「Omni XcalableMPコンパイラのC++フロントエンドおよびデコンパイラの開発」報告書

2016年9月29日

株式会社トランス・ニュー・テクノロジー

内容

[2 本報告書の構成 2](#_Toc462918603)

[3 納品ソフトウェアの利用方法 3](#_Toc462918604)

[3.1 gitリポジトリの構成 3](#_Toc462918605)

[3.2 ビルド手順 5](#_Toc462918606)

[3.2.1 CtoXcodeMLのビルド 7](#_Toc462918607)

[3.2.2 CXXtoXMLのビルド 8](#_Toc462918608)

[3.2.3 XcodeMLtoCXXのビルド 8](#_Toc462918609)

[3.3 ビルドされたツールの利用方法 9](#_Toc462918610)

[3.3.1 CtoXcodeMLの利用方法 9](#_Toc462918611)

[3.3.2 CXXtoXMLの利用方法 10](#_Toc462918612)

[3.3.3 XcodeMLtoCXXの利用方法 10](#_Toc462918613)

[4 作業報告 11](#_Toc462918614)

[4.1 納品ソフトウェアの内部実装に関する解説 11](#_Toc462918615)

[4.1.1 正変換ツールの各ソースコードについて 11](#_Toc462918616)

[4.1.2 逆変換ツールの各ソースコードについて 12](#_Toc462918617)

[4.2 テスト用ソースコードによる評価 14](#_Toc462918618)

[4.2.1 UnitTest/ サブディレクトリ内のテストについて 15](#_Toc462918619)

[4.2.2 Compatibility/ サブディレクトリ内のテストについて 15](#_Toc462918620)

[4.2.3 Compatibility/ サブディレクトリ中のCソースコード 16](#_Toc462918621)

[4.2.4 Compatibility/ サブディレクトリ中のC++ソースコード 18](#_Toc462918622)

# 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（ClangXcodeML）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
  + CtoXcodeMLreport-20160929.docx  
    本報告書である。
  + XcodeML\_CXX\_1.2.docx  
    現段階でのXcodeML C++ 仕様書である。
  + XcodeMLtoCXX/ サブディレクトリ  
    逆変換ツール（後述）のテストに関する情報であり、本報告書内にも「テスト用ソースコードによる評価」の節の内容として含んでいる。
  + doxygen/ サブディレクトリ  
    空ディレクトリである。  
    CtoXcodeML/src/および XcodeMLtoCXX/ 内でdoxygenを実行したときに、このディレクトリ内にドキュメントが生成される。
  + 上記の他、過去のドキュメントがas-isで含まれている。
    - CtoXcodeMLreport-20150925.docx  
      記述内容のうち、正変換ツール（現在は CtoXcodeML/ ディレクトリに収められているもの）についての基本設計（実装に用いているC++デザインパターンについての記述）については現状でも有効である。
    - CtoXcodeMLreport-20160330.docx  
      記述内容のうち、正変換ツールの動作テスト（現在はCtoXcodeML/testcases/ ディレクトリに収められているもの）についての記述は現状でも有効である。
    - XcodeML\_CXX\_0.2.docx, XcodeML\_CXX\_1.0.pdf, XcodeML\_CXX\_1.1.docx の各ファイル  
      過去の各段階でのXcodeML C++ 仕様書である。
* CtoXcodeML/ ディレクトリ  
  正変換（C/C++ソースコードを入力にとり、対応するXcodeMLを出力する操作）ツールに関するファイル一式を収めている。下記のファイル・サブディレクトリからなる。
  + Makefile  
    下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make clean と make check にのみ対応。
  + src/ サブディレクトリ  
    正変換ツールのソースコード一式を収めている。ここにある Makefile を用いてビルドできる（手順は後述）。
  + testcases/ サブディレクトリ  
    正変換ツールの動作テスト用のC/C++ソースコードと、それぞれについてのCtoXcodeMLによる変換結果を収めている。テスト結果についての報告は前述のCtoXcodeMLreport-20160330.docxに含まれている。
* CXXtoXML/ ディレクトリ  
  前述の正変換ツールを簡素化し、 Clang AST の生の姿に近い XML を出力するためのツールを収めている。このツールは、これまで及び今後の XcodeML の仕様検討のためのたたき台として活用することを目的にしたものであり、 as-is のままで提供される。
* XcodeMLtoCXX/ ディレクトリ  
  逆変換（XcodeMLを入力にとり、対応する C/C++ ソースコードを出力する操作）ツールに関するファイル一式を収めている。下記のファイル・サブディレクトリからなる。
  + Makefile  
    下記サブディレクトリ内のMakefileを再帰的に用いるためのMakefileである。make clean と make check にのみ対応。
  + src/ サブディレクトリ  
    逆変換ツールのソースコード一式を収めている。ここにある Makefile を用いてビルドできる（手順は後述）。
* tests/ サブディレクトリ  
  逆変換ツールの動作テスト用のC/C++ソースコードと、それらに対して正変換→逆変換を通した結果を作成してそれぞれ通常のC/C++コンパイラでコンパイルできることを確認するテストを収めている。テスト結果についての報告は本報告書の「テスト用ソースコードによる評価」の節にある。
* schema/ ディレクトリ  
  XcodeML の仕様を XML schema の記法で表したXcodeML\_CXX.xsdファイルを収めている。

