2016年3月30日

株式会社トランス・ニュー・テクノロジー

「Omni XcalableMPコンパイラの  
C++フロントエンドの開発」報告書

目次

[1 本報告書の構成 3](#_Toc447053931)

[2 納品ソフトウェアの利用方法 4](#_Toc447053932)

[2.1 gitリポジトリの構成 4](#_Toc447053933)

[2.2 ビルド手順 4](#_Toc447053934)

[2.2.1 Clangのインストール 4](#_Toc447053935)

[2.2.2 CtoXcodeMLのビルド 6](#_Toc447053936)

[2.3 ビルドされたツールの利用方法 7](#_Toc447053937)

[3 作業報告 9](#_Toc447053938)

[3.1 納品ソフトウェアの内部実装に関する解説 9](#_Toc447053939)

[3.1.1 各ソースコードについて 9](#_Toc447053940)

[3.1.2 実装に用いているC++デザインパターンについて 10](#_Toc447053941)

[3.2 テスト用ソースコードによる評価 13](#_Toc447053942)

[3.2.1 テスト00: アクセス指定子のテスト 14](#_Toc447053943)

[3.2.2 テスト01: メンバ関数宣言のテスト 14](#_Toc447053944)

[3.2.3 テスト02: コンストラクタのメンバ初期化子リストのテスト 15](#_Toc447053945)

[3.2.4 テスト03: 二項と単項の「＊」演算子オーバーロードのテスト 16](#_Toc447053946)

[3.2.5 テスト04: クラス継承のテスト 16](#_Toc447053947)

[3.2.6 テスト05: 各種の演算子オーバーロードのテスト 18](#_Toc447053948)

[3.2.7 テスト06: 構造体のメンバ利用のテスト 23](#_Toc447053949)

[3.2.8 テスト07: 同じ型を持つ関数が複数宣言される場合のテスト 24](#_Toc447053950)

[3.2.9 テスト08: 正規化可能な単純なクラス宣言のテスト 25](#_Toc447053951)

[3.2.10 テスト09: namespaceやクラスで囲われた宣言のテスト 26](#_Toc447053952)

[3.2.11 テスト10: 初期化リストのテスト 27](#_Toc447053953)

[3.2.12 テスト11: クラスでのメンバ関数呼び出しのテスト 28](#_Toc447053954)

[3.2.13 テスト12: 複数の名前空間での同名の関数定義のテスト 29](#_Toc447053955)

[3.2.14 テスト13: 複雑なクラス宣言のテスト 31](#_Toc447053956)

[3.2.15 テスト14: ネストされたクラスの先行宣言のテスト 33](#_Toc447053957)

[3.2.16 テスト15: const修飾版と非修飾版のメンバ関数定義のテスト 34](#_Toc447053958)

[3.2.17 テスト16: 複雑な型を持つ関数のテスト 35](#_Toc447053959)

[3.2.18 テスト17: ユーザー定義リテラルのテスト 37](#_Toc447053960)

[3.2.19 テスト18: 深くネストされたnamespaceのテスト 38](#_Toc447053961)

[3.2.20 テスト19: 正規化ができないクラス宣言のテスト 39](#_Toc447053962)

[3.2.21 テスト20: 初期化リストによるコンストラクタ呼び出しのテスト 40](#_Toc447053963)

[3.2.22 テスト21: オーバーロード演算子の呼び出しのテスト 40](#_Toc447053964)

[3.2.23 テスト22: C++スタイルのキャストのテスト 41](#_Toc447053965)

[3.2.24 テスト23: メンバへのポインタのテスト 42](#_Toc447053966)

[3.2.25 テスト24: thisポインタを使った式のテスト 42](#_Toc447053967)

[3.2.26 テスト25: tryとcatchのテスト 43](#_Toc447053968)

# 本報告書の構成

本報告書は、本案件に関する解説書を兼ねた文書であり、下記の内容からなる。

* 納品されたソフトウェア（CtoXcodeML）の利用方法について  
  下記の各パートからなる。
  + gitリポジトリの構成
  + ビルド方法
  + ビルドされたツールの利用方法
* 作業報告書一式  
  下記の各パートからなる。
  + 納品ソフトウェアの内部実装に関する解説
  + テスト用ソースコードによる評価

# 納品ソフトウェアの利用方法

本案件で開発されたソフトウェアは、omni-compiler.org上に設置されたgitレポジトリでバージョン管理されており、納品物はここに置かれている。

## gitリポジトリの構成

* docs/ ディレクトリ
  + docs/CtoXcodeMLreport-20160330.docx  
    本ドキュメントである。
  + docs/XcodeML\_CXX\_1.1.docx  
    現段階でのXcodeML C++ 仕様書である。
* src/ ディレクトリ  
  ソースコード一式を含む。
* testcases/ ディレクトリ  
  CtoXcodeMLの動作テストに用いたC/C++ソースコードと、それぞれについてのCtoXcodeMLによる変換結果を含む。

## ビルド手順

まず、Clangをソースコードからビルドし、 /usr/local 以下にインストールしておく必要がある。そのあと、ソースコードをビルドする。本開発の開始時点でベースとした Clangのバージョンは3.6.2であり、3.7.1でもビルドと実行ができることは確認できているが、最新の3.8.0ではビルド時にエラーになるので、3.6.2か3.7.1を用いるものとする。

### Clangのインストール

Clangのビルドには、同バージョンのLLVMコアとcompiler-rtが必要である。いずれも<http://llvm.org/releases/download.html>から各バージョンのものが取得できる（バージョン3.6.2は同ページの<http://llvm.org/releases/download.html#3.6.2> にあり、バージョン3.7.1は同ページの<http://llvm.org/releases/download.html#3.7.1> にある）。また同じページの先頭のリンクでsubversionを用いたリポジトリも公開されている。しかしそれらを利用して取得するよりも、github のミラーを取得したほうが利便性が高い（git checkoutだけで各バージョンのブランチに切り替えることができる）ため、そちらを利用する方法でビルド手順を示す。

% cd 作業用ディレクトリ  
% git clone https://github.com/llvm-mirror/llvm  
% cd llvm  
% git checkout release\_36（またはrelease\_37）

% cd 作業用ディレクトリ/llvm/projects  
% git clone https://github.com/llvm-mirror/compiler-rt  
% cd compiler-rt  
% git checkout release\_36（またはrelease\_37）

% cd 作業用ディレクトリ/llvm/tools  
% git clone https://github.com/llvm-mirror/clang  
% cd clang  
% git checkout release\_36（またはrelease\_37）

% cd 作業用ディレクトリ  
% mkdir build  
 ※llvmのビルド用のディレクトリを準備する。名前は何でもよい。  
% cmake –G Ninja ../llvm  
 ビルド用のサブディレクトリ・ファイル群が生成される。  
　　また、この時点ですでにCコンパイラとしてclangを用いることができる  
　　のであればCC=clang CXX=clang++ という環境変数を設定してcmakeを  
　　おこなってもよい（gcc/g++を用いるよりも高速にビルドが完了する）。  
% ninja  
 ※（PATH環境変数で探せる位置に無いなら適宜フルパスで指定する）  
 ビルドが実行される。マルチコアの場合には自動的に並列ビルドとなる。  
 初回は4コア2スレッドのCPU（8並列）でも二時間ぐらいかかる。  
 明示的に並列度を指定する場合は-jオプションを用いる。ビルドの終盤  
　　でリンクをおこなう際にはメモリ不足になりやすいので、ビルドに失敗  
　　した場合には並列度を下げてやり直すと良い。また、前述のようにして  
　　clang/clang++を用いた方がメモリ消費も少なくて済むようである。  
　　一度ビルドが成功したあとは、再ビルドする際にcmakeから実行する  
　　必要はなく、ninjaコマンドのみを実行すればよい。

