

```

1  //===== FuzzyParser.cpp - clang-highlight -----*- C++ -*-=====//
2  //
3  //                                The LLVM Compiler Infrastructure
4  //
5  // This file is distributed under the University of Illinois Open Source
6  // License. See LICENSE.TXT for details.
7  //
8  //=====
9  #include "llvm/Support/Debug.h"
10 #include "llvm/ADT/STLExtras.h"
11 #include "clang/Basic/IdentifierTable.h"
12 #include "clang/Basic/OperatorPrecedence.h"
13 #include "FuzzyAST.h"
14
15 using namespace llvm;
16
17 namespace clang {
18 namespace fuzzy {
19
20 namespace {
21 template <bool SkipPreprocessor> class BasicTokenFilter {
22     AnnotatedToken *First, *Last;
23
24     void skipWhitespaces() {
25         for (;;) {
26             while (First != Last && (First->getTokenKind() == tok::unknown ||
27                                     First->getTokenKind() == tok::comment))
28                 ++First;
29
30             if (SkipPreprocessor && First->getTokenKind() == tok::hash &&
31                 First->Tok().isAtStartOfLine())
32                 while (First != Last && First++->getTokenKind() != tok::eod)
33                     ;
34             else
35                 break;
36         }
37         assert(First <= Last);
38     }
39
40 public:
41     BasicTokenFilter(AnnotatedToken *First, AnnotatedToken *Last)
42         : First(First), Last(Last) {
43         skipWhitespaces();
44     }
45
46     AnnotatedToken *next() {

```

```

47     assert(!eof());
48     auto Ret = First++;
49     skipWhitespaces();
50     assert(Ret->getTokenKind() != tok::raw_identifier);
51     return Ret;
52 }
53
54 class TokenFilterState {
55     friend class BasicTokenFilter;
56     TokenFilterState(AnnotatedToken *First, AnnotatedToken *Last)
57         : First(First), Last(Last) {}
58     AnnotatedToken *First, *Last;
59 };
60
61 TokenFilterState mark() const { return TokenFilterState(First, Last); }
62 void rewind(TokenFilterState State) {
63     First = State.First;
64     Last = State.Last;
65 }
66
67 BasicTokenFilter<true> rangeAsTokenFilter(TokenFilterState From,
68                                           TokenFilterState To) const {
69     assert(From.Last == To.Last);
70     assert(From.First <= To.First);
71     assert(To.First < To.Last);
72     return BasicTokenFilter<true>(From.First, To.First + 1);
73 }
74
75 class TokenFilterGuard {
76     friend class BasicTokenFilter;
77     TokenFilterGuard(BasicTokenFilter *TF, TokenFilterState State)
78         : TF(TF), State(State) {}
79
80 public:
81     ~TokenFilterGuard() {
82         if (TF)
83             TF->rewind(State);
84     }
85     void dismiss() { TF = nullptr; }
86     BasicTokenFilter *TF;
87     TokenFilterState State;
88 };
89 TokenFilterGuard guard() { return TokenFilterGuard(this, mark()); }
90
91 AnnotatedToken *peek() { return First; }
92 const AnnotatedToken *peek() const { return First; }

