XcodeML/C++ 仕様書

**Draft 0.2J (September 2, 2015)**

**XcalableMP/Omni Compiler Project**

改版履歴

XcodeML/C Version 0.91J

* 配列要素の参照の要素を変更。
* 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++ draft 0.2J

目次

[1 はじめに 1](#_Toc422212366)

[2 XcodeProgram要素 1](#_Toc422212369)

[2.1 name要素 1](#_Toc422212370)

[2.2 value要素 1](#_Toc422212371)

[3 typeTable要素 2](#_Toc422212372)

[3.1 データ型識別名 2](#_Toc422212373)

[3.2 basicType要素 3](#_Toc422212374)

[3.3 pointerType要素 3](#_Toc422212375)

[3.4 functionType要素 4](#_Toc422212376)

[3.5 arrayType要素 4](#_Toc422212377)

[3.6 structType要素、unionType要素 5](#_Toc422212378)

[3.7 enumType要素 6](#_Toc422212379)

[3.8 データ型定義要素のオプション属性 7](#_Toc422212380)

[4 シンボルリスト 7](#_Toc422212381)

[4.1 id要素 7](#_Toc422212382)

[4.2 globalSymbols要素 8](#_Toc422212383)

[4.3 symbols要素 8](#_Toc422212384)

[5 globalDeclarations要素 8](#_Toc422212385)

[5.1 functionDefinition要素 9](#_Toc422212386)

[5.2 params要素 9](#_Toc422212387)

[5.3 varDecl要素 9](#_Toc422212388)

[5.4 functionDecl要素 10](#_Toc422212389)

[6 文の要素 10](#_Toc422212390)

[6.1 exprStatement要素 10](#_Toc422212391)

[6.2 compoundStatement要素 10](#_Toc422212392)

[6.3 ifStatement要素 10](#_Toc422212393)

[6.4 whileStatment要素 10](#_Toc422212394)

[6.5 doStatement要素 11](#_Toc422212395)

[6.6 forStatement要素 11](#_Toc422212396)

[6.7 rangeForStatement要素 11](#_Toc422212397)

[6.8 breakStatement要素 11](#_Toc422212398)

[6.9 continueStatement要素 11](#_Toc422212399)

[6.10 returnStatment要素 11](#_Toc422212400)

[6.11 gotoStatement要素 11](#_Toc422212401)

[6.12 statementLabel要素 12](#_Toc422212402)

[6.13 switchStatement要素 12](#_Toc422212403)

[6.14 caseLabel要素 12](#_Toc422212404)

[6.15 gccRangedCaseLabel要素 12](#_Toc422212405)

[6.16 defaultLabel要素 12](#_Toc422212406)

[6.17 pragma要素 12](#_Toc422212407)

[6.18 text要素 12](#_Toc422212408)

[6.19 行番号属性 13](#_Toc422212409)

[7 式の要素 13](#_Toc422212410)

[7.1 定数の要素 13](#_Toc422212411)

[7.2 変数参照の要素 13](#_Toc422212412)

[7.3 pointerRef要素 14](#_Toc422212413)

[7.4 配列要素の参照の要素 14](#_Toc422212414)

[7.5 構造体メンバの参照の要素 14](#_Toc422212415)

[7.6 scopeResolutionExpr要素 15](#_Toc422212416)

[7.7 thisExpr要素 15](#_Toc422212417)

[7.8 assignExpr 要素 15](#_Toc422212418)

[7.9 2項演算式の要素 15](#_Toc422212419)

[7.10 単項演算式の要素 16](#_Toc422212420)

[7.11 functionCall要素 17](#_Toc422212421)

[7.12 commaExpr要素 17](#_Toc422212422)

[7.13 postIncrExpr要素、postDecrExpr要素、preIncrExpr要素、preDecrExpr要素 17](#_Toc422212423)

[7.14 castExpr要素 17](#_Toc422212424)

[7.15 condExpr要素 18](#_Toc422212425)

[7.16 gccCompoundExpr要素 18](#_Toc422212426)

[7.17 lambdaExpr要素 18](#_Toc422212427)

[7.18 newExpr要素 18](#_Toc422212428)

[7.19 deleteExpr要素 18](#_Toc422212429)

[7.20 【その他未検討の式】 18](#_Toc422212430)

[8 XcalableMP固有の要素 18](#_Toc422212431)

[8.1 coArrayType要素 18](#_Toc422212432)

[8.2 coArrayRef要素 19](#_Toc422212433)

[8.3 subArrayRef要素 19](#_Toc422212434)

[8.4 indexRange要素 20](#_Toc422212435)

