Tree-based refactorings with Clang

THIS IS A PUBLICLY READABLE DOCUMENT

Authors: Dmitri Gribenko <gribozavr@gmail.com>
Ilya Biryukov <ibiryukov@google.com>

Link to the document

Introduction

Most Clang-based refactoring tools eventually end up doing text transformations, i.e. they traverse the Clang AST to find the source locations of AST nodes they need to change and later use the obtained locations to produce text transformation for the desired refactoring.

Producing correct text replacements is hard: one needs to take the preprocessor into account, take care to avoid introducing unintended changes (e.g. introducing dangling else), etc. Moreover, using textual replacements prohibits composition of transformations: the input to the refactoring is typically a Clang AST, an output is text. Going back to the Clang AST requires rerunning the compiler, making it non-practical to write small transformations that can be reused in more complex refactorings.

We propose a new tree-based API for writing transformations of C++ code that would make it easy to write reusable and composable transformations. Composability means output of the transformation can be used as inputs to other transformations. In addition, we plan to provide a curated set of well-behaved “core” transformations for writing refactorings, enabling the users of the API to write correct and easy to understand code for their refactoring tools.

Prototype implementation

A prototype implementation of this design is available on GitHub. At the time of writing (25 Feb 2019), it supports building syntax tree nodes for a subset of C++ expressions and statements, and implements the basic low-level mutation operations.
For the tree API see clang/include/clang/Tooling/Syntax/Syntax.h
For some simple example transformations see clang/tools/clang-syntax/Transformations.cpp
A sketch of other example transformations can be found in the appendix of this document.
The first user of this API will be the refactoring subsystem in clangd. We will implement a few refactorings in clangd based on it and iterate on the API in the process. When we are happy with the API, we will declare it good enough for wider usage.

Non-goals

- Define a complete and comprehensive refactoring API. This document is more narrow in scope, it aims to provide low-level building blocks to make it easier to do syntactic transformations of C++. Refactorings are a much broader topic.

Limitations of the Clang AST

Some traits of the Clang AST make it difficult to use as a data structure for writing the syntactic transformations.

Driven by the needs of the compiler, the AST is designed to provide easy access to **semantic** information about C++ programs. However, accessing **syntactic** information is not always easy, here are just two examples:

- Different ASTs for the same syntax:

```
int a;
std::ostream &os;
a << 1;  // Represented as BinaryOperator.
os << 1; // Represented as CXXOperatorCallExpr.
```

- Same ASTs for the different syntactic constructs:

```
int *a, *b, **c;          // Represented as 3 decls in the AST.
int *a; int *b; int **c; // Represented as 3 decls in the AST.
```

More importantly, the Clang AST cannot be mutated without reparsing the underlying C++ code. This renders it impossible to implement a composable mutation API on top of it without re-parsing of the source code between refactoring steps. Even if we implement re-reparsing, we would face a bigger issue: maintaining pointers to AST nodes across re-parses.

To address these limitations, we propose an alternative tree representation of source code that is more suitable for writing transformations and inspecting the concrete syntax of the language. We call it the Syntax Tree.

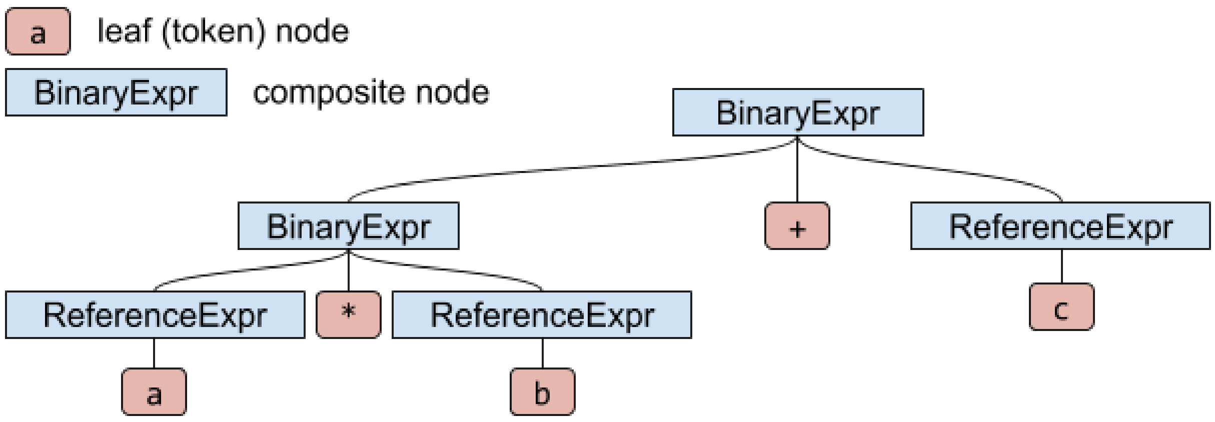
Syntax Tree

The syntax tree nodes form a class hierarchy independent from the Clang AST. The hierarchy of syntax tree nodes aims to capture the syntax of the language, i.e. closely corresponds to the C++ grammar.