```

```

93     tok::TokenKind peekKind() const { return First->getTokenKind(); }
94
95     bool eof() const { return peekKind() == tok::eof; }
96 };
97 using TokenFilter = BasicTokenFilter<true>;
98 using RawTokenFilter = BasicTokenFilter<false>;
99 } // end anonymous namespace
100
101 template <bool B>
102 static bool checkKind(BasicTokenFilter<B> &TF, tok::TokenKind Kind) {
103     return TF.peekKind() == Kind;
104 }
105
106 static int PrecedenceUnaryOperator = prec::PointerToMember + 1;
107 static int PrecedenceArrowAndPeriod = prec::PointerToMember + 2;
108
109 static std::unique_ptr<Expr> parseExpr(TokenFilter &TF, int Precedence = 1,
110                                     bool StopAtGreater = false);
111
112 static std::unique_ptr<Type> parseType(TokenFilter &TF,
113                                     bool WithDecorations = true);
114
115 static std::unique_ptr<Expr> parseUnaryOperator(TokenFilter &TF) {
116     if (checkKind(TF, tok::plus) || checkKind(TF, tok::minus) ||
117         checkKind(TF, tok::exclaim) || checkKind(TF, tok::tilde) ||
118         checkKind(TF, tok::star) || checkKind(TF, tok::amp) ||
119         checkKind(TF, tok::plusplus) || checkKind(TF, tok::minusminus)) {
120         AnnotatedToken *Op = TF.next();
121         auto Operand = parseUnaryOperator(TF);
122         if (!Operand)
123             return {};
124         return make_unique<UnaryOperator>(Op, std::move(Operand));
125     }
126
127     return parseExpr(TF, PrecedenceArrowAndPeriod);
128 }
129
130 static std::unique_ptr<Expr>
131 parseCallExpr(TokenFilter &TF, std::unique_ptr<DeclRefExpr> FunctionName) {
132     assert(checkKind(TF, tok::l_paren));
133     auto Func = make_unique<CallExpr>(std::move(FunctionName));
134     Func->setLeftParen(TF.next());
135     while (!checkKind(TF, tok::r_paren)) {
136         Func->Args.push_back(parseExpr(TF, prec::Comma + 1));
137         if (checkKind(TF, tok::comma))
138             Func->appendComma(TF.next());

```

```

139         else
140             break;
141     }
142     if (checkKind(TF, tok::r_paren)) {
143         Func->setRightParen(TF.next());
144         return std::move(Func);
145     }
146     return {};
147 }
148
149 static bool isLiteralOrConstant(tok::TokenKind K) {
150     if (isLiteral(K))
151         return true;
152
153     switch (K) {
154     case tok::kw_this:
155     case tok::kw_true:
156     case tok::kw_false:
157     case tok::kw___objc_yes:
158     case tok::kw___objc_no:
159     case tok::kw_nullptr:
160         return true;
161     default:
162         return false;
163     }
164 }
165
166 template <typename QualOwner>
167 static bool parseNamespaceQualifiers(TokenFilter &TF, QualOwner &Qual) {
168     auto Guard = TF.guard();
169
170     if (checkKind(TF, tok::kw_operator)) {
171         Qual.addNameQualifier(TF.next());
172         if (!TF.peek())
173             return false;
174         Qual.addNameQualifier(TF.next());
175         Guard.dismiss();
176         return true;
177     }
178
179     bool GlobalNamespaceColon = true;
180     do {
181         if (checkKind(TF, tok::coloncolon))
182             Qual.addNameQualifier(TF.next());
183         else if (!GlobalNamespaceColon)
184             return false;

```

```

185     GlobalNamespaceColon = false;
186     if (!checkKind(TF, tok::identifier))
187         return false;
188     Qual.addNameQualifier(TF.next());
189 } while (checkKind(TF, tok::coloncolon));
190
191 Guard.dismiss();
192 return true;
193 }
194
195 template <typename QualOwner>
196 static bool parseTemplateArgs(TokenFilter &TF, QualOwner &Qual,
197                             std::false_type) {
198     return true;
199 }
200 template <typename QualOwner>
201 static bool parseTemplateArgs(TokenFilter &TF, QualOwner &Qual,
202                             std::true_type) {
203     auto Guard = TF.guard();
204
205     if (checkKind(TF, tok::less)) {
206         Qual.makeTemplateArgs();
207         bool isFirst = true;
208         do {
209             Qual.addTemplateSeparator(TF.next());
210
211             if (isFirst && checkKind(TF, tok::greater))
212                 break;
213             isFirst = false;
214
215             if (auto Arg = parseType(TF))
216                 Qual.addTemplateArgument(std::move(Arg));
217             else if (auto E = parseExpr(TF, prec::Comma + 1, /*StopAtGreater=*/true))
218                 Qual.addTemplateArgument(std::move(E));
219             else
220                 return false;
221         } while (checkKind(TF, tok::comma));
222         if (!checkKind(TF, tok::greater))
223             return false;
224         Qual.addTemplateSeparator(TF.next());
225     }
226
227     Guard.dismiss();
228     return true;
229 }
230

