```cpp
//===--- FuzzyParser.cpp - clang-highlight ---------------------*- C++ -*-===//
//
//                     The LLVM Compiler Infrastructure
//
// This file is distributed under the University of Illinois Open Source
// License. See LICENSE.TXT for details.
//
//===----------------------------------------------------------------------===//
#include "llvm/Support/Debug.h"
#include "llvm/ADT/STLExtras.h"
#include "clang/Basic/IdentifierTable.h"
#include "clang/Basic/OperatorPrecedence.h"
#include "FuzzyAST.h"

using namespace llvm;

namespace clang {
namespace fuzzy {

namespace {
template <bool SkipPreprocessor> class BasicTokenFilter {
  AnnotatedToken *First, *Last;

  void skipWhitespaces() {
    for (;;) {
      while (First != Last && (First->getTokenKind() == tok::unknown ||
                               First->getTokenKind() == tok::comment))
        ++First;

      if (SkipPreprocessor && First->getTokenKind() == tok::hash &&
          First->Tok().isAtStartOfLine())
        while (First != Last && First++->getTokenKind() != tok::eod)
          ;
      else
        break;
    }
    assert(First <= Last);
  }

public:
  BasicTokenFilter(AnnotatedToken *First, AnnotatedToken *Last)
      : First(First), Last(Last) {
    skipWhitespaces();
  }

  AnnotatedToken *next() {
    assert(!eof());
    auto Ret = First++;
    skipWhitespaces();
    assert(Ret->getTokenKind() != tok::raw_identifier);
    return Ret;
  }

  class TokenFilterState {
    friend class BasicTokenFilter;
    TokenFilterState(AnnotatedToken *First, AnnotatedToken *Last)
        : First(First), Last(Last) {}
    AnnotatedToken *First, *Last;
  };

  TokenFilterState mark() const { return TokenFilterState(First, Last); }
  void rewind(TokenFilterState State) {
    First = State.First;
    Last = State.Last;
  }

  BasicTokenFilter<true> rangeAsTokenFilter(TokenFilterState From,
                                            TokenFilterState To) const {
    assert(From.Last == To.Last);
    assert(From.First <= To.First);
    assert(To.First < To.Last);
    return BasicTokenFilter<true>(From.First, To.First + 1);
  }

  class TokenFilterGuard {
    friend class BasicTokenFilter;
    TokenFilterGuard(BasicTokenFilter *TF, TokenFilterState State)
        : TF(TF), State(State) {}

  public:
    ~TokenFilterGuard() {
      if (TF)
        TF->rewind(State);
    }
    void dismiss() { TF = nullptr; }
    BasicTokenFilter *TF;
    TokenFilterState State;
  };
  TokenFilterGuard guard() { return TokenFilterGuard(this, mark()); }

  AnnotatedToken *peek() { return First; }
  const AnnotatedToken *peek() const { return First; }
  tok::TokenKind peekKind() const { return First->getTokenKind(); }

  bool eof() const { return peekKind() == tok::eof; }
};
using TokenFilter = BasicTokenFilter<true>;
using RawTokenFilter = BasicTokenFilter<false>;
} // end anonymous namespace

template <bool B>
static bool checkKind(BasicTokenFilter<B> &TF, tok::TokenKind Kind) {
  return TF.peekKind() == Kind;
}

static int PrecedenceUnaryOperator = prec::PointerToMember + 1;
static int PrecedenceArrowAndPeriod = prec::PointerToMember + 2;

static std::unique_ptr<Expr> parseExpr(TokenFilter &TF, int Precedence = 1,
                                       bool StopAtGreater = false);

static std::unique_ptr<Type> parseType(TokenFilter &TF,
                                       bool WithDecorations = true);

static std::unique_ptr<Expr> parseUnaryOperator(TokenFilter &TF) {
  if (checkKind(TF, tok::plus) || checkKind(TF, tok::minus) ||
      checkKind(TF, tok::exclaim) || checkKind(TF, tok::tilde) ||