以上は通常のRelease版のビルド手順である。これ以外に、デバッグ版のビルドを下記のようにしておこなうことができる。このようにすると、LLVMの各種のコマンド(optなど)で –debug や –debug-only オプションが使えるようになり、コンパイラ内部のデバッグに役立つ情報が得られるようになる。

% cmake –G ninja –DCMAKE\_BUILD\_TYPE=Debug ../llvm  
% ninja  
 ※ビルドを実行（既に実行済みであれば、必要な部分だけリビルド）する。  
 Releaseビルドに比べてさらに多くのメモリを必要とするので、  
 メモリ不足で失敗するようなら-jオプションで並列度を下げて調整する。  
% ninja install  
 ※/usr/local以下にインストールされる。

CMakeのこの他のオプション指定については<http://llvm.org/docs/CMake.html> に書かれているので、そちらを参照されたい。

### CtoXcodeMLのビルド

ビルドのためにlibxml2が必要である。Ubuntuであれば  
 % sudo apt-get libxml2-dev  
のようにしてインストールする。

gitレポジトリから取得した src/ ディレクトリ内で make を実行すればビルドが実行され、 CtoXcodeMLという名の実行ファイルが生成される。clang/LLVMとして3.6.2を用いている場合には警告が出ないが、3.7.1を用いている場合にはClang ASTの各種のenum型の定義において新規の値が追加されたために “…… not handled in switch” の警告が出る（現状ではそのようなASTを扱わないため無視して構わない）。

make distcleanをすると、ビルド結果が全て消される。

Make cleanをすると、ビルド結果のうち、もっともビルドに時間のかかるXcodeMlRAV.oを残して、その他を消す。

他にもMakefile内にいくつかターゲットが記述されているが、開発途上段階でのテスト用に準備されたものが多く、動作は保障されない。

## ビルドされたツールの利用方法

例として、

clang -I some-include-dir/ -D user\_defined\_macro=value foo.c

としてコンパイルするような foo.c を対象にする場合、

./CtoXcodeML foo.c -- -I some-include-dir/ -D user\_defined\_macro=value

のように「-- の前に対象ファイル名、 -- のあとにその他のオプション」として指定する。その他のオプションが不要なときも「--」が必要である（この -- が無いと ./compile\_commands.json というファイルが用いられる。これについての詳細は<http://clang.llvm.org/docs/JSONCompilationDatabase.html>を参照のこと）。

以下は、 -- よりも左側に指定することのできるCtoXcodeML独自のオプションについて解説する。

./CtoXcodeML -help

でオプションの一覧が出る。

-file - emit 'file'

-lineno - emit 'lineno'

-column - emit 'column'

…これらはそれぞれ、出力されるXcodeMLの要素にソースコードのファイル名情報・行番号情報・桁位置情報を付加するオプションである。 C\_Frontと同一の結果を得たい場合には –file と –lineno を指定する必要がある。

-disable-typeTable - disable <typeTable>

-disable-symbols - disable <globalSymbols>, <symbols>

-disable-declarations - disable <globalDeclarations>, <declarations>

…これらはそれぞれ、 XcodeMLの一部の出力を抑制するためのオプションである。主にデバッグ用に用いる。

-trace-typeTable - emit traces on <typeTable>

-fulltrace-typeTable - emit full-traces on <typeTable>

-trace-symbols - emit traces on <globalSymbols>, <symbols>

-trace-declarations - emit traces on <globalDeclarations>, <declarations>

-trace-rav - trace Recursive AST Visitor

…これらはそれぞれ、 XcodeMLの出力中にコメントとしてデバッグ情報を含めるためのオプションである。

-typenamemap=<string> - a map file of typename substitution

…これはXcodeMLのtypeTableで扱われる型名を置換するためのオプションである。置換したい型名と置換後の型名をそれぞれ空白区切りで交互に並べたファイルを準備し、そのファイル名を上記の<string>のところに指定する。ただしファイルの末尾に空白や改行があると正しく読み込めなくなるので、ファイルの最終行は改行無しで終わらなければならない。

これ以外にもいくつかのオプションがあるが動作保障はされない。

# 作業報告

## 納品ソフトウェアの内部実装に関する解説

### 各ソースコードについて

* XcodeMlVisitorBase.cpp, XcodeMlVisitorBase.h  
  下記の各種の \*Visitor.cpp, \*Visitor.h の実装の基底クラスである  
  class XcodeMlVisitorBase を準備している部分。  
  意味的にはこのさらに上位に CRTPパターンで書かれたRecursiveASTvisitor クラスを基底に持つのだが、RecursiveASTvisitorクラスは大量のメソッドを持つため、本当に各種の\*Visitorの基底クラスとして実装するとコンパイル時間が何倍にもなるので、pimpl イディオム相当の class RAVBidirBridge をつかって、下記のXcodeMlRAV のほうに RecursiveASTvisitor の実装の部分を隠蔽している。（CRTPパターンとpimplイディオムについては後述）
* XcodeMlRAV.cpp, XcodeMlRAV.h  
  clang の libtooling ライブラリ内の RecursiveASTvisitor.h を利用したクラスを実装している部分。 RAVBidirBridge をつかってclass XcodeMlVisitorBase との間で双方向に橋渡しをしている。
* TypeTableVisitor.cpp, TypeTableVisitor.h  
  clang の AST から XcodeML の <typeTable> 部を生成する部分。  
  また、 class TypeTableInfo というデータ構造を作成し、clang AST の QualType で示された値 (型の種別情報) とXcodeML の の type 名 (文字列) との対応関係を管理する。
* InheritanceInfo.cpp, InheritanceInfo.h  
  C++のクラス・構造体の継承関係の情報を扱う部分。TypeTableInfoでは表されていない情報を管理する。
* SymbolTableVisitor.cpp, SymbolTableVisitor.h  
  clang の AST から XcodeML の <globalSymbols> 部および <symbols> 部を生成する部分。
* DeclarationsVisitor.cpp, DeclarationsVisitor.h  
  clang の AST から XcodeML の <globalDeclarations> 部および <declarations> 部を生成する部分。この部分がCtoXcodeMLの中でもっとも大きな部分を占める。
* operator.cpp, operator.h  
  オーバーロードされた演算子の種類をXcodeMLでの演算子名に変換するためのOverloadedOperatorKindToString関数を持つ。
* CtoXcodeML.cpp  
  main 関数部分。与えられたコマンドラインから AST を構成し、上記各 Visitor にAST を渡す部分。

### 実装に用いているC++デザインパターンについて

これらのソースコードでは、C++のデザインパターンとして下記の二つのテクニックを用いている。

#### CRTP (Curiously Recurring Template Pattern)

親クラスとして  
　　template <class T> class 親クラス { … };  
のようなものを準備しておいて、利用者は  
　　class 利用者クラス : public 親クラス<利用者クラス> { … };  
のようにして用いる、というデザインパターン。