```

```

231 template <typename QualOwner, typename WithTemplateArgs = std::true_type>
232 static bool parseQualifiedID(TokenFilter &TF, QualOwner &Qual,
233                             WithTemplateArgs WTA = std::true_type{}) {
234     auto Guard = TF.guard();
235     if (parseNamespaceQualifiers(TF, Qual) && parseTemplateArgs(TF, Qual, WTA)) {
236         Guard.dismiss();
237         return true;
238     }
239     return false;
240 }
241
242 static std::unique_ptr<Expr> parseExpr(TokenFilter &TF, int Precedence,
243                                       bool StopAtGreater) {
244     if (!TF.peek())
245         return {};
246
247     if (Precedence == PrecedenceUnaryOperator)
248         return parseUnaryOperator(TF);
249
250     if (Precedence > PrecedenceArrowAndPeriod) {
251         if (isLiteralOrConstant(TF.peekKind()))
252             return make_unique<LiteralConstant>(TF.next());
253
254         if (checkKind(TF, tok::l_paren)) {
255             auto Left = TF.next();
256             auto Val = parseExpr(TF, 1, false);
257             if (!checkKind(TF, tok::r_paren))
258                 return {};
259             auto Right = TF.next();
260             return make_unique<ParenExpr>(Left, std::move(Val), Right);
261         }
262
263         if (checkKind(TF, tok::identifier) || checkKind(TF, tok::coloncolon)) {
264             auto DR = make_unique<DeclRefExpr>();
265             if (!parseQualifiedID(TF, *DR) &&
266                 !parseQualifiedID(TF, *DR, std::false_type{}))
267                 return {};
268             if (checkKind(TF, tok::l_paren))
269                 return parseCallExpr(TF, std::move(DR));
270             std::unique_ptr<Expr> Ret = std::move(DR);
271             while (checkKind(TF, tok::plusplus) || checkKind(TF, tok::minusminus))
272                 Ret = make_unique<UnaryOperator>(TF.next(), std::move(Ret));
273             return std::move(Ret);
274         }
275
276         return {};

```

```

277     }
278     auto LeftExpr = parseExpr(TF, Precedence + 1, StopAtGreater);
279     if (!LeftExpr)
280         return {};
281
282     while (!TF.eof()) {
283         if (StopAtGreater && checkKind(TF, tok::greater))
284             break;
285
286         int CurrentPrecedence = getBinOpPrecedence(TF.peekKind(), true, true);
287         if (checkKind(TF, tok::period) || checkKind(TF, tok::arrow))
288             CurrentPrecedence = PrecedenceArrowAndPeriod;
289         if (CurrentPrecedence == 0)
290             return LeftExpr;
291
292         assert(CurrentPrecedence <= Precedence);
293         if (CurrentPrecedence < Precedence)
294             break;
295         assert(CurrentPrecedence == Precedence);
296
297         AnnotatedToken *OperatorTok = TF.next();
298
299         auto RightExpr = parseExpr(TF, Precedence + 1, StopAtGreater);
300         if (!RightExpr)
301             return {};
302
303         LeftExpr = make_unique<BinaryOperator>(std::move(LeftExpr),
304                                                std::move(RightExpr), OperatorTok);
305     }
306
307     return LeftExpr;
308 }
309
310 static std::unique_ptr<Stmt> parseReturnStmt(TokenFilter &TF) {
311     auto Guard = TF.guard();
312     if (!checkKind(TF, tok::kw_return))
313         return {};
314     auto *Return = TF.next();
315     std::unique_ptr<Expr> Body;
316     if (!checkKind(TF, tok::semi)) {
317         Body = parseExpr(TF);
318         if (!Body || !checkKind(TF, tok::semi))
319             return {};
320     }
321     assert(checkKind(TF, tok::semi));
322     auto *Semi = TF.next();