[9 その他の要素・属性 20](#_Toc422212436)

[9.1 typeName要素 20](#_Toc422212437)

[9.2 is\_gccExtension属性 20](#_Toc422212438)

[9.3 gccAsm要素、gccAsmDefinition要素、gccAsmStatement要素 20](#_Toc422212439)

[9.4 gccAttributes要素 22](#_Toc422212440)

[9.5 builtin\_op要素 24](#_Toc422212441)

[9.6 is\_gccSyntax属性 25](#_Toc422212442)

[9.7 is\_modified属性 25](#_Toc422212443)

[9.8 【未検討項目】 25](#_Toc422212444)

[10 コード例 26](#_Toc422212445)

# はじめに

この仕様書は、C言語およびC++言語の中間表現であるXcodeMLを記述する。XcodeMLは、以下の特徴を持つ。

* 入力されたC言語またはC++言語のプログラムを意味的（semantic）に復元可能な情報を保持する。復元のために必要のない情報であっても、コンパイラ（トランスレータ）にとって意味のある情報が付加されることもある。
* C++言語相当の型情報を、シンボル情報から独立した形で表現する。
* human-readableなフォーマット(XML)をもつ。

# XcodeProgram要素

<XcodeProgram>

typeTable要素

globalSymbols要素

globalDeclarations要素

</XcodeProgram>

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

Xcodeによって表現されるプログラムは、Type tableとglobal Id table, 外部定義の列からなる。Xocdeファイルのトップレベルの要素は、XcodeProgram 要素である。XcodeProgram 要素は以下の要素を含む。

* typeTable要素　– プログラムで利用されているデータ型の情報
* globalSybmols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

## name要素

<name>[ スコープ名 : : ] …名前</name>

属性（optional）: type, fullName

変数名や、タイプ名などの名前を指定する要素である。０個以上のスコープ名と「::」の組と、名前を一連の文字列として持つ。一連の文字列は空白文字を含んではならない。

属性として、type属性とfullName属性を持つ。type属性の属性値はタイプ識別名である。fullName属性は、0個以上のスコープ名と「::」の組と、名前を一連の文字列として持つ。一連の文字列は空白文字を含んではならない。

name要素のスコープ名は、C++の文法に従ってその文脈で省略できるスコープ名を省略した形で表現されてよく、入力プログラムに近い表現であることが望ましい。fullName属性（必須でなない）は、オブジェクトが属するネストされた名前空間の名前をすべて列挙する表現であり、文脈に依存せずオブジェクトを特定することができる。

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てよい。

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

<name type=”int”>a</name>

<name type=”int”>NS::a</name>

(2)におけるaは、以下のように表現されてよい。

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

<name type=”int”>NS::a</name>

## value要素

<value>

[ 式 … ]　*or*　value要素

</value>

属性: なし

初期値を表現する。次の子要素を持つ。

* 式(0-複数)　－　値を示す式。
* value　－　ネストされた値。"{ ... }" に対応する。

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

<typeTable>

{ pointerType要素 *or* functionType要素 *or* arrayType要素 *or* structType要素 *or*

unitionType要素 *or* enumType要素 *or* basicType要素 }

…

</typeTable>

属性（optional）: なし

typeTable要素は、このファイル全体で使われているデータ型についての情報を定義する。 typeTable要素は、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。

* pointerType要素
* functionType要素
* arrayType要素
* structType要素
* unitionType要素
* enumType要素
* basicType要素

## データ型識別名【要修正】

プログラム内において、データ型はデータ名で区別される。その名前は、次の２つのいずれかである。

* 基本データ型名

C言語の基本データ型

'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), auto

【要検討】\_Bool型って？

【要検討】auto型の扱い

\_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'

* 派生データ型名　–　上記、基本データ型名以外の任意の英数字の並び。

派生データ型の名前は、プログラム内でユニークなものでなくてはならない。

## basicType要素

<basicType/>

属性（必須）: type, name

属性（optional）: データ型定義要素属性

basicType 要素は、基本データ型に、is\_constなどの属性を加えたものを定義する。 属性として以下のものをもつ。

* type
* name

例:

struct {int x; int y;} s;

struct s const \* volatile p;

は次のXcodeMLに変換される。

<structType type="S0">

...