「奇妙に再帰的なテンプレート」パターンという名前のとおり、親クラスの型引数のところに子クラスの型をいれてそれを継承するという、一見するととても奇妙な形をしている。これは、 template を使わずに下記のように書いた場合と、やりたいことはだいたい同じだと考えてよい。

まず親クラスを  
　　class 親クラス {  
 virtual 仮想メソッド(…) {デフォルトの定義};  
 普通のメソッド(…) { 上記仮想メソッドを使って何か実装; };  
 };  
と定義しておいて、このクラスを継承して利用する側では  
　　class 利用者クラス : public 親クラス {  
 仮想メソッド(…) override { 何か自分用の中身を実装; };  
 };  
のようにする。これにより、親の「普通のメソッド」の動作の一部が利用者クラス側で変更可能となる。

これに対し、CRTPを用いた書き方は次のようになる。

まず親クラスを  
　　template <class D> class 親クラス {  
 置き変え可能なメソッド(…) {デフォルトの定義};  
 普通のメソッド(…) {  
 static\_cast<D\*>(this)->置き変え可能なメソッド(…);  
 という形で呼び出して実装;  
 };  
 };  
としておいて、利用者側で  
 class 利用者クラス : public 親クラス<利用者クラス> {  
 置き変え可能なメソッド(…) { なんか自分用の中身を実装; };  
 };  
とする。これにより、  
 利用者クラス x;  
 x.普通のメソッド(…);  
という形で「普通のメソッド」を呼び出すと、テンプレート展開の結果  
 static\_cast<利用者クラス\*>(this)->置き変え可能なメソッド(…);  
が中で使われることになるので、結果的に「利用者クラス」の側で定義した「置き変え可能なメソッド」が用いられることになる。

virtual を使った場合との違いは下記のようになる。

* virtual を使うと、仮想関数テーブルという「関数のポインタ一覧」がオプジェクトの一部となって埋め込まれているので、 virtual メソッドの呼び出しはメモリアクセスを何度かやって関数ポインタを取得する、というオーバーヘッドがかかる。  
  一方、 CRTP では「テンプレートの実体化」の時点では型引数が静的に定まっているので、メソッド呼び出しは C の関数呼び出しと同等の「静的呼び出し」になる (静的呼び出しはインライン展開できるので、中身が小さいメソッドでは virtual の場合に比べて飛躍的に高速)。
* virtual を使っている場合には  
   親クラス \*pointer = new 利用者クラス(…);  
  のような形で基底クラスのポインタによる抽象化が可能だが、CRTP の「親クラス」というのは実際には単一の基底クラスではない (型引数を入れるごとに違う型ができる) ので、このようなことはできない。つまり、実際には「親クラス」というよりは、単に「同じソースコード片となる部分を共通化して必要に応じて展開いるだけ」といった方が良い。

CRTP を一言でいうと「静的な情報を増やしてコンパイル時点でコンパイラが頑張ることで、利用者の自由度と実行時の高速さの両立をはかったもの」と言える。

#### Pimplイディオム (Pointer to implementation)

CRTP とはまったく逆に、コンパイル時点での静的情報をなるべく分離してソースコード変更の際のリコンパイルの量を減らそうというイディオム。

具体的には、ヘッダファイルに  
　　class 外に見せるクラス {  
 private:  
 class 内部的なクラス \*pimpl;  
 public:  
 コンストラクタ();  
 何らかのメソッド(…);

};  
のような形だけを公開し、 \*.cpp の方 で  
 class 内部的なクラス {  
 …実装すべて…;  
 };  
 外に見せるクラス:コンストラクタ() {  
 pimpl = new 内部的なクラス();  
 }  
 外に見せるクラス:何らかのメソッド(…) {  
 pimpl->何らかのメソッド(…);  
 }  
のようにして外のクラス経由で中の実装にアクセスできるようにする。

デザインパターンの Bridge パターンの応用の一種ともいえる。

## テスト用ソースコードによる評価

テスト用に用いたソースコード一式は、gitレポジトリのtestcases/ ディレクトリに収められている。testcases/ ディレクトリ内には二桁の番号で名づけられたテスト群が納められている（テスト番号がサブディレクトリ名になっており、各サブディレクトリにテストが一個ずつ納められている）。一個のテストは一個のC++ソースファイル（拡張子が .cppで、テストの内容を表すファイル名をもつ）とそれに対応して生成されるべきXcodeML（拡張子が .xml）、およびテスト実行用のMakefileから構成されている。

テストの実行には次の手順を踏む。

* git レポジトリのsrc/ ディレクトリ内でmakeを実行し、 CtoXcodeMLの実行バイナリを作成しておく。
* git レポジトリの testcases/ ディレクトリ内で make check を実行する（各サブディレクトリを番号順にめぐり、それぞれのサブディレクトリ中でmake checkが実行される）。

各サブディレクトリのmake checkで実行される内容は、いずれも

* ../../src/CtoXcodeML に .cpp ファイルを処理させた結果を tmp.xml として生成する
* 「生成されるべきXcodeML」と「処理結果のtmp.xml」とのdiffを取る（ただし <XcodeProgram…> の行はタイムスタンプが含まれるので無視する）

### テスト00: アクセス指定子のテスト

入力ソースコードでのpublic:, protected:, private:の指定（宣言先頭部分でのデフォルトのprivate指定も含む）が、typeTable中での該当classType内で適切に反映されていることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| namespace N {  class A {  int x, y;  public:  int getX() { return x; }  int getY();  protected:  class B {  （以下略） | <classType type="Class0">  <name fullName="N::A">A</name>  <symbols>  <id type="int" access="private">  <name fullName="N::A::x">x</name>  </id>  <id type="int" access="private">  <name fullName="N::A::y">y</name>  </id>  <id type="Function0" sclass="extern\_def" access="public">  <name fullName="N::A::getX">getX</name>  </id>  <id type="Function0" sclass="extern\_def" access="public">  <name fullName="N::A::getY">getY</name>  </id>  <id type="Class1" access="protected">  <name fullName="N::A::b">b</name>  </id>  </symbols>  </classType> |

### テスト01: メンバ関数宣言のテスト

入力ソースコードでのメンバ関数宣言が、typeTable中の該当するclassTypeのsymbols部に列挙されることを確認するテストである。定義本体は外部化されてglobalDeclarationsに移動している（正規化についてのテストは08）。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| class A {  void f(int x, int y);  int g(int x, int y) {  return x + y;  }  int operator\* () { return 42; }  }; | <classType type="Class0">  <name fullName="A">A</name>  <symbols>  <id type="Function0" sclass="extern\_def" access="private">  <name fullName="A::f">f</name>  </id>  <id type="Function1" sclass="extern\_def" access="private">  <name fullName="A::g">g</name>  </id>  <id type="Function2" sclass="extern\_def" access="private">  <operator fullName="A::operator\*">pointerRef</operator>  </id>  </symbols>  </classType> |