```

```

323     Guard.dismiss();
324     return make_unique<ReturnStmt>(Return, std::move(Body), Semi);
325 }
326
327 static void parseTypeDecorations(TokenFilter &TF, Type &T) {
328     // TODO: add const and volatile
329     while (checkKind(TF, tok::star) || checkKind(TF, tok::amp) ||
330           checkKind(TF, tok::ampamp))
331         T.Decorations.push_back(Type::Decoration(checkKind(TF, tok::star)
332                                                    ? Type::Decoration::Pointer
333                                                    : Type::Decoration::Reference,
334                                                    TF.next()));
335     for (auto &Dec : T.Decorations)
336         Dec.fix();
337 }
338
339 static bool isBuiltinType(tok::TokenKind K) {
340     switch (K) {
341     case tok::kw_short:
342     case tok::kw_long:
343     case tok::kw___int64:
344     case tok::kw___int128:
345     case tok::kw_signed:
346     case tok::kw_unsigned:
347     case tok::kw__Complex:
348     case tok::kw__Imaginary:
349     case tok::kw_void:
350     case tok::kw_char:
351     case tok::kw_wchar_t:
352     case tok::kw_char16_t:
353     case tok::kw_char32_t:
354     case tok::kw_int:
355     case tok::kw_half:
356     case tok::kw_float:
357     case tok::kw_double:
358     case tok::kw_bool:
359     case tok::kw__Bool:
360     case tok::kw__Decimal32:
361     case tok::kw__Decimal64:
362     case tok::kw__Decimal128:
363     case tok::kw___vector:
364         return true;
365     default:
366         return false;
367     }
368 }

```



```

369
370 static bool isCVQualifier(tok::TokenKind K) {
371     switch (K) {
372         case tok::kw_const:
373         case tok::kw_constexpr:
374         case tok::kw_volatile:
375         case tok::kw_register:
376             return true;
377         default:
378             return false;
379     }
380 }
381
382 static std::unique_ptr<Type> parseType(TokenFilter &TF, bool WithDecorations) {
383     auto Guard = TF.guard();
384     std::unique_ptr<Type> T = make_unique<Type>();
385
386     while (isCVQualifier(TF.peekKind()) || checkKind(TF, tok::kw_typename))
387         T->addNameQualifier(TF.next());
388
389     if (checkKind(TF, tok::kw_auto)) {
390         T->addNameQualifier(TF.next());
391     } else if (isBuiltinType(TF.peekKind())) {
392         while (isBuiltinType(TF.peekKind()))
393             T->addNameQualifier(TF.next());
394     } else if (!parseQualifiedID(TF, *T)) {
395         return {};
396     }
397     while (isCVQualifier(TF.peekKind()))
398         T->addNameQualifier(TF.next());
399
400     if (WithDecorations)
401         parseTypeDecorations(TF, *T);
402
403     Guard.dismiss();
404     return T;
405 }
406
407 static std::unique_ptr<VarDecl> parseVarDecl(TokenFilter &TF,
408                                             Type *TypeName = 0,
409                                             bool NameOptional = false,
410                                             bool StopAtGreater = false) {
411     auto Guard = TF.guard();
412     auto VD = make_unique<VarDecl>();
413     VarDecl &D = *VD;
414

```

```

415     if (!TypeName) {
416         D.VariableType = parseType(TF);
417         if (!D.VariableType)
418             return {};
419     } else {
420         D.VariableType = TypeName->cloneWithoutDecorations();
421     }
422     parseTypeDecorations(TF, *D.VariableType);
423
424     if (checkKind(TF, tok::identifier)) {
425         D.setName(TF.next());
426     } else if (!NameOptional) {
427         return {};
428     }
429
430     if (checkKind(TF, tok::equal)) {
431         auto *EqualTok = TF.next();
432         if (auto Value = parseExpr(TF, prec::Comma + 1, StopAtGreater)) {
433             D.Value = VarInitialization();
434             D.Value->setAssignmentOps(VarInitialization::ASSIGNMENT, EqualTok);
435             D.Value->Value = std::move(Value);
436         } else {
437             return {};
438         }
439     } else {
440         // TODO: var(init) and var{init} not yet implemented
441     }
442     Guard.dismiss();
443     return VD;
444 }
445
446 static std::unique_ptr<Stmt> parseDeclStmt(TokenFilter &TF,
447                                           bool WithSemi = true) {
448     auto Guard = TF.guard();
449
450     auto TypeName = parseType(TF, /*WithDecorations=*/false);
451     if (!TypeName)
452         return {};
453     auto Declaration = make_unique<DeclStmt>();
454
455     while (!TF.eof()) {
456         if (checkKind(TF, tok::semi)) {
457             if (Declaration->Decls.empty())
458                 return {};
459             if (WithSemi)
460                 Declaration->setSemi(TF.next());