Syntax trees can be built from the Clang AST, however syntax trees do not store most semantic information that the AST has. E.g. to answer questions like “what decl does this reference resolve to?”, the users will have to look at the Clang AST.

The syntax tree is composed of two kinds of nodes: leaf nodes and composite nodes. Composite nodes correspond to C++ grammar constructs, leaf nodes correspond to tokens after preprocessing. Traversing all leaf nodes left-to-right produces exactly the preprocessed tokens[1] of the source file. E.g. here is a syntax tree for the `a * b + c` expression:

[1]We also need to take comments into account, although they are technically not part of the preprocessed tokens. Comments are discussed in more detail below.



The syntax trees are mutable - one can add or remove subtrees of composite nodes. Users are also provided with a more high-level APIs to transform syntax trees in a well-defined manner, preserving the invariants of the syntactic structure. E.g. one is allowed to remove arguments of a function call or add more statements to a compound statement but not allowed replace a top-level declaration with an expression.

As a consequence, syntax trees can also be created without actual source code, e.g. given a syntax tree for an expression `E`, one can create a syntax tree for an unary expression `!E` or a call expression `E()`.

A sequence of mutations can be turned into textual replacements. This is usually a final step of the transformation.

Links between the Clang AST and the Syntax Tree

Links from the Clang AST to the Syntax Tree

We anticipate that implementations of refactorings would want to find the Syntax Tree nodes using semantic information. They would use the existing Clang AST for that. Once the appropriate AST nodes have been identified, the user will have ways to find the corresponding syntax tree nodes.

Links from the Syntax Tree to the Semantic AST

Links from the syntax tree to the semantic AST are difficult to implement in a way that provides a good experience for the API user.

The simplest option is to have a pointer to the original semantic AST node in each syntax tree node. However, as the user mutates the syntax tree, the semantic AST can become logically invalid (not the right AST for the translation unit represented by the syntax tree). What should the syntax tree API do when the semantic AST is invalidated? It seems infeasible to detect when the underlying semantic AST becomes invalid, given the complexity of C++ language rules. If the syntax tree does not detect invalidation automatically, and asks the user to detect invalidation, it can be an API contract that the user can not satisfy.

An option that is more difficult to implement is to allow querying the semantic AST when the translation unit represented by the syntax tree is well-formed. In that case, we could serialize the source file back into text, re-parse it, generate a new syntax tree -- that should match the one that we serialized, find the corresponding syntax tree node, and from there find the corresponding semantic AST node.

Mutation API

We allow mutating the syntax trees, but want to restrict the set of available mutations to make sure that:

1. produced syntax trees comply with the invariants of the C++ grammar (e.g. translation unit cannot directly contain expressions as children nodes)
2. we can produce the resulting text replacements

Note that transformations do not aim to preserve semantics, users of the API are responsible for figuring out if a particular transformation is applicable for their purposes. E.g. inlining a variable might change the order of computations with side-effects, and the mutation APIs do not aim to prevent that. In fact, we anticipate that some users will want to write transformations that change the semantics of the program.

Preserving invariants on mutations

In order to avoid producing inconsistent ASTs we define a set of primitive transformations and leverage the type system to avoid invalid AST transformations.

```
/// Replaces an expression with a new one. Can parenthesize if necessary, e.g.
/// if we replace 'a * b' in 'a * b * c' with 'd + e', the resulting tree
/// has an extra pair of parentheses '(d+e)*c'.
void replaceExpression(syntax::Expr* Existing, syntax::Expr* New);
```

```
/// Removes the passed statement from the tree, will keep the semicolon to keep
/// the syntax correct, e.g.:
///      if (foo) StatementToRemove;
///   → if (foo) ;
/// will remove the whole statement, however:
/// has to keep an empty statement to keep the syntax correct.
///      { StatementToRemove; Stmt2; Stmt3; }
///   → { Stmt2; Stmt3; }
void removeStatement(syntax::Stmt* S);
```

```
// E.g. ParameterList = (int a, int b), ParameterDecl = 'double c'
// → (int a, double c, int b)
```

```
void addParameter(syntax::FunctionParameterList*, syntax::ParameterDecl*);
```