</structType>

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

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

## pointerType要素

<pointerType/>

属性（必須）: type, ref

属性（optional）: データ型定義要素属性

pointerType要素はポインタのデータ型を定義する。以下の属性を持つ。

* type　－　このポインタ型の派生データ型の名前。
* ref　－　このポインタ・データ型が参照するデータ型の名前

pointType要素は、他の要素を持たない。

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

<functionType>

　　[ params属性 ]

</functionType>

属性（必須）: type, return\_type

属性（optional）: is\_inline

funtionType要素は、関数データ型を定義する。

* type　－　この関数型の派生データ型の名前
* return\_type　－　この関数型が返すデータ型の名前
* 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、引数要素に対応するparam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>

[ 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、配列データ型を定義する。arrayType要素は以下の属性を持つ。

* type　－　この配列型の派生データ型の名前
* element\_type　－　配列要素のデータ型の識別子を指定する
* 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"/>

## structType要素、unionType要素

<structType>

symbols要素

</structType>

属性（必須）: type

属性（optional）: データ型定義要素属性

<unionType>

symbols要素

</unionType>

属性（必須）: type

属性（optional）: データ型定義要素属性

struct(構造体)データ型は、structType要素で定義する。structType要素は以下の属性を持つ。

* type　－　この配列型の派生データ型の名前

Union(共用体)データ型は、unionType要素で定義する。untionType要素は、structType要素と同じ、属性、要素を持つ。

StructType・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 bitField タグに記述する。bitFieldタグは式を子要素に持ち、bitField タグを持つid要素の bit\_field 属性の値は、"\*" とする。

例:

"struct {int x; int y : 8; int z : sizeof(int); } S;"のSに対するstructType要素は以下のようになる。

<stuctType type="S6e89">

<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>

## enumType要素

<enumType>

symbols要素

</enumType>

属性（必須）: type

属性（optional）: データ型定義要素属性

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。

次の子要素を持つ。

* symbols　－　メンバの識別子を定義する。メンバの初期値はid　－　value要素で表す。

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>

## データ型定義要素のオプション属性

データ型定義要素の属性として以下の属性を持つ（省略可）。

* is\_const　－　このデータ型がconst修飾子をもつかどうかの情報、 0 または 1、false または true。省略時は０（false）
* is\_volatile　－　このデータ型がvolatile修飾子をもつかどうかの情報、 0 または 1、false または true。省略時は０（false）
* is\_restrict　－　このデータ型がrestrict修飾子をもつかどうかの情報、 0 または 1、false または true。省略時は０（false）
* is\_static　－　このデータ型がstatic属性をもつかどうかの情報、 0 または 1、false または true。省略時は０（false）

# 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要素は次の属性を持つ。

* sclass　－　storage class をあらわし、 'auto', 'param', 'extern', 'extern\_def', 'static', 'register', 'label', 'tagname', 'moe', 'typedef\_name', ‘namespace\_name’, ‘thread\_local’, ‘mutable’ のいずれか。  
  【要検討】function specifierである ’inline’, ‘vertual’, ‘explicit’もここで表現するか？  
  【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？

【要検討】usingの扱い

* type　－　識別子のデータ型
* bit\_field属性　－　structType・unionTypeにおいてメンバのビットフィールドを指定する。
* is\_gccThread　－　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

以下の要素を持つ。

* 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。
* bitField要素　－　structType・unionTypeにおいてメンバのビットフィールドを指定する。

識別子が変数などの場合、そのアドレスを要素として持つ。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。

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。要素として、大域のスコープを持つ識別子のid要素を持つ。

## symbols要素

symbols要素は、局所スコープを持つ識別子を定義する。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

globalDeclarations要素は、プログラム中の大域変数の宣言や関数定義をするための要素である。以下の要素を持つ。

* 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

functionDefinition要素は、関数定義を行う。   
以下の要素を持つ

* name要素　－　関数名を指定する
* symbols要素　－　パラメータのシンボルリスト   
  symbols要素において、パラメータに対するシンボルテーブルを指定する。
* params要素　－　パラメータの定義
* body要素　－　関数本体,　body要素は要素として文またはfunctionDefinitionを含む。body要素内のfunctionDefinitionは、GCCのネストされた関数を表す。

以下の属性を持つ

* is\_gccExtension属性

## params要素

params要素は、関数の引数の並びを指定する。

* name要素　－　引数に対応するname要素を持つ。
* 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。 引数のデータ型の情報がある場合には、name要素のtype属性に指定する。