```

```

461     Guard.dismiss();
462     return std::move(Declaration);
463 }
464 if (auto D = parseVarDecl(TF, TypeName.get()))
465     Declaration->Decls.push_back(std::move(D));
466 else
467     return {};
468
469 if (checkKind(TF, tok::comma)) {
470     Declaration->appendComma(TF.next());
471 } else if (!checkKind(TF, tok::semi)) {
472     return {};
473 }
474 }
475
476 return {};
477 }
478
479 static bool parseDestructor(TokenFilter &TF, FunctionDecl &F) {
480     auto Pos = TF.mark();
481
482     int Tildes = 0;
483     while (checkKind(TF, tok::tilde) || checkKind(TF, tok::identifier) ||
484            checkKind(TF, tok::coloncolon)) {
485         Tildes += checkKind(TF, tok::tilde);
486         TF.next();
487     }
488     if (Tildes != 1)
489         return false;
490
491     if (!checkKind(TF, tok::l_paren))
492         return false;
493
494     TF.rewind(Pos);
495
496     F.ReturnType = make_unique<Type>();
497
498     while (checkKind(TF, tok::tilde) || checkKind(TF, tok::identifier) ||
499            checkKind(TF, tok::coloncolon)) {
500         if (checkKind(TF, tok::tilde))
501             F.addNameQualifier(TF.next());
502         else
503             F.ReturnType->addNameQualifier(TF.next());
504     }
505
506     return true;

```

```

507 }
508
509 static bool isDeclSpecifier(tok::TokenKind K) {
510     switch (K) {
511         case tok::kw_friend:
512             // case tok::kw_constexpr:
513             // case tok::kw_const:
514             // case tok::kw_mutable:
515         case tok::kw_typedef:
516             // case tok::kw_register:
517         case tok::kw_static:
518             // case tok::kw_thread_local:
519         case tok::kw_extern:
520         case tok::kw_inline:
521         case tok::kw_virtual:
522         case tok::kw_explicit:
523             return true;
524         default:
525             return false;
526     }
527 }
528
529 static std::unique_ptr<FunctionDecl>
530 parseFunctionDecl(TokenFilter &TF, bool NameOptional = false) {
531     auto Guard = TF.guard();
532     auto F = make_unique<FunctionDecl>();
533
534     while (isDeclSpecifier(TF.peekKind()))
535         F->addDeclSpecifier(TF.next());
536
537     bool InDestructor = false;
538
539     if (auto T = parseType(TF)) {
540         F->ReturnType = std::move(T);
541     } else if (NameOptional && parseDestructor(TF, *F)) {
542         InDestructor = true;
543     } else {
544         return {};
545     }
546
547     if (!InDestructor) {
548         if (!checkKind(TF, tok::identifier) && !checkKind(TF, tok::kw_operator)) {
549             if (!NameOptional)
550                 return {};
551             } else if (!parseQualifiedID(TF, *F, std::false_type{})) {
552                 return {};