      checkKind(TF, tok::star) || checkKind(TF, tok::amp) ||
      checkKind(TF, tok::plusplus) || checkKind(TF, tok::minusminus)) {
    AnnotatedToken *Op = TF.next();
    auto Operand = parseUnaryOperator(TF);
    if (!Operand)
      return {};
    return make_unique<UnaryOperator>(Op, std::move(Operand));
  }

  return parseExpr(TF, PrecedenceArrowAndPeriod);
}

static std::unique_ptr<Expr>
parseCallExpr(TokenFilter &TF, std::unique_ptr<DeclRefExpr> FunctionName) {
  assert(checkKind(TF, tok::l_paren));
  auto Func = make_unique<CallExpr>(std::move(FunctionName));
  Func->setLeftParen(TF.next());
  while (!checkKind(TF, tok::r_paren)) {
    Func->Args.push_back(parseExpr(TF, prec::Comma + 1));
    if (checkKind(TF, tok::comma))
      Func->appendComma(TF.next());
    else
      break;
  }
  if (checkKind(TF, tok::r_paren)) {
    Func->setRightParen(TF.next());
    return std::move(Func);
  }
  return {};
}

static bool isLiteralOrConstant(tok::TokenKind K) {
  if (isLiteral(K))
    return true;

  switch (K) {
  case tok::kw_this:
  case tok::kw_true:
  case tok::kw_false:
  case tok::kw___objc_yes:
  case tok::kw___objc_no:
  case tok::kw_nullptr:
    return true;
  default:
    return false;
  }
}

template <typename QualOwner>
static bool parseNamespaceQualifiers(TokenFilter &TF, QualOwner &Qual) {
  auto Guard = TF.guard();

  if (checkKind(TF, tok::kw_operator)) {
    Qual.addNameQualifier(TF.next());
    if (!TF.peek())
      return false;
    Qual.addNameQualifier(TF.next());
    Guard.dismiss();
    return true;
  }

  bool GlobalNamespaceColon = true;
  do {
    if (checkKind(TF, tok::coloncolon))
      Qual.addNameQualifier(TF.next());
    else if (!GlobalNamespaceColon)
      return false;
    GlobalNamespaceColon = false;
    if (!checkKind(TF, tok::identifier))
      return false;
    Qual.addNameQualifier(TF.next());
  } while (checkKind(TF, tok::coloncolon));

  Guard.dismiss();
  return true;
}

template <typename QualOwner>
static bool parseTemplateArgs(TokenFilter &TF, QualOwner &Qual,
                              std::false_type) {
  return true;
}
template <typename QualOwner>
static bool parseTemplateArgs(TokenFilter &TF, QualOwner &Qual,
                              std::true_type) {
  auto Guard = TF.guard();

  if (checkKind(TF, tok::less)) {
    Qual.makeTemplateArgs();
    bool isFirst = true;
    do {
      Qual.addTemplateSeparator(TF.next());

      if (isFirst && checkKind(TF, tok::greater))
        break;
      isFirst = false;

      if (auto Arg = parseType(TF))
        Qual.addTemplateArgument(std::move(Arg));
      else if (auto E = parseExpr(TF, prec::Comma + 1, /*StopAtGreater=*/true))
        Qual.addTemplateArgument(std::move(E));
      else
        return false;
    } while (checkKind(TF, tok::comma));
    if (!checkKind(TF, tok::greater))
      return false;
    Qual.addTemplateSeparator(TF.next());
  }

  Guard.dismiss();
  return true;
}

```

```
template <typename QualOwner, typename WithTemplateArgs = std::true_type>
static bool parseQualifiedID(TokenFilter &TF, QualOwner &Qual,
                             WithTemplateArgs WTA = std::true_type{}) {
  auto Guard = TF.guard();
  if (parseNamespaceQualifiers(TF, Qual) && parseTemplateArgs(TF, Qual, WTA)) {
    Guard.dismiss();
    return true;
  }
  return false;
}

static std::unique_ptr<Expr> parseExpr(TokenFilter &TF, int Precedence,
                                       bool StopAtGreater) {
  if (!TF.peek())
    return {};

  if (Precedence == PrecedenceUnaryOperator)
    return parseUnaryOperator(TF);