## 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を用いてもビルドと実行ができることは確認できている。3.8.0以降のclang はサポート対象外とするが、 as-is 提供の形で support-clang-3.8.0 ブランチを準備しており、こちらを用いると clang-3.8.0 を用いてもビルドが通ることは確認できている。

下記に Clang のインストール手順を述べる。また、下記のようにしてインストールするかわりに、 docker hub より koyama41/llvm-3.6 のイメージを取得することで同様の環境をdocker を用いて構築可能である。

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](http://llvm.org/releases/download.html" \l "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](http://llvm.org/releases/download.html" \l "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 CtoXcodeML/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。

### CXXtoXMLのビルド

このツールはas-is提供である。

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 CXXtoXML/ ディレクトリ内で make を実行すればビルドが実行され、 CXXtoXMLという名の実行ファイルが生成される。

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XMLRAV.oを残して、その他を消す。

また、 CXXtoXML/XSLTs/ サブディレクトリ内に、 CXXtoXMLが出力したXMLファイルを XSLT プロセッサで加工するためのサンプルが収められている。これらを利用するには、 xsltproc をインストールする必要がある。 Ubuntuであれば  
 % sudo apt-get xsltproc  
のようにしてインストールする。

### XcodeMLtoCXX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 XcodeMLtoCXX/src/ ディレクトリ内で make を実行すればビルドが実行され、 XcodeMLtoCXX/ ディレクトリ直下にXcodeMLtoCXXという名の実行ファイルが生成される。

make cleanをすると、ビルド結果がすべて消される。

## ビルドされたツールの利用方法

### CtoXcodeML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。

### CXXtoXMLの利用方法

このツールはas-is提供であるので、利用法は概要にとどめる。

* コマンドラインオプションの指定は CtoXcodeMLに準じる。  
  ただし、 -file, -lineno, -column オプションは無く、常にこれらすべて（ファイル名、ライン番号、カラム位置）の情報が出力に含まれる。
* 全体は Program というXML要素となり、その子要素として clangASTという要素が生成され、そこにDeclないしStmtクラスの構造を中心としたXML構造が生成される。また、 typeTableはCtoXcodeMLと同様のものが生成される。

### XcodeMLtoCXXの利用方法

XcodeMLtoCXXコマンドにはコマンドラインオプションは無く、入力ファイル（XcodeMLファイル）の名前を一つだけ指定する。結果は標準出力に出される。

# 作業報告

## 納品ソフトウェアの内部実装に関する解説

### 正変換ツールの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。
* 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。