Ensuring mutations correspond to text changes

Not all transformations on syntax trees can be represented as meaningful corresponding text replacements. E.g. consider

```
#define macro(a, b) !(a || b)
void example(bool x, bool y) {
   !(x || b);    // #1
   macro(x, b);  // #2
}
```

Say, we want to “simplify” all expressions of the form `!(a || b)` into `a && b`. Both #1 and #2 produce the same syntax tree, however it’s not clear what textual replacements should correspond to number #2: we can’t apply the transformation directly without either expanding a macro at the call site or changing the macro definition. In most cases, neither of these is desirable without an explicit acknowledgement from the user, so we choose to restrict syntax tree transformations in presence of macros to a few well-defined cases:

1. Replacing a subtree fully produced by the macro expansion
2. Replacing a subtree fully covered by the macro arguments

```
#define macro(a, b) a+b
#define assert(x) if (!x) exit(1)
void example(int a) {
  assert(a == 10);
  assert(macro(a == 10, 10));
}
```

Users would have to explicitly check that transformations on the trees they modify are well-defined.

Handling comments

To produce high-quality textual replacements, syntax tree mutation APIs will initially automatically handle moving/removing comments in most cases. At a later stage, we might provide finer-grained APIs for dealing with comments in more complicated cases, as we collect them.

Here are examples of some cases that should be handled automatically:

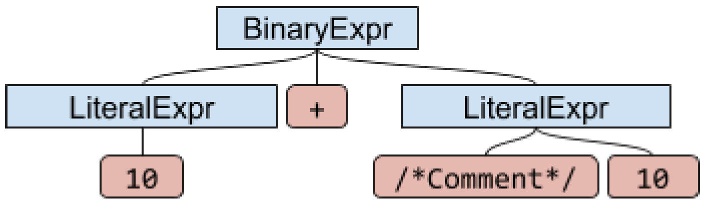
- Comments in the middle of the subtrees:

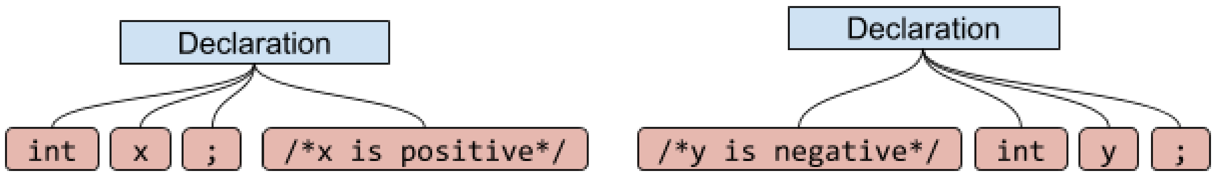
```
void test() {
  int x = 10 + /*Comment*/20;
  int y = x; // inlining 'x' should retain the comment before "20".
}
```

- Comments at the edges of subtrees

```
struct Foo {
  int x; // x is positive
  // y is negative
  int y;
};
// Moving 'x' and 'y' around should retain respective comments.
```

Roughly, the idea is to detect the syntax nodes the comment corresponds to and attach a comment as a leaf node. Manipulations on the tree would retain the attached comments just like any other node. E.g. the syntax trees for the examples above would roughly look like
:





FAQ

What if refactorings require semantic information?

If the refactoring code requires semantic information, it should obtain the desired information from the Clang AST before doing any transformations on the syntax trees. One would be able to get a mapping from the AST nodes to the syntax tree nodes.

However, preserving some information in the syntax trees during transformations can be useful in practice, e.g. keeping targets of resolve when moving code around would greatly help to write simpler refactorings. Nevertheless, we choose to restrict the initial implementation to not support such use-case for simplicity.

Do you build the Syntax Tree for the whole Translation Unit?

Different clients of the Syntax Tree API will have different use cases. Some clients will only require the syntax tree for syntactic constructs in the primary file. Some clients will need to see the syntax tree for everything, including code that came from included headers. We intend to eventually support both use cases, but initially we want to support constructing the syntax tree for the primary file only.

