Na terytorium kompilatora: libclang

W ostatnich latach kompilator Clang/LLVM zdobył ogromną popularność, stając się pełnowartościową alternatywą dla GCC. Z punktu widzenia użytkowników kompilatorów głównymi cechami, które to umożliwiły są: kompatybilność interfejsu linii komend z GCC, przejrzystość i jakość informacji zwrotnej o błędach i ostrzeżeniach podczas kompilacji oraz szybkie zapewnienie wsparcia dla nowego standardu języka C++.

Clang to jednak nie tylko kompilator. To również zestaw bibliotek do przetwarzania kodu źródłowego. Większość z nich oferuje interfejs w języku C++, który może zmieniać się pomiędzy kolejnymi wersjami Clanga i wymaga posiadania wiedzy o budowie i działaniu kompilatorów. Przygotowana została także współdzielona biblioteka oferująca stabilny interfejs w języku C: *libclang* [1, 2]. W dalszej części artykułu zaprezentowany zostanie zarys jej możliwości.

Możliwości biblioteki libclang

Biblioteka libclang operuje na poziomie abstrakcyjnego drzewa rozbioru (ang. *Abstract Syntax Tree, AST*), które reprezentuje kod programu w języku C, Objective-C lub C++. Wśród jej możliwości znajdują się:

- podział kodu na tokeny,
- budowa drzew AST,
- poruszanie się po drzewie AST za pomocą kursorów,
- odwzorowanie pomiędzy kursorami a lokacjami w kodzie,
- raportowanie informacji diagnostycznych,
- zapewnianie podpowiedzi kontekstowych.

Największe zalety biblioteki to:

- nieskomplikowany i stabilny interfejs w języku C,
- możliwość uzyskania dokładnej informacji o lokalizacji w kodzie na każdym etapie pracy z biblioteką,
- popularność i renoma stojącego za nią projektu.

Od czego zacząć?

Aby rozpocząć pracę z biblioteką libclang należy skorzystać z odpowiedniego pliku nagłówkowego a następnie stworzyć indeks oraz wczytać jednostkę translacji (ang. *Translation Unit, TU*). Przykład kodu realizującego te zadania przestawiono na listingu 1. Zmienne `argv` oraz `argc` powinny przechowywać liczbę parametrów oraz same parametry kompilacji (np. „`-std=c++11 -Wall file.cpp`” w formie, jaka zwykle trafia do funkcji `main`). Kod operujący na jednostce translacji powinien trafić w miejsce oznaczone przez komentarz.

Listing 1. Tworzenie jednostki translacji

```
#include <clang-c/Index.h>

auto index = clang_createIndex(0, 0);
auto tu = clang_parseTranslationUnit(
                index, 0, argv, argc, 0, 0,
                CXTranslationUnit_None);

// operacje na jednostce translacji

clang_disposeTranslationUnit(tu);
clang_disposeIndex(index);
```

Informacje o błędach i ostrzeżeniach

Korzystając z biblioteki libclang, można uzyskać wysokiej jakości informacje diagnostyczne a następnie wykorzystać je we własnym programie. Kod ilustrujący otrzymywanie *sformatowanej* informacji przedstawiony został na listingu 2. a jego działanie na listingach 3. i 4.

Listing 2. Uzyskiwanie informacji o błędach i ostrzeżeniach

```
for (auto diagNum = 0u;
   diagNum < clang_getNumDiagnostics(tu); ++diagNum) {
  auto diag = clang_getDiagnostic(tu, diagNum);
  auto diagStr = clang_formatDiagnostic(diag,
          clang_defaultDiagnosticDisplayOptions());
  std::cout << clang_getCString(diagStr) << "\n";
  clang_disposeString(diagStr);
}
```

Listing 3. Niepoprawny kod C++

```
class X {
    const int a;
}
```

Listing 4. Informacja diagnostyczna dla kodu z listingu 3.

```
class.cpp:1:7: warning: class 'X' does not declare any
constructor to initialize its non-modifiable members
class.cpp:3:2: error: expected ';' after class
class.cpp:2:15: warning: private field 'a' is not used
[-Wunused-private-field]
```

Powyższy przykład jest najprostszą metodą otrzymywania informacji diagnostycznej. Każdy z jej komponentów: lokalizację, poziom i tekst, można uzyskać oddzielnie. Ponadto za pomocą funkcji `clang_getDiagnosticNumFixIts` i `clang_getDiagnosticFixIt` możliwe jest otrzymanie opcjonalnych podpowiedzi o tym, jak rozwiązać problem (ang. *fix-its*), o którym Clang informuje. Dla błędu braku średnika w kodzie z listingu 3. fix-it będzie następujący: „`3:2 - 3:2:  ;`” (w trzeciej linii, w drugiej kolumnie postaw znak „;”).