### 逆変換ツールの各ソースコードについて

* LibXMLUtil.h, LibXMLUtil.cpp  
  libxmlを用いてXcodeMLを容易に解析するためのユーティリティライブラリ。
* XMLString.h, XMLString.cpp  
  libxmlの文字列(xmlChar\*)をC++で容易に扱うためのラッパーを定義している部分。
* SourceInfo.h  
  SourceInfoクラスを定義しているヘッダーファイル。  
  入力された XcodeML の構造すべてにアクセスできる XPath コンテキスト情報と、そこから解析された XcodeML::Environment 情報(後述)・SymbolMap情報(後述) を束ねたデータ構造である。また、C/C++プログラムを出力する際に必要となるインデントレベルの情報も格納する。
* Symbol.h  
  SymbolMap型を定義しているヘッダーファイル。  
  SymbolMapは、XcodeMLの表す抽象構文木のある位置から見える名前と、その名前が表す変数の型との対応関係に関する情報を保存している。  
  XcodeMLにおけるname要素はデータ型に関する情報を必ずしも持たないため、SymbolMapはC/C++プログラムを出力する際に変数の型情報を得るのに使われている。
* XcodeMlType.h, XcodeMLType.cpp  
  XcodeMl::Typeクラスを定義している部分。  
  XcodeMl::Typeは、後述するXcodeMl::Environmentと合わせてXcodeMLで定義されるデータ型を表現する。
* XcodeMlEnvironment.h, XcodeMlEnvironment.cpp  
  XcodeMl::Environmentクラスを定義している部分。  
  XcodeMl::Environmentは、データ型識別名と実際のデータ型との対応関係に関する情報を保存している。
* AttrProc.h  
  AttrProcクラステンプレートを定義しているヘッダーファイル。  
  AttrProcを使うことで、与えられたXMLの各要素に対し、その属性(XML attribute)に応じた柔軟な処理を行うことができる。  
  後述するSymbolBuilderで、XcodeMLの<globalSymbols>部の要素を処理するために使われている。
* XMLWalker.h  
  XMLWalkerクラステンプレートを定義しているヘッダーファイル。  
  XMLの各要素を処理する際、要素の種類に合わせて別々の処理を行うことが必要になる場合がある。XMLWalkerはこれを実現する。  
  後述するSymbolAnalyzer、TypeAnalyzer、CodeBuilderでXcodeMLの各部分を処理するために使われている。
* SymbolAnalyzer.h, SymbolAnalyzer.cpp  
  XcodeMLの<globalSymbols>, <symbols>部を解析してプログラム中の名前とそのデータ型との対応関係を管理する部分。
* SymbolBuilder.h, SymbolBuilder.cpp  
  XcodeMLの<globalSymbols>, <symbols>部を解析してC/C++プログラムにおける宣言(グローバル変数宣言、クラス・構造体定義など)部を出力する部分。
* TypeAnalyzer.h, TypeAnalyzer.cpp  
  XcodeMLの<typeTable>部を解析してデータ型識別名と実際のデータ型との対応関係を管理する部分。
* CodeBuilder.h, CodeBuilder.cpp  
  XcodeMLの<globalDeclarations>部を解析してC/C++プログラムを出力する部分。
* XcodeMLtoCXX.cpp  
  main関数部分。コマンドライン引数として与えられたファイル名が表すXcodeML文書を読み、上記各Walkerを用いてC/C++プログラムを出力する。

## テスト用ソースコードによる評価

今回の作業では、CないしC++で記述されたソースコードに対応するXcodeMLについての逆変換ツールの評価を主眼におき、下記のような方針で動作を確認した。

* 正変換の結果を逆変換し、その結果がC/C++コンパイラでコンパイルできるものであるかを確認する。
* 正変換の部分については、ソースコードがC++で記述されたものについては本件で開発された正変換ツールを用いる。ソースコードがCで記述されたものについては、本件で開発された正変換ツールあるいはomni-compilerのC\_Frontツールをのいずれを用いるか選択可能にして、両方で確認する。

テスト用に用いたソースコード一式は、gitレポジトリのXcodeMLtoCXX/tests/ ディレクトリに収められている。UnitTest/ サブディレクトリと Compatibility/ サブディレクトリがあり、前者は逆変換ツールの基本的な挙動を確認している。後者が前述の方針で作成された各種のテストである。