## varDecl要素

varDecl要素は、変数の宣言を行う。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、この要素で指定する。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

functionDecl要素は、関数宣言を行う。   
以下の要素を持つ

* name要素　－　関数名を指定する

# 文の要素

Cの文の構文要素に対応するXML要素である。それぞれの要素には、行番号が属性として付加されており、文が現れた行番号やファイルの情報を取り出すことができる。

## exprStatement要素

exprStatment要素で、式で表現される文を表す。式の要素を持つ。

## compoundStatement要素

compoundStatement要素で、複文を表現する。以下の要素を持つ。

* symbols要素　－　この複文の中で定義されているシンボルリスト
* declarations要素　－　この複文の中で定義する宣言に対するvarDecl要素、functionDefinition要素、およびfunctionDecl要素を含む
* body 要素　－　複文本体を含む

## ifStatement要素

if文をあらわす要素。以下の要素を持つ。

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

## whileStatment要素

while文を表す要素。以下の要素を持つ

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

## doStatement要素

do文を表す要素。以下の要素を持つ。

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

## forStatement要素

for文を表す要素。以下の要素を持つ。

* init要素　－　初期化式を要素として含む 【要確認】式だけでなく単純宣言文も許されるがサポートできているか？
* condition要素　－　条件式を含む
* iter要素　－　繰り返し式を含む
* body要素　－　for文の本体

## rangeForStatement要素

C++の以下の形のfor文を表す要素。

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

以下の要素を持つ。

* var要素　－　【要検討】宣言文の表現
* range要素　－　【要検討】
* body要素　－　for文の本体

## breakStatement要素

break文を表す要素。空要素である。

## continueStatement要素

continue文を表す要素。空要素である。

## returnStatment要素

returnを表す要素。returnする式を、要素として持つ。

## gotoStatement要素

goto文を表す要素。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　－　ジャンプ先のアドレス値を指定する。

## statementLabel要素

goto文のターゲットのラベルを表す。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文を表す要素。以下の要素を持つ。

* value要素　－　switchする値を指定する。
* body要素　－　switch文の本体を指定する。

## caseLabel要素

switch文のcase文を表す要素。caseの値を要素としてもつ。

* 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要素。 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。

## pragma要素

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

## 行番号属性

すべての文を表す要素は、文の行番号を表す以下の属性を持つことができる。

* 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

# 式の要素

Cの式の構文要素に対応するXML要素である。それぞれの要素には、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。
* lvalue属性　―　式が左辺値であることを示す。C++では演算子はoverrideできるので、オペランドが左辺値か右辺値かは演算子から判断するのでなく、この属性の有無で判断しなければならない。

## 定数の要素

定数は以下の要素によって表現する。

* intConstant要素　－　整数の値を持つ定数を表す。内容として、十進数もしくは、16進数（0xから始まる）を記述する。
* longlongContant要素　－　32ビット16進数(0xから始まる)の２つの数字を記述する。
* floatConstant要素　－　float・double・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。
* stringConstant要素　－　内容に文字列を記述する。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。  
  【要検討】char16\_tとchar32\_tのconstant表現が必要か？
* moeConstant要素　－　enum型の定数を表す。内容にenum定数を記述する。
* booleanConstant要素　－　真理値リテラル。falseまたはtrue。
* funcAddr要素　－　関数のアドレスを表す。内容に関数名を記述する。

定数のデータ型は、type属性で指定する。

moeConstantの場合は、この式を含むスコープのシンボルテーブルの中に、指定されたmoe定数がふくまれていなくてはならない。

## 変数参照の要素

変数への参照は、大域変数、パラメータ変数、局所変数、それぞれに対して、異なる要素を用いる。

* Var要素　－　大域変数を参照する式。内容に変数名を指定する。
* varAddr要素　－　大域変数のアドレスを参照する式。内容に変数名を指定する。

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

<Var>var\_name</Var>

は、

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

と等価。

<varAddr>var\_name</varAddr>

は,Cでの表現は&var\_name

## pointerRef要素

pointerRef要素は、sub要素の式をアドレスとし、メモリ参照を行う。

## 配列要素の参照の要素

配列への参照には、以下の要素を用いる。

* arrayRef　－　配列要素を参照する式。内容に先頭アドレスと添字を指定する。
* arrayAddr　－　配列の先頭アドレスを参照する式。内容に配列名を指定する。

配列要素への参照は、arrayAddrを用いてアドレスを取得し、arrayRefで要素にアクセスする。   
変数参照の要素と同様に、scope属性を使い局所変数を区別する。

## 構造体メンバの参照の要素

構造体メンバへの参照は、参照アドレスを参照するmemberAddrとメンバを参照するmemberRef、メンバ配列のアドレスを参照するmemberArrayAddrがある。