```

```

553     }
554 }
555
556 if (!checkKind(TF, tok::l_paren))
557     return {};
558
559 F->setLeftBrace(TF.next());
560 while (!checkKind(TF, tok::r_paren)) {
561     F->Params.push_back(parseVarDecl(TF, 0, true));
562     if (!F->Params.back())
563         return {};
564     if (checkKind(TF, tok::comma))
565         F->appendComma(TF.next());
566     else
567         break;
568 }
569 if (!checkKind(TF, tok::r_paren))
570     return {};
571
572 F->setRightBrace(TF.next());
573
574 // if (InConstructor && checkKind(TF, tok::colon)) {
575 // TODO: Don't skip initializer list and [[x]] and const
576 while (!TF.eof() && !checkKind(TF, tok::l_brace) && !checkKind(TF, tok::semi))
577     TF.next();
578 //}
579
580 if (checkKind(TF, tok::semi))
581     F->setSemi(TF.next());
582 Guard.dismiss();
583 return std::move(F);
584 }
585
586 static std::unique_ptr<Stmt> skipUnparsable(TokenFilter &TF) {
587     assert(!TF.eof());
588     auto UB = make_unique<UnparsableBlock>();
589     while (!TF.eof()) {
590         auto Kind = TF.peekKind();
591         UB->push_back(TF.next());
592         if (Kind == tok::semi || Kind == tok::r_brace || Kind == tok::l_brace)
593             break;
594     }
595     return std::move(UB);
596 }
597
598 static std::unique_ptr<Stmt> parseLabelStmt(TokenFilter &TF) {

```

```

599     auto Guard = TF.guard();
600     if (!(checkKind(TF, tok::identifier) || checkKind(TF, tok::kw_private) ||
601           checkKind(TF, tok::kw_protected) || checkKind(TF, tok::kw_public)))
602         return {};
603     auto *LabelName = TF.next();
604     if (!checkKind(TF, tok::colon))
605         return {};
606     Guard.dismiss();
607     return make_unique<LabelStmt>(LabelName, TF.next());
608 }
609
610 static std::unique_ptr<PPInclude> parseIncludeDirective(RawTokenFilter &TF) {
611     if (!checkKind(TF, tok::hash))
612         return {};
613     auto Guard = TF.guard();
614
615     auto *HashTok = TF.next();
616     if (TF.peek()->Tok().getIdentiferInfo()->getPPKeywordID() != tok::pp_include)
617         return {};
618
619     auto Inc = make_unique<PPInclude>();
620     Inc->setHash(HashTok);
621     Inc->setInclude(TF.next());
622     Inc->Path = make_unique<PPString>();
623
624     while (!checkKind(TF, tok::eod)) {
625         Inc->Path->addToken(TF.next());
626     }
627     Inc->setEOD(TF.next());
628     return Inc;
629 }
630
631 static std::unique_ptr<PPIf> parsePPIf(RawTokenFilter &TF) {
632     if (!checkKind(TF, tok::hash))
633         return {};
634     auto Guard = TF.guard();
635
636     auto *HashTok = TF.next();
637
638     if (TF.peek()->Tok().getIdentiferInfo()->getPPKeywordID() != tok::pp_else &&
639         TF.peek()->Tok().getIdentiferInfo()->getPPKeywordID() != tok::pp_if &&
640         TF.peek()->Tok().getIdentiferInfo()->getPPKeywordID() != tok::pp_elif &&
641         TF.peek()->Tok().getIdentiferInfo()->getPPKeywordID() != tok::pp_endif)
642         return {};
643
644     auto If = make_unique<PPIf>();