  if (Precedence > PrecedenceArrowAndPeriod) {
    if (isLiteralOrConstant(TF.peekKind()))
      return make_unique<LiteralConstant>(TF.next());

    if (checkKind(TF, tok::l_paren)) {
      auto Left = TF.next();
      auto Val = parseExpr(TF, 1, false);
      if (!checkKind(TF, tok::r_paren))
        return {};
      auto Right = TF.next();
      return make_unique<ParenExpr>(Left, std::move(Val), Right);
    }

    if (checkKind(TF, tok::identifier) || checkKind(TF, tok::coloncolon)) {
      auto DR = make_unique<DeclRefExpr>();
      if (!parseQualifiedID(TF, *DR) &&
          !parseQualifiedID(TF, *DR, std::false_type{}))
        return {};
      if (checkKind(TF, tok::l_paren))
        return parseCallExpr(TF, std::move(DR));
      std::unique_ptr<Expr> Ret = std::move(DR);
      while (checkKind(TF, tok::plusplus) || checkKind(TF, tok::minusminus))
        Ret = make_unique<UnaryOperator>(TF.next(), std::move(Ret));
      return std::move(Ret);
    }

    return {};
  }
  auto LeftExpr = parseExpr(TF, Precedence + 1, StopAtGreater);
  if (!LeftExpr)
    return {};

  while (!TF.eof()) {
    if (StopAtGreater && checkKind(TF, tok::greater))
      break;

    int CurrentPrecedence = getBinOpPrecedence(TF.peekKind(), true, true);
    if (checkKind(TF, tok::period) || checkKind(TF, tok::arrow))
      CurrentPrecedence = PrecedenceArrowAndPeriod;
    if (CurrentPrecedence == 0)
      return LeftExpr;

    assert(CurrentPrecedence <= Precedence);
    if (CurrentPrecedence < Precedence)
      break;
    assert(CurrentPrecedence == Precedence);

    AnnotatedToken *OperatorTok = TF.next();

    auto RightExpr = parseExpr(TF, Precedence + 1, StopAtGreater);
    if (!RightExpr)
      return {};

    LeftExpr = make_unique<BinaryOperator>(std::move(LeftExpr),
                                           std::move(RightExpr), OperatorTok);
  }

  return LeftExpr;
}

static std::unique_ptr<Stmt> parseReturnStmt(TokenFilter &TF) {
  auto Guard = TF.guard();
  if (!checkKind(TF, tok::kw_return))
    return {};
  auto *Return = TF.next();
  std::unique_ptr<Expr> Body;
  if (!checkKind(TF, tok::semi)) {
    Body = parseExpr(TF);
    if (!Body || !checkKind(TF, tok::semi))
      return {};
  }
  assert(checkKind(TF, tok::semi));
  auto *Semi = TF.next();
  Guard.dismiss();
  return make_unique<ReturnStmt>(Return, std::move(Body), Semi);
}

static void parseTypeDecorations(TokenFilter &TF, Type &T) {
  // TODO: add const and volatile
  while (checkKind(TF, tok::star) || checkKind(TF, tok::amp) ||
         checkKind(TF, tok::ampamp))
    T.Decorations.push_back(Type::Decoration(checkKind(TF, tok::star)
                                                 ? Type::Decoration::Pointer
                                                 : Type::Decoration::Reference,
                                             TF.next()));
  for (auto &Dec : T.Decorations)
    Dec.fix();
}

static bool isBuiltinType(tok::TokenKind K) {
  switch (K) {
  case tok::kw_short:
  case tok::kw_long:
  case tok::kw___int64:
  case tok::kw___int128:
  case tok::kw_signed:
  case tok::kw_unsigned:
  case tok::kw__Complex:
  case tok::kw__Imaginary:
  case tok::kw_void:
  case tok::kw_char:
  case tok::kw_wchar_t:
  case tok::kw_char16_t:
  case tok::kw_char32_t:
  case tok::kw_int:
  case tok::kw_half:
  case tok::kw_float:
  case tok::kw_double:
  case tok::kw_bool:
  case tok::kw__Bool:
  case tok::kw__Decimal32:
  case tok::kw__Decimal64:
  case tok::kw__Decimal128:
  case tok::kw___vector:
    return true;
  default:
    return false;
  }
}
```

```
static bool isCVQualifier(tok::TokenKind K) {