テストの実行には次の手順を踏む。

* git レポジトリのCtoXcodeML/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XcodeMLtoCXX/src/ ディレクトリ内でmakeを実行し、XcodeMLtoCXXの実行バイナリを作成しておく。
* Boostライブラリをインストールしておく。これはUnitTest/ で用いている。
* git レポジトリの XcodeMLtoCXX/ ディレクトリ内で make check を実行する。これは XcodeMLtoCXX/tests/ ディレクトリで make を実行したのと同じである（そこからさらに各サブディレクトリをめぐり、それぞれのサブディレクトリ中でmake checkが実行される）。

以下、各サブディレクトリごとのテスト内容について解説する。

### UnitTest/ サブディレクトリ内のテストについて

XcodeMLtoCXX実装のクラス単位での基本的なテストを担当する。主に、XcodeMLのTypeTableで表現されるさまざまな型情報が正しくハンドルされるかを確認する。この部分だけのテストは次のコマンドで行う。

cd XcodeMLtoCXX/tests/UnitTest

make check

### Compatibility/ サブディレクトリ内のテストについて

このテストはデコンパイラ(XcodeMLtoCXX)が合法(well-formed)なC/C++ソースコードを出力することを検証する。この部分だけのテストは次のコマンドで行う。

cd XcodeMLtoCXX/tests/Compilability

make check

テストではまず、Compilability/以下にあるテスト用ソースコード (そのファイル名は テストの概要.src.c または テストの概要.src.cpp という形式である) をフロントエンド(CtoXcodeML)に入力し、その出力をデコンパイラに入力する。

デコンパイラの出力はCompilability/以下に保存される。 そのファイル名はもとのファイル名の".src."を".dst."に変えたものである。

フロントエンドおよびデコンパイラが正常に終了し、なおかつデコンパイラの出力するC/C++ソースコードが合法であればテストは成功する。 そうでなければ、テストは失敗する。

フロントエンドおよびデコンパイラは意味を変えない範囲でプログラムの正規化を行うことがあるから、 もとの入力(src)と最終的な出力(dst)との単純な文字列比較や、それらの抽象構文木の比較をもって検証を行うことはできない。 このため、最終的な出力がソースコードとして正しいかどうかのみを試験している。

### Compatibility/ サブディレクトリ中のCソースコード

#### array\_cv.src.c

配列型の引数に与えられたCV修飾子および要素数を扱うテスト。

void f(const int a[const 10], double b[const volatile 20]) {

}

#### bitfield.src.c

構造体のビットフィールド宣言の名前とビット長を扱うテスト。

: 5のような無名ビットフィールドはunused fieldを表す。 :0 は直後のビットフィールドが新しいメモリー境界から始まることを保証する。

struct A {

unsigned a : 3, b : 10, : 5, c : 10;

unsigned :0;

double g;

signed d : 15, e : 17;

signed f : 20;

int h : 4;

};

#### fundamental\_type.src.c

基本型(fundamental type)を扱うテスト。 特に、long long, long long int, unsigned intを正しく復元できることを確認する。

char c;

short s;

int i;

long l;

unsigned u;

float f;

double d;

long long ll;

unsigned char uc;

unsigned short us;

unsigned int ui;

unsigned long ul;

unsigned long long ull;

long double ld;

#### several\_var\_in\_single\_decl.src.c

複数の変数を宣言する単一の宣言文を扱うテスト。

int i, j, k,

\*pi, \* const cpi, \* volatile vpi, \* const volatile cvpi,

\*\* ppi, \* const \* pcpi,

a10i[10], \*a5pi[5],

fi(),

(\*pfi)();

#### struct\_in\_struct.src.c

別の構造体定義(struct B)を含む構造体の定義(struct A)を扱うテスト。

struct A {

struct B { int i; } b;

struct B b2;

} ;

#### struct\_type.src.c

単純な構造体定義を扱うテスト。

struct A {

int i;

};

struct B {

double d;

int i;

};

struct C {

char \*pc;

double d;

int i;

};

### Compatibility/ サブディレクトリ中のC++ソースコード

#### cpp\_struct.src.cpp

structキーワードで宣言されたクラスを扱うテスト。

struct A {

int x;

};

class B {

int x;

};