### テスト02: コンストラクタのメンバ初期化子リストのテスト

入力ソースコードでの、メンバ初期化子リストを持つコンストラクタが、globalDeclarations部でfunctionDefinitionとして生成され、<name>のかわりに<constructor/>が生成されることや<constructorInitializerList>が適切に生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| class A {  public:  A(int a, int b):  x(a \* b),  y(a + b),  z(0) {}  private:  int x, y, z;  }; | <functionDefinition>  <constructor/>  <symbols>  <id type="int" sclass="param">  <name>a</name>  </id>  <id type="int" sclass="param">  <name>b</name>  </id>  </symbols>  <params>  <name type="int">a</name>  <name type="int">b</name>  </params>  <constructorInitializerList>  <constructorInitializer>  <mulExpr type="int">  <Var type="int" scope="param">a</Var>  <Var type="int" scope="param">b</Var>  </mulExpr>  </constructorInitializer>  <constructorInitializer>  <plusExpr type="int">  <Var type="int" scope="param">a</Var>  <Var type="int" scope="param">b</Var>  </plusExpr>  </constructorInitializer>  <constructorInitializer>  <intConstant type="int">0</intConstant>  </constructorInitializer>  </constructorInitializerList>  <body>  <compoundStatement>  <symbols/>  <declarations/>  </compoundStatement>  </body>  </functionDefinition> |

課題： 現状では、<constructorInitializer> は出現順にならぶだけであり、どのメンバに対する初期化なのかが出力されていない。

### テスト03: 二項と単項の「＊」演算子オーバーロードのテスト

入力ソースコードでの演算子オーバーロード（二項演算子としての＊、すなわち乗算演算子と、単項演算子としての＊、すなわち間接演算子）の宣言が、typeTable中の該当するclassType内で適切な名前を持つ<operator>として生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| class A {  public:  （略）  int operator\*() {  return value;  }  A operator\*(A other);  （以下略） | <classType type="Class0">  <name fullName="A">A</name>  <symbols>  <id type="Function0" sclass="extern\_def" access="public"/>  <id type="Function1" sclass="extern\_def" access="public">  <operator fullName="A::operator\*">pointerRef</operator>  </id>  <id type="Function2" sclass="extern\_def" access="public">  <operator fullName="A::operator\*">mulExpr</operator>  </id>  <id type="int" access="private">  <name fullName="A::value">value</name>  </id>  </symbols>  </classType> |

### テスト04: クラス継承のテスト

入力ソースコードでのクラス定義で各種の継承を記述したものが、typeTable中の該当するclassType内で適切な<inheritedFrom>として生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| class A {};  class B {}; | <classType type="Class0">  <name fullName="A">A</name>  <symbols/>  </classType>  <classType type="Class1">  <name fullName="B">B</name>  <symbols/>  </classType> |
| class D : A {}; | <classType type="Class2">  <name fullName="D">D</name>  <inheritedFrom>  <typeName ref="Class0" access="private"/>  </inheritedFrom>  <symbols/>  </classType> |
| class PD1 : public A {}; | <classType type="Class3">  <name fullName="PD1">PD1</name>  <inheritedFrom>  <typeName ref="Class0" access="public"/>  </inheritedFrom>  <symbols/>  </classType> |
| class PD2 : private A {}; | <classType type="Class4">  <name fullName="PD2">PD2</name>  <inheritedFrom>  <typeName ref="Class0" access="private"/>  </inheritedFrom>  <symbols/>  </classType> |
| class PD3 : protected A {}; | <classType type="Class5">  <name fullName="PD3">PD3</name>  <inheritedFrom>  <typeName ref="Class0" access="protected"/>  </inheritedFrom>  <symbols/>  </classType> |
| class MD1 : A, B {}; | <classType type="Class6">  <name fullName="MD1">MD1</name>  <inheritedFrom>  <typeName ref="Class0" access="private"/>  <typeName ref="Class1" access="private"/>  </inheritedFrom>  <symbols/>  </classType> |
| class MD2 : public PD1,  private PD2,  protected PD3 {}; | <classType type="Class7">  <name fullName="MD2">MD2</name>  <inheritedFrom>  <typeName ref="Class3" access="public"/>  <typeName ref="Class4" access="private"/>  <typeName ref="Class5" access="protected"/>  </inheritedFrom>  <symbols/>  </classType> |
| class VD1 : virtual A {}; | <classType type="Class8">  <name fullName="VD1">VD1</name>  <inheritedFrom>  <typeName ref="Class0" access="private" is\_virtual="1"/>  </inheritedFrom>  <symbols/>  </classType> |
| class VD2 : virtual public A {}; | <classType type="Class9">  <name fullName="VD2">VD2</name>  <inheritedFrom>  <typeName ref="Class0" access="public" is\_virtual="1"/>  </inheritedFrom>  <symbols/>  </classType> |
| class VD3 : virtual private A {}; | <classType type="Class10">  <name fullName="VD3">VD3</name>  <inheritedFrom>  <typeName ref="Class0" access="private" is\_virtual="1"/>  </inheritedFrom>  <symbols/>  </classType> |
| class VD4 : virtual protected A {}; | <classType type="Class11">  <name fullName="VD4">VD4</name>  <inheritedFrom>  <typeName ref="Class0" access="protected" is\_virtual="1"/>  </inheritedFrom>  <symbols/>  </classType> |
| class CD : VD1,  public VD2,   virtual protected D,  MD1 {}; | <classType type="Class12">  <name fullName="CD">CD</name>  <inheritedFrom>  <typeName ref="Class8" access="private"/>  <typeName ref="Class9" access="public"/>  <typeName ref="Class2" access="protected" is\_virtual="1"/>  <typeName ref="Class6" access="private"/>  </inheritedFrom>  <symbols/>  </classType> |

