\* 文1 \*/

a[4] = S.y; /\* 文2 \*/

}

文1:

<exprStatement>

<assignExpr type=" P6fc98">

　　<ponterRef type=" P6fc98">

<varAddr scope="local" type="P70768">p</varAddr>

</pointerRef>

<varAddr type=" P70828">xyz</varAddr>

</assignExpr>

</exprStatement>

もしくは、

<exprStatement>

<assignExpr type=" P6fc98">

　　<Var scope="local" type=" P6fc98">p</Var>

<varAddr type=" P70828">xyz</varAddr>

</assignExpr>

</exprStatement>

文2:

<exprStatement>

<assignExpr type="int">

<pointerRef type="int">

<plusExpr type=" P6fc98">

<arrayAddr type=" P708e8">a</arrayAddr>

<intConstant type="int">4</intConstant>

</plusExpr>

</pointerRef>

<pointerRef type="int">

<memberAddr type="P0dede" member="y">

<varAddr type= "P70988">S</varAddr>

</memberAddr>

</pointerRef>

</assignExpr>

</exprStatement>

もしくは、

<exprStatement>

<assignExpr type="int">

<pointerRef type="int">

<plusExpr type=" P6fc98">

<arrayAddr type=" P708e8">a</arrayAddr>

<intConstant type="int">4</intConstant>

</plusExpr>

</pointerRef>

<memberRef type="int" member="y">

<varAddr type= "P70988">S</varAddr>

</memberRef>

</assignExpr>

</exprStatement>

<?xml version="1.0" encoding="ISO-8859-1"?>

<XcodeProgram source="t3.c">

<!--

typedef struct complex {

double real;

double img;

} complex\_t;

complex\_t x;

complex\_t complex\_add(complex\_t x, double y);

main()

{

complex\_t z;

x.real = 1.0;

x.img = 2.0;

z = complex\_add(x,1.0);

printf("z=(%f,%f)\n",z.real,z.img);

}

complex\_t complex\_add(complex\_t x, double y)

{

x.real += y;

return x;

}

-->

<typeTable>

<pointerType type="P0" ref="S0"/>

<pointerType type="P1" ref="S0"/>

<pointerType type="P2" ref="S0"/>

<pointerType type="P3" ref="S0"/>

<pointerType type="P4" ref="S0"/>

<pointerType type="P5" ref="F0"/>

<pointerType type="P6" is\_restrict="1" ref="char"/>

<pointerType type="P7" ref="F2"/>

<structType type="S0">

<symbols>

<id type="double">

<name>real</name>

</id>

<id type="double">

<name>img</name>

</id>

</symbols>

</structType>

<functionType type="F0" return\_type="S0">

<params>

<name type="S0">x</name>

<name type="double">y</name>

</params>

</functionType>

<functionType type="F1" return\_type="int">

<params/>

</functionType>

<functionType type="F2" return\_type="int">

<params/>

</functionType>

<functionType type="F3" return\_type="S0">

<params>

<name type="S0">x</name>

<name type="double">y</name>

</params>

</functionType>

</typeTable>

<globalSymbols>

<id type="F0" sclass="extern\_def">

<name>complex\_add</name>

</id>

<id type="S0" sclass="extern\_def">

<name>x</name>

</id>

<id type="F1" sclass="extern\_def">

<name>main</name>

</id>

<id type="F2" sclass="extern\_def">

<name>printf</name>

</id>

<id type="S0" sclass="typedef\_name">

<name>complex\_t</name>

</id>

<id type="S0" sclass="tagname">

<name>complex</name>

</id>

</globalSymbols>

<globalDeclarations>

<varDecl>

<name>x</name>

</varDecl>

<funcDecl>

<name>complex\_add</name>

</funcDecl>

<functionDefinition>

<name>main</name>

<symbols>

<id type="S0" sclass="auto">

<name>z</name>

</id>

</symbols>

<params/>

<body>

<compoundStatement>

<symbols>

<id type="S0" sclass="auto">

<name>z</name>

</id>

</symbols>

<declarations>

<varDecl>

<name>z</name>

</varDecl>

</declarations>

<body>

<exprStatement>

<assignExpr type="double">

<memberRef type="double" member="real">

<varAddr type="P0" scope="local">x</varAddr>

</memberRef>

<floatConstant type="double">1.0</floatConstant>

</assignExpr>

</exprStatement>

<exprStatement>

<assignExpr type="double">

<memberRef type="double" member="img">

<varAddr type="P1" scope="local">x</varAddr>

</memberRef>

<floatConstant type="double">2.0</floatConstant>

</assignExpr>

</exprStatement>

<exprStatement>

<assignExpr type="S0">

<Var type="S0" scope="local">z</Var>

<functionCall type="S0">

<function>

<funcAddr type="P5">complex\_add</funcAddr>

</function>

<arguments>

<Var type="S0" scope="local">x</Var>

<floatConstant type="double">1.0</floatConstant>

</arguments>

</functionCall>

</assignExpr>

</exprStatement>

<exprStatement>

<functionCall type="int">

<function>

<funcAddr type="F2">printf</funcAddr>

</function>

<arguments>

<stringConstant>z=(%f,%f)\n</stringConstant>

<memberRef type="double" member="real">

<varAddr type="P2" scope="local">z</varAddr>

</memberRef>

<memberRef type="double" member="img">

<varAddr type="P3" scope="local">z</varAddr>

</memberRef>

</arguments>

</functionCall>

</exprStatement>

</body>

</compoundStatement>

</body>

</functionDefinition>

<functionDefinition>

<name>complex\_add</name>

<symbols>

<id type="S0" sclass="param">

<name>x</name>

</id>

<id type="double" sclass="param">

<name>y</name>

</id>

</symbols>

<params>

<name type="S0">x</name>

<name type="double">y</name>

</params>

<body>

<compoundStatement>

<symbols>

<id type="S0" sclass="param">

<name>x</name>

</id>

<id type="double" sclass="param">

<name>y</name>

</id>

</symbols>

<declarations/>

<body>

<exprStatement>

<asgPlusExpr type="double">

<memberRef type="double" member="real">

<varAddr type="P4" scope="param">x</varAddr>

</memberRef>

<Var type="double" scope="param">y</Var>

</asgPlusExpr>

</exprStatement>

<returnStatement>

<Var type="S0" scope="param">x</Var>

</returnStatement>

</body>

</compoundStatement>

</body>

</functionDefinition>

</globalDeclarations>

</XcodeProgram>