  switch (K) {
  case tok::kw_const:
  case tok::kw_constexpr:
  case tok::kw_volatile:
  case tok::kw_register:
    return true;
  default:
    return false;
  }
}

static std::unique_ptr<Type> parseType(TokenFilter &TF, bool WithDecorations) {
  auto Guard = TF.guard();
  std::unique_ptr<Type> T = make_unique<Type>();

  while (isCVQualifier(TF.peekKind()) || checkKind(TF, tok::kw_typename))
    T->addNameQualifier(TF.next());

  if (checkKind(TF, tok::kw_auto)) {
    T->addNameQualifier(TF.next());
  } else if (isBuiltinType(TF.peekKind())) {
    while (isBuiltinType(TF.peekKind()))
      T->addNameQualifier(TF.next());
  } else if (!parseQualifiedID(TF, *T)) {
    return {};
  }
  while (isCVQualifier(TF.peekKind()))
    T->addNameQualifier(TF.next());

  if (WithDecorations)
    parseTypeDecorations(TF, *T);

  Guard.dismiss();
  return T;
}

static std::unique_ptr<VarDecl> parseVarDecl(TokenFilter &TF,
                                             Type *TypeName = 0,
                                             bool NameOptional = false,
                                             bool StopAtGreater = false) {
  auto Guard = TF.guard();
  auto VD = make_unique<VarDecl>();
  VarDecl &D = *VD;

  if (!TypeName) {
    D.VariableType = parseType(TF);
    if (!D.VariableType)
      return {};
  } else {
    D.VariableType = TypeName->cloneWithoutDecorations();
  }
  parseTypeDecorations(TF, *D.VariableType);

  if (checkKind(TF, tok::identifier)) {
    D.setName(TF.next());
  } else if (!NameOptional) {
    return {};
  }

  if (checkKind(TF, tok::equal)) {
    auto *EqualTok = TF.next();
    if (auto Value = parseExpr(TF, prec::Comma + 1, StopAtGreater)) {
      D.Value = VarInitialization();
      D.Value->setAssignmentOps(VarInitialization::ASSIGNMENT, EqualTok);
      D.Value->Value = std::move(Value);
    } else {
      return {};
    }
  } else {
    // TODO: var(init) and var{init} not yet implemented
  }
  Guard.dismiss();
  return VD;
}

static std::unique_ptr<Stmt> parseDeclStmt(TokenFilter &TF,
                                           bool WithSemi = true) {
  auto Guard = TF.guard();

  auto TypeName = parseType(TF, /*WithDecorations=*/false);
  if (!TypeName)
    return {};
  auto Declaration = make_unique<DeclStmt>();

  while (!TF.eof()) {
    if (checkKind(TF, tok::semi)) {
      if (Declaration->Decls.empty())
        return {};
      if (WithSemi)
        Declaration->setSemi(TF.next());
      Guard.dismiss();
      return std::move(Declaration);
    }
    if (auto D = parseVarDecl(TF, TypeName.get()))
      Declaration->Decls.push_back(std::move(D));
    else
      return {};

    if (checkKind(TF, tok::comma)) {
      Declaration->appendComma(TF.next());
    } else if (!checkKind(TF, tok::semi)) {
      return {};
    }
  }

  return {};
}

static bool parseDestructor(TokenFilter &TF, FunctionDecl &F) {
  auto Pos = TF.mark();

  int Tildes = 0;
  while (checkKind(TF, tok::tilde) || checkKind(TF, tok::identifier) ||
         checkKind(TF, tok::coloncolon)) {
    Tildes += checkKind(TF, tok::tilde);
    TF.next();
  }
  if (Tildes != 1)
    return false;

  if (!checkKind(TF, tok::l_paren))
    return false;

  TF.rewind(Pos);

  F.ReturnType = make_unique<Type>();

  while (checkKind(TF, tok::tilde) || checkKind(TF, tok::identifier) ||
         checkKind(TF, tok::coloncolon)) {
    if (checkKind(TF, tok::tilde))
      F.addNameQualifier(TF.next());
    else
      F.ReturnType->addNameQualifier(TF.next());
  }

  return true;
```

```
}

static bool isDeclSpecifier(tok::TokenKind K) {
  switch (K) {