### テスト05: 各種の演算子オーバーロードのテスト

入力ソースコードでのクラス内演算子オーバーロードがtypeTable内で適切に扱われること、およびトップレベルでの演算子オーバーロードがglobalDeclarationsで適切に扱われることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| class Ratio {  public:  （略）  Ratio operator++(int);  Ratio operator++() {  a += b;  return \*this;  }  Ratio operator--(int);  Ratio operator--() {  a -= b;  return \*this;  }  （以下略） | <classType type="Class0">  <name fullName="Ratio">Ratio</name>  <symbols>  （略）  <id type="Function2" sclass="extern\_def" access="public">  <operator fullName="Ratio::operator++">postIncrExpr</operator>  </id>  <id type="Function3" sclass="extern\_def" access="public">  <operator fullName="Ratio::operator++">preIncrExpr</operator>  </id>  <id type="Function2" sclass="extern\_def" access="public">  <operator fullName="Ratio::operator--">postDecrExpr</operator>  </id>  <id type="Function3" sclass="extern\_def" access="public">  <operator fullName="Ratio::operator--">preIncrExpr</operator>  </id>  （略）  </symbols>  </classType> |
| 上記のメンバ関数のうち、operator++()の定義について、 globalDeclarations内に右記の部分が生成される | <functionDefinition>  <operator fullName="Ratio::operator++">preIncrExpr</operator>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <exprStatement>  <asgPlusExpr type="int">  <memberRef type="int" member="a">  <thisExpr/>  </memberRef>  <memberRef type="int" member="b">  <thisExpr/>  </memberRef>  </asgPlusExpr>  </exprStatement>  <returnStatement>  <Stmt\_CXXConstructExprClass>  <pointerRef type="Class0">  <thisExpr/>  </pointerRef>  </Stmt\_CXXConstructExprClass>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |
| 上記のメンバ関数のうち、operator--()の定義について、 globalDeclarations内に右記の部分が生成される | <functionDefinition>  <operator fullName="Ratio::operator--">preIncrExpr</operator>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <exprStatement>  <asgMinusExpr type="int">  <memberRef type="int" member="a">  <thisExpr/>  </memberRef>  <memberRef type="int" member="b">  <thisExpr/>  </memberRef>  </asgMinusExpr>  </exprStatement>  <returnStatement>  <Stmt\_CXXConstructExprClass>  <pointerRef type="Class0">  <thisExpr/>  </pointerRef>  </Stmt\_CXXConstructExprClass>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |
| Ratio Ratio::operator++(int) {  Ratio self(\*this);  a += b;  return self;  }  （次にRatio Ratio::operator--(int)についても同様の内容が生成されるが、省略） | <functionDefinition>  <NestedNameSpecifier\_TypeSpec/>  <operator fullName="Ratio::operator++">postIncrExpr</operator>  <symbols>  <id type="int" sclass="param"/>  </symbols>  <params/>  <name type="int"></name>  <body>  <compoundStatement>  <symbols>  <id type="Class0" sclass="auto">  <name>self</name>  </id>  </symbols>  <declarations>  <varDecl>  <name>self</name>  <value>  <Stmt\_CXXConstructExprClass>  <pointerRef type="Class0">  <thisExpr/>  </pointerRef>  </Stmt\_CXXConstructExprClass>  </value>  </varDecl>  </declarations>  <body>  <exprStatement>  <asgPlusExpr type="int">  <memberRef type="int" member="a">  <thisExpr/>  </memberRef>  <memberRef type="int" member="b">  <thisExpr/>  </memberRef>  </asgPlusExpr>  </exprStatement>  <returnStatement>  <Stmt\_CXXConstructExprClass>  <Var type="Class0" scope="local">self</Var>  </Stmt\_CXXConstructExprClass>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |
| Ratio operator-(const Ratio& a) {  return Ratio(-a.numer(), a.denom());  } | <functionDefinition>  <operator fullName="operator-">unaryMinusExpr</operator>  <symbols>  <id type="Pointer3" sclass="param">  <name>a</name>  </id>  </symbols>  <params/>  <name type="Pointer3">a</name>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <Stmt\_CXXConstructExprClass>  <Stmt\_MaterializeTemporaryExprClass>  <Stmt\_CXXTemporaryObjectExprClass>  <unaryMinusExpr type="int">  <functionCall type="int">  <memberRef   type="\_bound\_member\_function\_type\_"  member="numer">  <varAddr type="Pointer2"  scope="param">a</varAddr>  </memberRef>  </functionCall>  </unaryMinusExpr>  <functionCall type="int">  <memberRef  type="\_bound\_member\_function\_type\_"  member="denom">  <varAddr type="Pointer2" scope="param">a</varAddr>  </memberRef>  </functionCall>  </Stmt\_CXXTemporaryObjectExprClass>  </Stmt\_MaterializeTemporaryExprClass>  </Stmt\_CXXConstructExprClass>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |
| Ratio operator-(const Ratio& a,  const Ratio& b) {  return a + (-b);  } | <functionDefinition>  <operator fullName="operator-">minusExpr</operator>  <symbols>  <id type="Pointer3" sclass="param">  <name>a</name>  </id>  <id type="Pointer3" sclass="param">  <name>b</name>  </id>  </symbols>  <params/>  <name type="Pointer3">a</name>  <name type="Pointer3">b</name>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <Stmt\_CXXConstructExprClass>  <Stmt\_MaterializeTemporaryExprClass>  <functionCall>  <operator>plusExpr</operator>  <Var type="Class1" scope="param">a</Var>  <Stmt\_MaterializeTemporaryExprClass>  <functionCall>  <operator>unaryMinusExpr</operator>  <Var type="Class1" scope="param">b</Var>  </functionCall>  </Stmt\_MaterializeTemporaryExprClass>  </functionCall>  </Stmt\_MaterializeTemporaryExprClass>  </Stmt\_CXXConstructExprClass>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |

課題：

* いくつかのClang ASTが、対応するXcodeML表記をもたないのでClang ASTの都合で決めた名前のXML要素として生成されている。
* <memberRef type="\_bound\_member\_function\_type\_" member="…”> のような型がついている部分がある。XcodeML的にはそもそも「メンバ関数呼び出しを意味するfunctionCallのときのmemberRefにtypeをつけるべきなのか」という問題が定まっていない。（そもそもXcodeML for CのfunctionCallには関数の型の情報が必要なかったが、C++では関数オーバーロードがあるので型の情報も必要）

### テスト06: 構造体のメンバ利用のテスト

入力ソースコードで構造体（structキーワードを用いる形）で定義されたメンバの利用（メンバ変数の利用およびメンバ関数の呼び出し）がglobalDeclarations内でmemberRefとして生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| int main() {  struct A a;  a.x = 0;  return a.f();  } | <functionDefinition>  <name fullName="main">main</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols>  <id type="Struct0" sclass="auto">  <name>a</name>  </id>  </symbols>  <declarations>  <varDecl>  <name>a</name>  <value>  <Stmt\_CXXConstructExprClass/>  </value>  </varDecl>  </declarations>  <body>  <exprStatement>  <assignExpr type="int">  <memberRef type="int" member="x">  <varAddr type="Pointer0" scope="local">a</varAddr>  </memberRef>  <intConstant type="int">0</intConstant>  </assignExpr>  </exprStatement>  <returnStatement>  <functionCall type="int">  <memberRef type="\_bound\_member\_function\_type\_" member="f">  <varAddr type="Pointer0" scope="local">a</varAddr>  </memberRef>  </functionCall>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |

課題：テスト05のところで指摘したように、メンバ関数の型のところが\_bound\_member\_function\_type\_ となっている。

### テスト07: 同じ型を持つ関数が複数宣言される場合のテスト

入力ソースコードで同一の型を持つ関数を記述した場合に、typeTableが適切に生成されてglobalSymbols等で適切に参照されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコード全体 | 左記の部分に対応するXcodeML |
| int f(int x, int y);  int h(int x, int y) {  return x \* y;  } | <typeTable>  <pointerType type="Pointer0" ref="int"/>  <functionType type="Function0" return\_type="int">  <params>  <name type="int">x</name>  <name type="int">y</name>  </params>  </functionType>  </typeTable>  <globalSymbols>  <id type="Function0" sclass="extern\_def">  <name fullName="f">f</name>  </id>  <id type="Function0" sclass="extern\_def">  <name fullName="h">h</name>  </id>  </globalSymbols>  <globalDeclarations>  <functionDecl>  <name fullName="f">f</name>  </functionDecl>  <functionDefinition>  <name fullName="h">h</name>  <symbols>  <id type="int" sclass="param">  <name>x</name>  </id>  <id type="int" sclass="param">  <name>y</name>  </id>  </symbols>  <params>  <name type="int">x</name>  <name type="int">y</name>  </params>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <mulExpr type="int">  <Var type="int" scope="param">x</Var>  <Var type="int" scope="param">y</Var>  </mulExpr>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </globalDeclarations> |