【要検討】例をあげる。メンバポインタ演算子 X.\*Y、X->\*Y 対応を追加する。

* memberAddr　－　構造体メンバのアドレスを参照する。内容に構造体のアドレスの式を指定し、member属性値にメンバ名を指定する。
* memberRef　－　構造体メンバを参照する。内容に構造体のアドレスの式を指定し、member属性値にメンバ名を指定する。
* memberArrayAddr　－　構造体の配列メンバのアドレスを参照する。内容に構造体のアドレスの式を指定し、member属性値にメンバ名を指定する。
* memberArrayRef　－　構造体の配列メンバを参照する。内容に構造体のアドレスの式を指定し、member属性値にメンバ名を指定する。

<memberRef memer="xxx">addr</memberRef>

は

<pointerRef >

<memberAddr memer="xxx">addr<memberAddr>

</pointerRef>

と等価。

## scopeResolutionExpr要素

scopeResolutionExpr 要素は、スコープ解決演算子 :: を使った以下の式に対応する。

:: identifier

class-name :: identifier

namespace :: identifier

【未稿】

## thisExpr要素

thisExpr 要素は、C++の this に対応する。

## 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。

## 2項演算式の要素

以下の要素は算術２項演算式を表す。被演算子の２つの要素を内容に指定する。左式が第一要素、右式が第２要素である。

* 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

これに加えて、代入式とあわせた以下の要素がある。

* 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

以上の代入つき演算式の左式（初めの要素）は、lvalueでなくてはならない。

以下は論理２項演算式を表す要素である。被演算子の２つの要素を内容に指定する。

* 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

## 単項演算式の要素

以下の要素は算術単項演算式を表す。被演算子の式の要素を内容に指定する。

* unaryMinusExpr　－　符号反転
* bitNotExpr　－　ビット反転

以下の要素は、論理単項演算式を表す。被演算子の式の要素を内容に指定する。

* logNotExpr　－　論理否定

以下の要素は、sizeof演算子とGCC拡張の演算子を表す。

* sizeOfExpr　－　sizeof演算子。子要素に式またはtypeName要素を指定する。
* 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。以下の２つの要素を持つ。

* function要素　－　呼び出す関数のアドレスを指定する。
* arguments要素　－　引数の式を指定する。

## 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）は、commaExpr要素で表す。

## postIncrExpr要素、postDecrExpr要素、preIncrExpr要素、preDecrExpr要素

postIncrExpr要素、postDecrExpr要素は、C言語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言語のプレインクリメント、デクリメント式を表す。

## castExpr要素

castExpr要素は型変換の式、または複合リテラルを表す。

以下の属性を持つ

* type属性　－　変換後の式のタイプを指定する。
* is\_gccExtension属性

以下の子要素を持つ。

* value　－　複合リテラルのリテラル部を表す。

## condExpr要素

condExpr要素は、三項演算子 ? : に対応する要素である。

次の例の様に子要素として3つの式の要素を持つ。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）。

<condExpr>

(式)

(式)

(式)

</condExpr>

## gccCompoundExpr要素

gcc拡張の複文式に対応する要素。

compoundStatement要素を要素としてもつ。

* compoundStatement要素　－　複文式の内容を指定する。

## lambdaExpr要素

【未稿】

## newExpr要素

【未稿】

## deleteExpr要素

【未稿】

## 【その他未検討の式】

* ~ クラス名
* テンプレート名 < … >
* typeid ( … )
* dynamic\_cast, static\_cast, reinterpret\_cast, const\_cast

dynamic\_cast < *type\_id* > ( *expression* )

# XcalableMP固有の要素

## 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

## typeName要素

typeName要素は型名を表す。sizeOfExpr、gccAlignOfExpr、builtin\_opなどの子要素として指定される。属性にデータ型識別名を示す type を持つ。

## is\_gccExtension属性

is\_gccExtension属性は、GCCの \_\_extension\_\_ キーワードを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。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。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】

* 宣言
  + asm ( … )
  + 結合指定 extern “C” double x;
* 宣言子
  + ポインタ演算子 & &&
  + メンバへのポインタ
  + 省略時引数 void foo( int, int = 7 )
* クラス
  + クラス宣言 class X { … }
  + メンバ関数
* Derived class
  + 基底指定子、アクセス指定子
    - Virtual
    - private protected public
  + 多重継承
  + 抽象クラス
  + 随伴クラス
* 特殊メンバ関数
  + constructor, destructor
  + 一時object
  + 変換関数
  + コピー
* Overloading（多重定義）
  + 要検討：多重定義解決可能な情報をもつこと
  + 多重定義演算子の宣言　operator
* テンプレート  
  【未検討】抽象とインスタンスの両方を表現する。
* 例外処理
  + try/catch
  + throw
  + noexcept

# 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>