  case tok::kw_friend:
  // case tok::kw_constexpr:
  // case tok::kw_const:
  // case tok::kw_mutable:
  case tok::kw_typedef:
  // case tok::kw_register:
  case tok::kw_static:
  // case tok::kw_thread_local:
  case tok::kw_extern:
  case tok::kw_inline:
  case tok::kw_virtual:
  case tok::kw_explicit:
    return true;
  default:
    return false;
  }
}

static std::unique_ptr<FunctionDecl>
parseFunctionDecl(TokenFilter &TF, bool NameOptional = false) {
  auto Guard = TF.guard();
  auto F = make_unique<FunctionDecl>();

  while (isDeclSpecifier(TF.peekKind()))
    F->addDeclSpecifier(TF.next());

  bool InDestructor = false;

  if (auto T = parseType(TF)) {
    F->ReturnType = std::move(T);
  } else if (NameOptional && parseDestructor(TF, *F)) {
    InDestructor = true;
  } else {
    return {};
  }

  if (!InDestructor) {
    if (!checkKind(TF, tok::identifier) && !checkKind(TF, tok::kw_operator)) {
      if (!NameOptional)
        return {};
    } else if (!parseQualifiedID(TF, *F, std::false_type{})) {
      return {};
    }
  }

  if (!checkKind(TF, tok::l_paren))
    return {};

  F->setLeftBrace(TF.next());
  while (!checkKind(TF, tok::r_paren)) {
    F->Params.push_back(parseVarDecl(TF, 0, true));
    if (!F->Params.back())
      return {};
    if (checkKind(TF, tok::comma))
      F->appendComma(TF.next());
    else
      break;
  }
  if (!checkKind(TF, tok::r_paren))
    return {};

  F->setRightBrace(TF.next());

  // if (InConstructor && checkKind(TF, tok::colon)) {
  // TODO: Don't skip initializer list and [[x]] and const
  while (!TF.eof() && !checkKind(TF, tok::l_brace) && !checkKind(TF, tok::semi))
    TF.next();
  //}

  if (checkKind(TF, tok::semi))
    F->setSemi(TF.next());
  Guard.dismiss();
  return std::move(F);
}

static std::unique_ptr<Stmt> skipUnparsable(TokenFilter &TF) {
  assert(!TF.eof());
  auto UB = make_unique<UnparsableBlock>();
  while (!TF.eof()) {
    auto Kind = TF.peekKind();
    UB->push_back(TF.next());
    if (Kind == tok::semi || Kind == tok::r_brace || Kind == tok::l_brace)
      break;
  }
  return std::move(UB);
}

static std::unique_ptr<Stmt> parseLabelStmt(TokenFilter &TF) {
  auto Guard = TF.guard();
  if (!(checkKind(TF, tok::identifier) || checkKind(TF, tok::kw_private) ||
        checkKind(TF, tok::kw_protected) || checkKind(TF, tok::kw_public)))
    return {};
  auto *LabelName = TF.next();
  if (!checkKind(TF, tok::colon))
    return {};
  Guard.dismiss();
  return make_unique<LabelStmt>(LabelName, TF.next());
}

static std::unique_ptr<PPInclude> parseIncludeDirective(RawTokenFilter &TF) {
  if (!checkKind(TF, tok::hash))
    return {};
  auto Guard = TF.guard();

  auto *HashTok = TF.next();
  if (TF.peek()->Tok().getIdentifierInfo()->getPPKeywordID() != tok::pp_include)
    return {};

  auto Inc = make_unique<PPInclude>();
  Inc->setHash(HashTok);
  Inc->setInclude(TF.next());
  Inc->Path = make_unique<PPString>();

  while (!checkKind(TF, tok::eod)) {
    Inc->Path->addToken(TF.next());
  }
  Inc->setEOD(TF.next());
  return Inc;
}

static std::unique_ptr<PPIf> parsePPIf(RawTokenFilter &TF) {
  if (!checkKind(TF, tok::hash))
    return {};
  auto Guard = TF.guard();

  auto *HashTok = TF.next();

  if (TF.peek()->Tok().getIdentifierInfo()->getPPKeywordID() != tok::pp_else &&
      TF.peek()->Tok().getIdentifierInfo()->getPPKeywordID() != tok::pp_if &&
      TF.peek()->Tok().getIdentifierInfo()->getPPKeywordID() != tok::pp_elif &&
      TF.peek()->Tok().getIdentifierInfo()->getPPKeywordID() != tok::pp_endif)
    return {};

  auto If = make_unique<PPIf>();
  If->setHash(HashTok);