### テスト08: 正規化可能な単純なクラス宣言のテスト

入力ソースコードで単純なクラスを宣言した場合に、クラス内で定義されたメンバ関数が適切に外部化（すなわちクラス宣言の正規化）がなされて、typeTableとglobalDeclarationsがそれぞれ適切に生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコード全体 | 左記の部分に対応するXcodeML |
| class A {  int x;  void f() {  return;  }  int operator\*() {  return x;  }  }; | <typeTable>  <pointerType type="Pointer0" ref="Class0"/>  <classType type="Class0">  <name fullName="A">A</name>  <symbols>  <id type="int" access="private">  <name fullName="A::x">x</name>  </id>  <id type="Function0" sclass="extern\_def" access="private">  <name fullName="A::f">f</name>  </id>  <id type="Function1" sclass="extern\_def" access="private">  <operator fullName="A::operator\*">pointerRef</operator>  </id>  </symbols>  </classType>  （略）  </typeTable>  <globalSymbols/>  <globalDeclarations>  <classDecl type="Class0"/>  <functionDefinition>  <name fullName="A::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement/>  </body>  </compoundStatement>  </body>  </functionDefinition>  <functionDefinition>  <operator fullName="A::operator\*">pointerRef</operator>  <symbols/>  <params/>  <body>  （略）  </body>  </functionDefinition>  </globalDeclarations> |

### テスト09: namespaceやクラスで囲われた宣言のテスト

入力ソースコードでのnamespace内の宣言やクラス内クラスの宣言など、名前空間による名前の修飾に関連した宣言に対して、typeTableやglobalDeclarations内に適切なfullName属性が生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| namespace A {  int x;  enum B {  E, N, U, M  };  } | （<typeTable>内に）  <enumType type="Enum0">  <symbols>  <id>  <name fullName="A::B::E">E</name>  </id>  <id>  <name fullName="A::B::N">N</name>  </id>  <id>  <name fullName="A::B::U">U</name>  </id>  <id>  <name fullName="A::B::M">M</name>  </id>  </symbols>  </enumType>  （globalDeclarations内に）  <varDecl>  <name fullName="A::x">x</name>  </varDecl> |
| namespace B {  class C {  public:  int x;  };  const int override = 108;  int f() {  int answer = 42;  return answer;  }  }; | （<typeTable>内に）  <classType type="Class0">  <name fullName="B::C">C</name>  <symbols>  <id type="int" access="public">  <name fullName="B::C::x">x</name>  </id>  </symbols>  </classType>  （globalDeclarations内に）  <varDecl>  <name fullName="B::override">override</name>  <value>  <intConstant type="int">108</intConstant>  </value>  </varDecl>  <functionDefinition>  <name fullName="B::f">f</name>  <symbols/>  <params/>  <body>  （略）  </body>  </functionDefinition> |
| class D {  class E {  public:  int y;  };  }; | （<typeTable>内に）  <classType type="Class1">  <name fullName="D">D</name>  <symbols/>  </classType>  <classType type="Class2">  <name fullName="D::E">E</name>  <symbols>  <id type="int" access="public">  <name fullName="D::E::y">y</name>  </id>  </symbols>  </classType>  （globalDeclarations内に： ネストされたクラスは正規化されないので出力される）  <classDecl type="Class1">  <classDecl type="Class2">  <id type="int">  <name fullName="D::E::y">y</name>  </id>  </classDecl>  </classDecl> |
| namespace F {  namespace G {  class H;  }  }  class F::G::H { }; | （<typeTable>内に）  <classType type="Class3">  <name fullName="F::G::H">H</name>  <symbols/>  </classType> |

### テスト10: 初期化リストのテスト

入力ソースコードでの初期化子リストを使った配列初期化構文に対して、globalDeclarations内のvarDecl部に適切なvalue要素が生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| int a[] = {1,2,3,4}; | <varDecl>  <name>a</name>  <value>  <intConstant type="int">1</intConstant>  <intConstant type="int">2</intConstant>  <intConstant type="int">3</intConstant>  <intConstant type="int">4</intConstant>  </value>  </varDecl>  </declarations> |

### テスト11: クラスでのメンバ関数呼び出しのテスト

入力ソースコードでの無名クラス宣言とそのメンバ関数呼び出しに対して、globalDeclarations部が適切に生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコード全体 | 左記の部分に対応するXcodeML |
| class {  public:  int f() { return 0; }  } a;  int g() {  return a.f();  } | <globalDeclarations>  <classDecl type="Class0">  <functionDefinition>  <name fullName="(anonymous class)::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">0</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </classDecl>  <varDecl>  <name fullName="a">a</name>  <value>  <Stmt\_CXXConstructExprClass/>  </value>  </varDecl>  <functionDefinition>  <name fullName="g">g</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <functionCall type="int">  <memberRef type="\_bound\_member\_function\_type\_" member="f">  <varAddr type="Pointer0" scope="global">a</varAddr>  </memberRef>  </functionCall>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </globalDeclarations> |

課題：Stmt\_CXXConstructorExprClassと\_bound\_member\_function\_type の部分（テスト05と同様の問題）。

### テスト12: 複数の名前空間での同名の関数定義のテスト

入力ソースコードで同名の関数を複数の名前空間内で定義した場合に、globalDeclarationsが適切に生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| namespace {  int f() {  return 42;  }  } | <Decl\_Namespace>  <functionDefinition>  <name fullName="(anonymous namespace)::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">42</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </Decl\_Namespace> |
| namespace A {  int f() {  return 52;  }  } | <Decl\_Namespace>  <functionDefinition>  <name fullName="A::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">52</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </Decl\_Namespace> |
| inline namespace B {  int f() {  return 62;  }  } | <Decl\_Namespace>  <functionDefinition>  <name fullName="B::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">62</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </Decl\_Namespace> |
| namespace {  namespace {  int f() {  return 72;  }  }  } | <Decl\_Namespace>  <Decl\_Namespace>  <functionDefinition>  <name fullName="(anonymous namespace)::(anonymous namespace)::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">72</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </Decl\_Namespace>  </Decl\_Namespace> |
| int f() {  return 82;  } | <functionDefinition>  <name fullName="f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">82</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |

課題：もともとのソースコード上にnamespace という構文が出現した場所をDecl\_Namespaceとして再現しているが、正規化という意味ではこの再現は不要。