Tak obszerne informacje diagnostyczne wykorzystywane są np. do analizy kodu źródłowego pod kątem występowania ostrzeżeń lub w zintegrowanych środowiskach programistycznych (ang. *Integrated Development Environment, IDE*) do prezentacji bieżącego stanu rozwijanego kodu.

Drzewo rozbioru

Drzewo AST generowane przez kompilator Clang można zobaczyć w formie tekstowej, jeśli wywołany zostanie on z argumentami „`-Xclang -ast-dump`”. Drzewo dla kodu z listingu 5. prezentuje listing 6. Może ono być przydatne w analizie błędów podczas pisania kodu, ponieważ zawiera również elementy wygenerowane przez kompilator, np. domyślne konstruktory, operatory kopiowania i instancje szablonów dla konkretnych typów.

Listing 5. Przykładowy kod C++

```
int main() {
    auto a = 1;
    return 10 + a;
}
```

Listing 6. Drzewo AST (wybrane informacje) dla kodu z listingu 5.

```
TranslationUnitDecl 0x9c0fc00 <<>> <>
|-FunctionDecl 0x9c0ff60 <ex.cpp:1:1, line:4:1>
line:1:5 main 'int (void)'
  `-CompoundStmt 0x9c10148 <col:12, line:4:1>
    |-DeclStmt 0x9c100d0 <line:2:5, col:15>
    | `-VarDecl 0x9c10000 <col:5, col:14> col:10 used
a 'int':'int' cinit
    |   `-IntegerLiteral 0x9c10030 <col:14> 'int' 1
    `-ReturnStmt 0x9c10138 <line:3:5, col:17>
      `-BinaryOperator 0x9c10120 <col:12, col:17>
'int' '+'
        |-IntegerLiteral 0x9c100e0 <col:12> 'int' 10
        `-ImplicitCastExpr 0x9c10110 <col:17>
'int':'int' <LValueToRValue>
          `-DeclRefExpr 0x9c100f8 <col:17> 'int':'int'
lvalue Var 0x9c10000 'a' 'int':'int'
```

Poruszanie się po drzewie AST

W bibliotece libclang udostępniony został interfejs wspierający poruszanie się po drzewie AST zbudowanym dla danej jednostki translacji. Umożliwia to analizę jednostki nie tylko pod względem składniowym, ale również semantycznym.

Przeglądanie drzewa AST zaimplementowano z użyciem wzorca wizytatora (ang. *visitor pattern*). Bytem, za pomocą którego odwiedzamy węzły drzewa, jest `CXCursor`. Kursor może wskazywać m.in. takie elementy języka C++ jak deklaracje, definicje, wyrażenia lub referencje. Niesie on informacje o nazwie, typie, lokalizacji w kodzie źródłowym oraz swoich dzieciach.

Drzewo AST mogłoby zostać przejrzane w celu znalezienia deklaracji funkcji i metod, np. aby sprawdzić czy nazwy tych elementów są zgodne z przyjętą w projekcie konwencją. Działanie takie jest przedstawione na listingu 7. Funkcja wizytująca (`guest`) sprawdza rodzaj każdego kursora (`clang_getCursorKind`) i podejmuje odpowiednie akcje dla interesujących nas kursorów oraz decyduje jak przeglądać drzewo. Aby podążać w głąb niego konieczne jest zwrócenie wartości `CXChildVisit_Recurse`.