  If->setKeyword(TF.next());

  auto Start = TF.mark();

  if (!checkKind(TF, tok::eod)) {
    while (!checkKind(TF, tok::eod))
      TF.next();
    assert(checkKind(TF, tok::eod));

    TokenFilter SubTF = TF.rangeAsTokenFilter(Start, TF.mark());

    auto SubStart = SubTF.mark();
    std::unique_ptr<ASTElement> Cond;
    if ((Cond = parseExpr(SubTF)) && checkKind(TF, tok::eod))
      If->Cond = std::move(Cond);
    else {
      SubTF.rewind(SubStart);
      auto UB = make_unique<UnparsableBlock>();
      while (!checkKind(SubTF, tok::eod))
        UB->push_back(SubTF.next());
      If->Cond = std::move(UB);
    }
  }

  assert(checkKind(TF, tok::eod));
  If->setEOD(TF.next());
  return If;
}

static std::unique_ptr<PPDirective> parsePPDirective(RawTokenFilter &TF) {
  assert(checkKind(TF, tok::hash));
  if (auto I = parseIncludeDirective(TF))
    return std::move(I);
  if (auto D = parsePPIf(TF))
    return std::move(D);
  auto UP = make_unique<UnparsablePP>();
  while (!checkKind(TF, tok::eod))
    UP->push_back(TF.next());
  return std::move(UP);
}

static std::unique_ptr<Stmt> parseAny(TokenFilter &TF,
                                      bool SkipUnparsable = true,
                                      bool NameOptional = false);

```

```
static bool parseScope(TokenFilter &TF, Scope &Sc, bool NameOptional = false) {
  if (checkKind(TF, tok::r_brace))
    return true;
  while (auto St = parseAny(TF, true, NameOptional)) {
    Sc.addStmt(std::move(St));
    if (TF.eof())
      return false;
    if (checkKind(TF, tok::r_brace))
      return true;
  }
  return checkKind(TF, tok::r_brace);
}

static std::unique_ptr<CompoundStmt> parseCompoundStmt(TokenFilter &TF) {
  if (!checkKind(TF, tok::l_brace))
    return {};
  auto C = make_unique<CompoundStmt>();
  C->setLeftBrace(TF.next());
  parseScope(TF, *C);
  if (checkKind(TF, tok::r_brace))
    C->setRightBrace(TF.next());
  // else: just pass
  return C;
}

static std::unique_ptr<Stmt> parseControlFlowBody(TokenFilter &TF) {
  return checkKind(TF, tok::l_brace) ? parseCompoundStmt(TF) : parseAny(TF);
}

static std::unique_ptr<ASTElement> parseCond(TokenFilter &TF,
                                             bool ForLoopInit = false) {
  if (ForLoopInit)
    if (auto D = parseDeclStmt(TF, /*WithSemi=*/false))
      return std::move(D);
  {
    auto Guard = TF.guard();
    if (auto D = parseVarDecl(TF)) {
      if (checkKind(TF, tok::r_paren)) {
        Guard.dismiss();
        return std::move(D);
      }
    }
  }
  if (auto E = parseExpr(TF))
    return std::move(E);

  auto UB = make_unique<UnparsableBlock>();
  int ParenOpen = 1;
  while (!TF.eof()) {
    if (checkKind(TF, tok::l_paren)) {
      ++ParenOpen;
    } else if (checkKind(TF, tok::r_paren)) {
      if (--ParenOpen == 0) {
        return std::move(UB);
      }
    }

    if (checkKind(TF, tok::l_brace) || checkKind(TF, tok::r_brace) ||
        checkKind(TF, tok::semi))
      return std::move(UB);

    UB->push_back(TF.next());
  }
  return std::move(UB);
}

static std::unique_ptr<Stmt> parseControlFlowStmt(TokenFilter &TF) {
  auto Guard = TF.guard();

  if (checkKind(TF, tok::kw_while)) {
    auto S = make_unique<WhileStmt>();

    S->setKeyword(TF.next());
    if (!checkKind(TF, tok::l_paren))
      return {};
    S->setLeftParen(TF.next());

    if (!(S->Cond = parseCond(TF)))
      return {};

    if (checkKind(TF, tok::r_paren))
      S->setRightParen(TF.next());

    S->Body = parseControlFlowBody(TF);

    Guard.dismiss();
    return std::move(S);
  }

  if (checkKind(TF, tok::kw_if)) {