### テスト13: 複雑なクラス宣言のテスト

入力ソースコードでのクラス宣言がコンストラクタのdefault, deleteやfinal指定などを含む場合にも正規化が正しくおこなわれてtypeTable部とglobalDeclarations部がそれぞれ生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| class A {  A() = default;  A(A &a) = delete;  virtual int f() = 0;  void g() { return; }  protected:  const int x;  static int a;  const static int b = 42;  }; | （<typeTable>内に）  <classType type="Class0">  <name fullName="A">A</name>  <symbols>  <id type="Function0" sclass="extern\_def" access="private"/>  <id type="Function1" sclass="extern\_def" access="private"/>  <id type="Function2" sclass="extern\_def" access="private">  <name fullName="A::f">f</name>  </id>  <id type="Function0" sclass="extern\_def" access="private">  <name fullName="A::g">g</name>  </id>  <id type="Basic0" access="protected">  <name fullName="A::x">x</name>  </id>  <id type="int" sclass="static">  <name fullName="A::a">a</name>  </id>  <id type="Basic0" sclass="static">  <name fullName="A::b">b</name>  </id>  </symbols>  </classType>  （<globalDeclarations>内に： 正規化されたので<classDecl>はプレイスホルダだけになっている）  <classDecl type="Class0"/>  <functionDefinition>  <constructor/>  <symbols/>  <params/>  <body/>  </functionDefinition>  <functionDefinition>  <constructor/>  <symbols>  <id type="Pointer0" sclass="param">  <name>a</name>  </id>  </symbols>  <params>  <name type="Pointer0">a</name>  </params>  <body/>  </functionDefinition>  <functionDecl>  <name fullName="A::f">f</name>  </functionDecl>  <functionDefinition>  <name fullName="A::g">g</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement/>  </body>  </compoundStatement>  </body>  </functionDefinition> |
| class B final : public A {  int f() override;  }; | （<typeTable>内に）  <classType type="Class1">  <name fullName="B">B</name>  <inheritedFrom>  <typeName ref="Class0" access="public"/>  </inheritedFrom>  <symbols>  <id type="Function2" sclass="extern\_def" access="private">  <name fullName="B::f">f</name>  </id>  <gccAttribute name="final"/>  </symbols>  </classType>  （<globalDeclarations>内に： 正規化されたので<classDecl>はプレイスホルダだけになっている）  <classDecl type="Class1"/>  <functionDecl>  <name fullName="B::f">f</name>  </functionDecl> |
| int B::f() {  return x;  } | <functionDefinition>  <NestedNameSpecifier\_TypeSpec/>  <name fullName="B::f">f</name>  <symbols>  <gccAttribute name="override"/>  </symbols>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <memberRef type="Basic0" member="x">  <thisExpr/>  </memberRef>  </returnStatement>  </body>  </compoundStatement>  <gccAttribute name="override"/>  </body>  </functionDefinition> |
| int A::a = 1; | <varDecl>  <name fullName="A::a">a</name>  <NestedNameSpecifier\_TypeSpec/>  <value>  <intConstant type="int">1</intConstant>  </value>  </varDecl> |

課題：

* コンストラクタのdefault指定やdelete指定についての情報の扱い方が定まっていないので、XcodeMLに反映されていない。
* final指定やoverride指定がClang ASTではAttrというクラスで扱われており、これはgccの \_\_attribute\_\_ と同じ扱いであるため、現状のCtoXcodeMLでは <gccAttribute> 要素を生成している。

### テスト14: ネストされたクラスの先行宣言のテスト

入力ソースコードでのネストされたクラスの先行宣言に対してtypeTableとglobalDeclarationsにそれぞれ生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコード全体 | 左記の部分に対応するXcodeML |
| class A {  class B;  };  class A::B { }; | <typeTable>  <classType type="Class0">  <name fullName="A">A</name>  <symbols/>  </classType>  <classType type="Class1">  <name fullName="A::B">B</name>  <symbols/>  </classType>  </typeTable>  <globalSymbols/>  <globalDeclarations>  <classDecl type="Class0">  <classDecl type="Class1"/>  </classDecl>  <classDecl type="Class1">  <NestedNameSpecifier\_TypeSpec/>  </classDecl>  </globalDeclarations> |

課題：<NestedNameSpecifier\_TypeSpec/> が無駄に生成されている。

### テスト15: const修飾版と非修飾版のメンバ関数定義のテスト

入力ソースコードで同じ名前のメンバ関数のconst修飾版と非修飾版をともに定義したものに対して、typeTableが生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| class A {  public:  int f();  int f() const;  private:  int x;  }; | <classType type="Class0">  <name fullName="A">A</name>  <symbols>  <id type="Function0" sclass="extern\_def" access="public">  <name fullName="A::f">f</name>  </id>  <id type="Function1" sclass="extern\_def" access="public">  <name fullName="A::f">f</name>  </id>  <id type="int" access="private">  <name fullName="A::x">x</name>  </id>  </symbols>  </classType>  （略）  <functionType type="Function0" return\_type="int">  <params/>  </functionType>  <functionType type="Function1" return\_type="int">  <params/>  </functionType> |

課題：Function0型・Function1型がともに同じ型になっており、const修飾が表現されていない。メンバ関数のconst修飾を表現するためのXcodeML記法を決める必要がある。

### テスト16: 複雑な型を持つ関数のテスト

入力ソースコードでの複雑な型の関数（関数ポインタを受け取り関数ポインタを返す関数）に対して適切なtypeTableが生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコード全体 | 左記の部分に対応するXcodeML |
| double (\*f(int (\*p)(int))) (double) {  return (double(\*)(double))f;  } | <typeTable>  <pointerType type="Pointer0" ref="Function0"/>  <pointerType type="Pointer1" ref="Function2"/>  <pointerType type="Pointer2" ref="Function1"/>  <functionType type="Function0" return\_type="double"/>  <functionType type="Function1" return\_type="Pointer0">  <params>  <name type="double"></name>  <name type="Pointer1">p</name>  <name type="double"></name>  </params>  </functionType>  <functionType type="Function2" return\_type="int"/>  </typeTable>  <globalSymbols>  <id type="Function1" sclass="extern\_def">  <name fullName="f">f</name>  </id>  </globalSymbols>  <globalDeclarations>  <functionDefinition>  <name fullName="f">f</name>  <symbols>  <id type="double" sclass="param"/>  <id type="Pointer1" sclass="param">  <name>p</name>  </id>  </symbols>  <params>  <name type="double"></name>  <name type="Pointer1">p</name>  </params>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <castExpr type="Pointer0">  <name type="double"></name>  <Var type="Function1">f</Var>  </castExpr>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </globalDeclarations> |

課題：関数ポインタの型や引数の個数が正しく反映されていない。

### テスト17: ユーザー定義リテラルのテスト

入力ソースコードでのユーザー定義リテラルのオーバーロード定義に対して、適切なglobalDeclarationsが生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| int operator"" \_times(unsigned long long x) {  return x;  } | <functionDefinition>  <name fullName="operator &quot;&quot; \_times"/>  <symbols>  <id type="unsigned\_long\_long" sclass="param">  <name>x</name>  </id>  </symbols>  <params>  <name type="unsigned\_long\_long">x</name>  </params>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <Var type="unsigned\_long\_long" scope="param">x</Var>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition> |

課題：name要素の内容が空になっており、fullName属性部に &quot;&quot; という文字列を含む名前がついている。そもそもこれは演算子オーバーロードの一部とみなすべきかもしれないのでname要素ではなくoperator要素の方が近いが、operator要素に任意の文字列を含むことは想定されていないので、さらに新しい要素（userDefinedLiteralOperatorといった名称）を準備した方がよいかもしれない。