How to create syntax trees nodes?

We will provide factory functions to create well-defined syntax nodes from the subtrees. E.g. when creating a binary expression, the factory function would take care of parenthesizing the subcomponents if necessary:

```
/// Adds parentheses around if necessary.
syntax::BinaryExpr* createBinary(syntax::Expr* Left,
                                 tok::Kind Operator,
                                 syntax::Expr* Right);

syntax::Expr* Left  = … /* tree for 'a + b' */;
syntax::Expr* Right = … /* tree for 'c' */;
auto *NoParens  = createBinary(Left, tok::plus, Right); // result is `a + b +
c`.
auto *HasParens = createBinary(Left, tok::mult, Right); // result is `(a + b) *
c`.
```

What happens with the implicitly generated code?

Syntax trees do not have nodes that correspond to the implicitly generated AST nodes. E.g. syntax trees would not have nodes corresponding to the implicit calls to `begin()` and `end()` in a range-based for statement.

Will we end up needing to reinvent matchers for this representation?

We want to avoid this. Matchers are a complicated beast and having a separate set of those for syntax trees would be unfortunate. If one wants to take advantage of the concept of matchers and syntax trees, they have options to do so based on existing implementation of ASTMatchers, e.g. they could:

1. use ASTMatchers to find interesting Clang AST nodes,
2. find their syntax tree counterparts,
3. apply transformations to syntax trees.

Appendix. API Evaluation. Case Study.

In this section we present a sample refactoring transformation and two implementation approaches: a functional approach (create new nodes without touching existing nodes) and an in-place mutation.

Refactoring Transformation: Negate Boolean Expression

We would like to implement a “negate boolean expression” refactoring. E.g. the user selects a boolean expression, and the IDE inverts it.

When applied to an expression “E”, it should be rewritten into “!E”. However, applying this transformation mechanically would create non-idiomatic code. Therefore, we would like to support the following special cases.

- When applied to expression that has shape “E1 == E2”, the result should be “E1 != E2”.
- When applied to expression that has shape “E1 && E2”, the result should be “!E1 || E2”.

Functional approach

In this approach we create new expression instead of modifying the existing ones on-the-fly. Clients are responsible for replacing the subtrees themselves.

```
// Returns a copy of a BinaryExpr, replacing the operator and maintaining any
// ancillary tokens (like comments).
replaceOperator: syntax::BinaryExpr* x Token -> syntax::BinaryExpr*
```

```
// Negating an expression.
syntax::Expr* negate(syntax::Expr* E) {
  if (auto Binary = dyn_cast<syntax::BinaryExpr*>(E)) {
    if (Binary.Operator.Kind == tok::EqEq) {
      // a == b → a != b
      return replaceOperator(Binary, createToken(tok::ExclEq));
    } else if (Binary.Operator.Kind == tok::AmpAmp) {
      // a && b → !a || !b
      return Binary(negate(Binary.LeftArg),
                    createToken(tok::PipePipe),
                    negate(Binary.RightArg));
    }
    // fallthrough
  } else if (auto Unary = dyn_cast<syntax::UnaryExpr*>(E)) {
    if (Unary.Operator.Kind == tok::Excl) {
      // !a → a
      return Unary.Argument;
    }
    // fallthrough
  }
  // E → !E
```

In-place mutation

In this approach, we mutate expressions in-place. This can be more efficient, but might lead to unexpected surprises like unexpectedly accessing child nodes that were mutated.

```
// Negating an expression.
void negate(syntax::Expr* E) {
  if (auto Binary = dyn_cast<syntax::BinaryExpr*>(E)) {
    if (Binary.Operator.Kind == tok::EqEq) {
      // a == b → a != b
      replaceToken(Binary.Operator, createToken(tok::ExclEq));
      return;
    } else if (Binary.Operator.Kind == tok::AmpAmp) {
      // a && b → !a || !b
      negate(Binary.LeftArg);
      negate(Binary.RightArg);
      replaceToken(Binary.Operator, createToken(tok::PipePipe));
      return;
    }
    // fallthrough
  } else  if (auto Unary = dyn_cast<syntax::UnaryExpr*>(E)) {
    if (Unary.Operator.Kind == tok::Excl) {
      // !a → a
      replaceExpression(Unary, Unary.Argument);
      return;
    }
    // fallthrough
  }
  // E → !E
  replaceExpression(E, createUnaryExpr(/*Arg=*/E, tok::Excl));
}
```