    auto If = make_unique<IfStmt>();
    for (bool ElseBranch = false, First = true; !ElseBranch; First = false) {
      AnnotatedToken *KW1, *KW2 = nullptr;
      if (First && checkKind(TF, tok::kw_if)) {
        KW1 = TF.next();
      } else if (checkKind(TF, tok::kw_else)) {
        KW1 = TF.next();
        if (checkKind(TF, tok::kw_if))
          KW2 = TF.next();
        else
          ElseBranch = true;
      } else {
        break;
      }

      std::unique_ptr<ASTElement> Cond;
      AnnotatedToken *LPar = nullptr, *RPar = nullptr;

      if (!ElseBranch) {
        if (!checkKind(TF, tok::l_paren))
          return {};
        LPar = TF.next();

        if (!(Cond = parseCond(TF)))
          return {};

        if (checkKind(TF, tok::r_paren))
          RPar = TF.next();
      }

      auto Body = parseControlFlowBody(TF);

      If->addBranch(KW1, KW2, LPar, std::move(Cond), RPar, std::move(Body));
    }
    Guard.dismiss();
    return std::move(If);
  }

  if (checkKind(TF, tok::kw_for)) {
    auto S = make_unique<ForStmt>();

    S->setKeyword(TF.next());
    if (!checkKind(TF, tok::l_paren))
      return {};
    S->setLeftParen(TF.next());

    if (!checkKind(TF, tok::semi) &&
        !(S->Init = parseCond(TF, /*ForLoopInit=*/true)))
      return {};
    if (!checkKind(TF, tok::semi))
      return {};
    S->setSemi1(TF.next());
    if (!checkKind(TF, tok::semi) && !(S->Cond = parseCond(TF)))
      return {};
    if (!checkKind(TF, tok::semi))
      return {};
    S->setSemi2(TF.next());
    if (!checkKind(TF, tok::r_paren) && !(S->Inc = parseExpr(TF)))
      return {};

    if (checkKind(TF, tok::r_paren))
      S->setRightParen(TF.next());

    S->Body = parseControlFlowBody(TF);

    Guard.dismiss();
    return std::move(S);
  }

  return {};
}

static bool parseClassScope(TokenFilter &TF, ClassDecl &C) {
  if (!checkKind(TF, tok::l_brace))
    return false;

  C.setLeftBrace(TF.next());
  if (!parseScope(TF, C, true))
    return false;

  if (checkKind(TF, tok::r_brace))
    C.setRightBrace(TF.next());

  if (checkKind(TF, tok::semi))
    C.setSemi(TF.next());
  // else: just pass

  return true;
}

static std::unique_ptr<Stmt> parseNamespaceDecl(TokenFilter &TF) {
  if (!checkKind(TF, tok::kw_namespace))
    return {};
  auto Guard = TF.guard();

  AnnotatedToken *NSTok = TF.next(), *NameTok = nullptr;
  if (checkKind(TF, tok::identifier))
    NameTok = TF.next();

  if (!checkKind(TF, tok::l_brace))
    return {};

  auto NS = make_unique<NamespaceDecl>();
  NS->setNamespace(NSTok);
  NS->setName(NameTok);
  NS->setLeftBrace(TF.next());

  parseScope(TF, *NS);

  if (checkKind(TF, tok::r_brace))
    NS->setRightBrace(TF.next());

  Guard.dismiss();
  return std::move(NS);
}

static std::unique_ptr<ClassDecl> parseClassDecl(TokenFilter &TF) {
  if (!(checkKind(TF, tok::kw_class) || checkKind(TF, tok::kw_struct) ||
        checkKind(TF, tok::kw_union) || checkKind(TF, tok::kw_enum)))
    return {};

  auto Guard = TF.guard();

  auto C = make_unique<ClassDecl>();
  C->setClass(TF.next());

  if (!(C->Name = parseType(TF)))
    return {};

  if (checkKind(TF, tok::colon)) {
    C->setColon(TF.next());
    bool Skip = true;
    for (;;) {
      AnnotatedToken *Accessibility = nullptr;
      if (checkKind(TF, tok::kw_private) || checkKind(TF, tok::kw_protected) ||
          checkKind(TF, tok::kw_public))
        Accessibility = TF.next();
      auto T = parseType(TF, false);
      if (!T)
        break;
      if (checkKind(TF, tok::l_brace)) {
        C->addBaseClass(Accessibility, std::move(T), nullptr);
        Skip = false;