### テスト18: 深くネストされたnamespaceのテスト

入力ソースコードでのnamespaceのネストに対してtypeTableやglobalDeclarationsが適切に生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| namespace {  namespace A {  namespace {  namespace B {  namespace {  class K {  int f();  };  }  }  }  }  } | （<typeTable>内に）  <classType type="Class0">  <name fullName="(anonymous namespace)::A::(anonymous namespace)::B::(anonymous namespace)::K">K</name>  <symbols>  <id type="Function0" sclass="extern\_def" access="private">  <name fullName="(anonymous namespace)::A::(anonymous namespace)::B::(anonymous namespace)::K::f">f</name>  </id>  </symbols>  </classType>  （<globalDeclarations>内に）  <Decl\_Namespace>  <Decl\_Namespace>  <Decl\_Namespace>  <Decl\_Namespace>  <Decl\_Namespace>  <classDecl type="Class0"/>  <functionDecl>  <name fullName="(anonymous namespace)::A::(anonymous namespace)::B::(anonymous namespace)::K::f">f</name>  </functionDecl>  </Decl\_Namespace>  </Decl\_Namespace>  </Decl\_Namespace>  </Decl\_Namespace>  </Decl\_Namespace> |
| namespace A {  class K {  int f();  };  } | （<typeTable>内に）  <classType type="Class1">  <name fullName="A::K">K</name>  <symbols>  <id type="Function0" sclass="extern\_def" access="private">  <name fullName="A::K::f">f</name>  </id>  </symbols>  </classType>  （<globalDeclarations>内に）  <Decl\_Namespace>  <classDecl type="Class1"/>  <functionDecl>  <name fullName="A::K::f">f</name>  </functionDecl>  </Decl\_Namespace> |

### テスト19: 正規化ができないクラス宣言のテスト

入力ソースコードで正規化ができないクラス宣言（ネストされたクラスや関数内クラスなど）に対し、globalDeclarationsにAST構造の全てが生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコード一部 | 左記の部分に対応するXcodeML |
| class {  class A {  int f() { return 0; }  };  } b; | <classDecl type="Class0">  <functionDefinition>  <name fullName="(anonymous class)::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">0</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </classDecl> |
| （main関数内で）  class B {  int x;  }; | （<functionDefinition> <name fullName="main">main</name>の内部で）  <classDecl type="Class3">  <id type="int">  <name fullName="main()::B::x">x</name>  </id>  </classDecl> |
| （main関数内で）  class C {  class {  int f() { return 0; }  } x;  }; | （<functionDefinition> <name fullName="main">main</name>の内部で）  <classDecl type="Class4">  <classDecl type="Class5">  <functionDefinition>  <name fullName="main()::C::(anonymous class)::f">f</name>  <symbols/>  <params/>  <body>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <returnStatement>  <intConstant type="int">0</intConstant>  </returnStatement>  </body>  </compoundStatement>  </body>  </functionDefinition>  </classDecl>  <id type="Class5">  <name fullName="main()::C::x">x</name>  </id>  </classDecl> |

### テスト20: 初期化リストによるコンストラクタ呼び出しのテスト

入力ソースコードでの初期化子リスト記法を用いる形でのコンストラクタ呼び出しに対して初期化構文形式でのvarDeclが生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| A a = {10}; | <varDecl>  <name>a</name>  <value>  <intConstant type="int">10</intConstant>  </value>  </varDecl> |

### テスト21: オーバーロード演算子の呼び出しのテスト

入力ソースコードでのオーバーロードされた演算子を用いた式に対してfunctionCallが適切に生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| a += 3; | <exprStatement>  <functionCall>  <operator>asgPlusExpr</operator>  <Stmt\_CXXConstructExprClass>  <Var type="Struct0" scope="local">a</Var>  </Stmt\_CXXConstructExprClass>  <intConstant type="int">3</intConstant>  </functionCall>  </exprStatement> |

課題：functionCall要素内にoperator要素は適切に生成されているが、左辺の a がStmt\_CXXConstructExprClassという値になっている。左辺値の扱いはXcodeMLの不完全な部分なので根本的な改善が必要であると考えられる。

### テスト22: C++スタイルのキャストのテスト

入力ソースコードでの各種のC++スタイルキャストに対してそれぞれconstCast, staticCast, reinterpretCast, dynamicCast要素が生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| int\* pi = const\_cast<int\*>(&ci); | <varDecl>  <name>pi</name>  <value>  <constCast type="Pointer1">  <varAddr type="Pointer2" scope="local">ci</varAddr>  </constCast>  </value>  </varDecl> |
| double d = static\_cast<double>(ci); | <varDecl>  <name>d</name>  <value>  <staticCast type="double">  <Var type="Basic0" scope="local">ci</Var>  </staticCast>  </value>  </varDecl> |
| void\* pv = reinterpret\_cast<void\*>(ci); | <varDecl>  <name>pv</name>  <value>  <reinterpretCast type="Pointer3">  <Var type="Basic0" scope="local">ci</Var>  </reinterpretCast>  </value>  </varDecl> |
| Derived\* pd = dynamic\_cast<Derived\*>(pb); | <varDecl>  <name>pd</name>  <value>  <dynamicCast type="Pointer4">  <Var type="Pointer0" scope="param">pb</Var>  </dynamicCast>  </value>  </varDecl> |

### テスト23: メンバへのポインタのテスト

入力ソースコードでのメンバへのポインタ演算子に対してmemterPointerRef要素が生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| int A::\* p = &A::x; | <varDecl>  <name>p</name>  <value>  <NestedNameSpecifier\_TypeSpec/>  <varAddr type="Pointer1">x</varAddr>  </value>  </varDecl> |
| void (A::\* func)() = &A::f; | <varDecl>  <name>func</name>  <value>  <NestedNameSpecifier\_TypeSpec/>  <varAddr type="Pointer3">f</varAddr>  </value>  </varDecl> |
| (a.\*p)++; | <exprStatement>  <postIncrExpr type="int">  <memberPointerRef type="int">  <Var type="Struct0" scope="local">a</Var>  <Var type="Pointer0" scope="local">p</Var>  </memberPointerRef>  </postIncrExpr>  </exprStatement> |
| (a.\*func)(); | <exprStatement>  <functionCall type="void">  <memberPointerRef type="\_bound\_member\_function\_type\_">  <varAddr type="Pointer4" scope="local">a</varAddr>  <Var type="Pointer2" scope="local">func</Var>  </memberPointerRef>  </functionCall>  </exprStatement> |

### テスト24: thisポインタを使った式のテスト

入力ソースコードでthisを用いた式に対してthisExpr要素が生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| return \*this; | <returnStatement>  <Stmt\_CXXConstructExprClass>  <pointerRef type="Class0">  <thisExpr/>  </pointerRef>  </Stmt\_CXXConstructExprClass>  </returnStatement> |

課題：返却値がStmt\_CXXConstructExprClass となっている。

### テスト25: tryとcatchのテスト

入力ソースコードでの例外処理構文（try, throw, catch）に対してtryStatement要素、throwStatement要素、catchStatement要素が生成されることを確認するテストである。

|  |  |
| --- | --- |
| 入力ソースコードの一部 | 左記の部分に対応するXcodeML |
| try {  A a;  throw a;  } catch (A a) {  a.x = 0;  } | <tryStatement>  <compoundStatement>  <symbols>  <id type="Struct0" sclass="auto">  <name>a</name>  </id>  </symbols>  <declarations>  <varDecl>  <name>a</name>  <value>  <Stmt\_CXXConstructExprClass/>  </value>  </varDecl>  </declarations>  <body>  <exprStatement>  <throwStatement/>  </exprStatement>  </body>  </compoundStatement>  <catchStatement>  <varDecl>  <name>a</name>  </varDecl>  <compoundStatement>  <symbols/>  <declarations/>  <body>  <exprStatement>  <assignExpr type="int">  <memberRef type="int" member="x">  <varAddr type="Pointer0" scope="local">a</varAddr>  </memberRef>  <intConstant type="int">0</intConstant>  </assignExpr>  </exprStatement>  </body>  </compoundStatement>  </catchStatement>  </tryStatement> |