Listing 7. Przeglądanie drzewa AST w celu znalezienia deklaracji funkcji i metod

```
CXChildVisitResult guest(CXCursor cursor,
                         CXCursor parent,
                         CXClientData client_data) {
  switch (clang_getCursorKind(cursor)) {
    case CXCursor_FunctionDecl:
      std::cout << "function"; break;
    case CXCursor_CXXMethod:
      std::cout << "cxxmethod"; break;
    default: break;
  }
  return CXChildVisit_Recurse;
}

clang_visitChildren(
      clang_getTranslationUnitCursor(tu), guest, 0);
```

Kursor udostępnia informacje nie tylko o sobie samym, lecz także o elementach z nim powiązanych. Dla kursorów wskazujących deklaracje można znaleźć ich definicję (`clang_getCursorDefinition`) natomiast dla referencji otrzymać kursor, do którego się one odwołują (`clang_getCursorReferenced`). Nie jest problemem uzyskanie informacji o składniowym i semantycznym rodzicu danego kursora (`clang_getCursorLexicalParent`, `clang_getCursorSemanticParent`). Różnicę między nimi obrazuje definicja metody klasy znajdująca się poza ciałem tej klasy. Jej semantycznym rodzicem będzie klasa, do której należy, natomiast leksykalnym otoczenie definicji (np. pewna przestrzeń nazw).

Dzięki swobodnemu przemieszczaniu się po drzewie AST, można także wykonywać obliczenia związane ze strukturą kodu. Umożliwia to analizę programu reprezentowanego przez dane AST.

Podpowiadanie semantyczne

Jedną z cech, z których znana jest biblioteka libclang, to generowanie podpowiedzi semantycznych dla tworzonego kodu. Środowiska takie XCode, QtCreator, KDevelop korzystają lub są w trakcie migracji do mechanizmów oferowanych przez libclang. Dotychczasowe rozwiązania, oparte najczęściej na mniej lub bardziej poprawnych parserach napisanych na potrzeby mechanizmu podpowiadania, okazały się zbyt mało elastyczne, by dostosować się do standardu C++11. Niewątpliwą zaletą podpowiadania przy pomocy libclang jest również zawarcie w zbiorze podpowiedzi elementów generowanych przez kompilator.

Eksperymenty z autouzupełnianiem można zacząć od narzędzia `c-index-test`, dostarczanego razem z biblioteką:

```
$ c-index-test –code-completion-at=\
          <plik>:<linia>:<kolumna> <plik + argument>
```

Kod, który umożliwia otrzymanie oraz przetworzenie podpowiedzi przedstawia listing 8. Niestety w tym momencie interfejs biblioteki staje się skomplikowany, dlatego na listingu zaprezentowany został tylko zarys jego użycia.

Listing 8. Zarys kodu uzyskującego i przetwarzającego podpowiedzi

```
auto compls = clang_codeCompleteAt(
                tu, "file.cpp", line, column, 0, 0,
                clang_defaultCodeCompleteOptions());

for (auto i = 0u; i < compls->NumResults; ++i) {
  auto &cStr = compls->Results[i].completionString;
  for (auto j = 0u; j <
       clang_getNumCompletionChunks(cStr); ++j) {
       auto chunkStr =
              clang_getCompletionChunkText (cStr, j);
     std::cout << toString(chunkStr) << " ";
  }
}

clang_disposeCodeCompleteResults(compls);
```

Podsumowanie

Przykłady zaprezentowane powyżej dają zaledwie przedsmak tego, co można osiągnąć z wykorzystaniem biblioteki libclang. Dzięki niej można cieszyć się lepszymi środowiskami IDE. Deweloperzy mogą wykorzystać ją do tworzenia narzędzi do refaktorowania, formatowania, migracji, analizy i generowania kodu. Nie jest przy tym konieczne bezpośrednie używanie interfejsu C, ponieważ udostępnione zostały także wiązania w języku Python [3]. Gdy mechanizmy oferowane przez libclang okażą się niewystarczające względem potrzeb, można skorzystać z wewnętrznych interfejsów kompilatora Clang.

W sieci

[1] http://clang.llvm.org/doxygen/group__CINDEX.html
[2] http://llvm.org/devmtg/2010-11/Gregor-libclang.pdf
[3] http://eli.thegreenplace.net/2011/07/03/parsing-c-in-python-with-clang
[4] http://cppwroclaw.pl/dokuwiki/_media/spotkania/005/01_libclang.pdf