#### cv\_qualifier.src.cpp

引数および自動変数に与えられたCV修飾子を扱うテスト。

void f(int i, int const ci, short \* ps, short \* const cps, short const \* pcs, short const \* const cpcs);

void f(int i, int const ci, short \* ps, short \* const cps, short const \* pcs, short const \* const cpcs) {

double d = 0;

double const cd = 0;

int \* pi = &i;

int const \* pci = &ci;

int \* const cpi = pi;

int const \* const cpci = pci;

}

void g(int i, int volatile ci, short \* ps, short \* volatile cps, short volatile \* pcs, short volatile \* volatile cpcs) {

double d = 0;

double volatile cd = 0;

int \* pi = &i;

int volatile \* pci = &ci;

int \* volatile cpi = pi;

int volatile \* volatile cpci = pci;

}

void h(int i, int const volatile ci, short \* ps, short \* const volatile cps, short const volatile \* pcs, short const volatile \* const volatile cpcs) {

double d = 0;

double const volatile cd = 0;

int \* pi = &i;

int const volatile \* pci = &ci;

int \* const volatile cpi = pi;

int const volatile \* const volatile cpci = pci;

}

#### empty\_do.src.cpp

本体が空のdo文を扱うテスト。

void f() {

do {

} while (0);

}

#### empty\_for.src.cpp

本体が空のfor文を扱うテスト。

void f() {

for (;;) { }

}

#### empty\_while.src.cpp

本体が空のwhile文を扱うテスト。

void f() {

while(1) { }

}

#### for.src.cpp

本体が空でない繰り返し構文(for, while, do)を扱うテスト。

void f() {

int i = 0;

for (i = 0; i - 10; i = i + 1) {

int num = 1;

}

while (1 == 2) {

return;

}

do {

i = i - 1;

} while (i);

}

int square(int x) {

return x \* x;

}

#### function\_decl.src.cpp

関数のプロトタイプ宣言を扱うテスト。

int f(int, int);

#### function\_param\_same\_name.src.cpp

関数名と同じ名前の引数をもつ関数定義を(特に型について)正しく復元できることを確認するテスト。

int f(int f) {

return f;

}

#### int\_return.src.cpp

値を返すreturn文が正しく復元されることを確認するテスト。

int return\_the\_answer() {

return 42;

}

#### local\_variable\_declaration.src.cpp

自動変数の宣言を扱うテスト。

void f() {

int i = 1;

}

#### multi\_dim\_array.src.cpp

多重配列を扱うテスト。

int b[1][2][3][4][5];

#### params\_of\_function\_declaration.src.cpp

引数名を含む関数プロトタイプ宣言を扱うテスト。

void f();

int g(int i);

int h(double d, int i);

double i(double d1, int, int i, double d2, short s);

#### params\_of\_function\_definition.src.cpp

引数名を含む関数プロトタイプ宣言を(特に i という名前で宣言された引数の型について)正しく復元できることを確認するテスト。

void f() {

}

int g(int i) {

return i;

}

int h(double d, int i) {

return i;

}

double i(double d1, int, int i, double d2, short s) {

return d2;

}

#### pointer\_to\_function.src.cpp

関数ポインターの宣言を扱うテスト。

void declare\_pointer() {

void (\*f)(int, int);

void (\*fp)(int (\*)(int, int));

}

#### pointer\_type.src.cpp

ポインターの宣言を扱うテスト。

void declare\_pointers() {

int n;

int \*a = &n;

double \*\*b;

const int \*c;

int \* const d = 0;

const int \* const e = 0;

void (\*f)(int, int);

}

#### simple\_class.src.cpp

単純なクラス定義を扱うテスト。

class A {

public:

int x;

int y;

double dist();

private:

int pv;

};

#### variables\_with\_the\_same\_name\_in\_different\_scopes.src.cpp

異なるスコープを持つ同じ名前を正しく復元できることを確認するテスト。

void f() {

int i = 0;

{

long i = 0;

}

}

void g() {

int g = 0;

}

void h(int g) {

}

#### void\_return.src.cpp

空のreturn文を扱うテスト。

void f() {

return;

}

#### while.src.cpp

while文を扱うテスト。

void f() {

while (1) {

int i = 1;

}

}