        break;
      }
      if (!checkKind(TF, tok::comma))
        break;
      C->addBaseClass(Accessibility, std::move(T), TF.next());
    }
    if (Skip) {
      while (!checkKind(TF, tok::l_brace))
        TF.next();
    }
  }

  if (checkKind(TF, tok::semi))
    C->setSemi(TF.next());
  else
    parseClassScope(TF, *C);

  Guard.dismiss();
  return C;
}

static std::unique_ptr<TemplateParameterType>
parseTemplateParameterType(TokenFilter &TF) {
  if (!(checkKind(TF, tok::kw_typename) || checkKind(TF, tok::kw_class)))
    return {};
  auto Guard = TF.guard();

  auto TPT = make_unique<TemplateParameterType>();
  TPT->setKeyword(TF.next());
  if (!checkKind(TF, tok::identifier))
    return {};
  TPT->setName(TF.next());

  if (checkKind(TF, tok::equal)) {
    TPT->setEqual(TF.next());
    if (!(TPT->DefaultType = parseType(TF)))
      return {};
  }

  Guard.dismiss();
  return TPT;
}
static std::unique_ptr<TemplateDecl> parseTemplateDecl(TokenFilter &TF) {
  if (!checkKind(TF, tok::kw_template))
    return {};

  auto Guard = TF.guard();
  auto T = make_unique<TemplateDecl>();
  T->setKeyword(TF.next());

  if (!checkKind(TF, tok::less))
    return {};
  T->setLess(TF.next());

  while (!checkKind(TF, tok::greater)) {
    if (auto D = parseVarDecl(TF, /*TypeName=*/0, /*NameOptional*/ false,
                              /*StopAtGreater=*/true))
      T->addParam(std::move(D));
    else if (auto TPT = parseTemplateParameterType(TF))
      T->addParam(std::move(TPT));
    else
      return {};

    if (checkKind(TF, tok::comma))
      T->addComma(TF.next());
    else if (!checkKind(TF, tok::greater))
      return {};
  }

  assert(checkKind(TF, tok::greater));
  T->setGreater(TF.next());

  if (auto F = parseFunctionDecl(TF))
    T->Templated = std::move(F);
  else if (auto C = parseClassDecl(TF))
    T->Templated = std::move(C);
  else
    return {};

  Guard.dismiss();
  return T;
}

static std::unique_ptr<Stmt> parseAny(TokenFilter &TF, bool SkipUnparsable,
                                      bool NameOptional) {
  if (auto S = parseDeclStmt(TF))
    return S;
  if (auto S = parseReturnStmt(TF))
    return S;
  if (auto S = parseLabelStmt(TF))
    return S;
  if (auto S = parseControlFlowStmt(TF))
    return S;
  if (auto S = parseTemplateDecl(TF))
    return std::move(S);
  if (auto S = parseFunctionDecl(TF, NameOptional)) {
    if (checkKind(TF, tok::semi))
      S->setSemi(TF.next());
    else if (checkKind(TF, tok::l_brace)) {
      S->Body = parseCompoundStmt(TF);
    }
    return std::move(S);
  }
  if (auto S = parseNamespaceDecl(TF))
    return S;

  if (auto S = parseClassDecl(TF)) {
    if (checkKind(TF, tok::semi))
      S->setSemi(TF.next());
    else if (checkKind(TF, tok::l_brace)) {
      parseClassScope(TF, *S);
    }
    return std::move(S);
  }
  {
    auto Guard = TF.guard();
    if (auto E = parseExpr(TF)) {
      if (checkKind(TF, tok::semi)) {
        Guard.dismiss();
        return make_unique<ExprLineStmt>(std::move(E), TF.next());
      }
    }
  }
  return SkipUnparsable ? skipUnparsable(TF) : std::unique_ptr<Stmt>();
}

TranslationUnit fuzzyparse(AnnotatedToken *first, AnnotatedToken *last) {
  TranslationUnit TU;
  {
    BasicTokenFilter<false> TF(first, last);
    while (!TF.eof()) {
      if (TF.peekKind() == tok::hash && TF.peek()->Tok().isAtStartOfLine())
        TU.addPPDirective(parsePPDirective(TF));
      TF.next();
    }
  }
  {
    TokenFilter TF(first, last);
    while (!TF.eof())
      TU.addStmt(parseAny(TF));
  }
  return TU;
}

} // end namespace fuzzy
} // end namespace clang
```