```

```

645     If->setHash(HashTok);
646     If->setKeyword(TF.next());
647
648     auto Start = TF.mark();
649
650     if (!checkKind(TF, tok::eod)) {
651         while (!checkKind(TF, tok::eod))
652             TF.next();
653         assert(checkKind(TF, tok::eod));
654
655         TokenFilter SubTF = TF.rangeAsTokenFilter(Start, TF.mark());
656
657         auto SubStart = SubTF.mark();
658         std::unique_ptr<ASTElement> Cond;
659         if ((Cond = parseExpr(SubTF)) && checkKind(TF, tok::eod))
660             If->Cond = std::move(Cond);
661         else {
662             SubTF.rewind(SubStart);
663             auto UB = make_unique<UnparsableBlock>();
664             while (!checkKind(SubTF, tok::eod))
665                 UB->push_back(SubTF.next());
666             If->Cond = std::move(UB);
667         }
668     }
669
670     assert(checkKind(TF, tok::eod));
671     If->setEOD(TF.next());
672     return If;
673 }
674
675 static std::unique_ptr<PPDirective> parsePPDirective(RawTokenFilter &TF) {
676     assert(checkKind(TF, tok::hash));
677     if (auto I = parseIncludeDirective(TF))
678         return std::move(I);
679     if (auto D = parsePPIf(TF))
680         return std::move(D);
681     auto UP = make_unique<UnparsablePP>();
682     while (!checkKind(TF, tok::eod))
683         UP->push_back(TF.next());
684     return std::move(UP);
685 }
686
687 static std::unique_ptr<Stmt> parseAny(TokenFilter &TF,
688                                     bool SkipUnparsable = true,
689                                     bool NameOptional = false);
690

```

```

691 static bool parseScope(TokenFilter &TF, Scope &Sc, bool NameOptional = false) {
692     if (checkKind(TF, tok::r_brace))
693         return true;
694     while (auto St = parseAny(TF, true, NameOptional)) {
695         Sc.addStmt(std::move(St));
696         if (TF.eof())
697             return false;
698         if (checkKind(TF, tok::r_brace))
699             return true;
700     }
701     return checkKind(TF, tok::r_brace);
702 }
703
704 static std::unique_ptr<CompoundStmt> parseCompoundStmt(TokenFilter &TF) {
705     if (!checkKind(TF, tok::l_brace))
706         return {};
707     auto C = make_unique<CompoundStmt>();
708     C->setLeftBrace(TF.next());
709     parseScope(TF, *C);
710     if (checkKind(TF, tok::r_brace))
711         C->setRightBrace(TF.next());
712     // else: just pass
713     return C;
714 }
715
716 static std::unique_ptr<Stmt> parseControlFlowBody(TokenFilter &TF) {
717     return checkKind(TF, tok::l_brace) ? parseCompoundStmt(TF) : parseAny(TF);
718 }
719
720 static std::unique_ptr<ASTElement> parseCond(TokenFilter &TF,
721                                             bool ForLoopInit = false) {
722     if (ForLoopInit)
723         if (auto D = parseDeclStmt(TF, /*WithSemi=*/false))
724             return std::move(D);
725     {
726         auto Guard = TF.guard();
727         if (auto D = parseVarDecl(TF)) {
728             if (checkKind(TF, tok::r_paren)) {
729                 Guard.dismiss();
730                 return std::move(D);
731             }
732         }
733     }
734     if (auto E = parseExpr(TF))
735         return std::move(E);
736

```



```

737     auto UB = make_unique<UnparsableBlock>();
738     int ParenOpen = 1;
739     while (!TF.eof()) {
740         if (checkKind(TF, tok::l_paren)) {
741             ++ParenOpen;
742         } else if (checkKind(TF, tok::r_paren)) {
743             if (--ParenOpen == 0) {
744                 return std::move(UB);
745             }
746         }
747
748         if (checkKind(TF, tok::l_brace) || checkKind(TF, tok::r_brace) ||
749             checkKind(TF, tok::semi))
750             return std::move(UB);
751
752         UB->push_back(TF.next());
753     }
754     return std::move(UB);
755 }
756
757 static std::unique_ptr<Stmt> parseControlFlowStmt(TokenFilter &TF) {
758     auto Guard = TF.guard();
759
760     if (checkKind(TF, tok::kw_while)) {
761         auto S = make_unique<WhileStmt>();
762
763         S->setKeyword(TF.next());
764         if (!checkKind(TF, tok::l_paren))
765             return {};
766         S->setLeftParen(TF.next());
767
768         if (!(S->Cond = parseCond(TF)))
769             return {};
770
771         if (checkKind(TF, tok::r_paren))
772             S->setRightParen(TF.next());
773
774         S->Body = parseControlFlowBody(TF);
775
776         Guard.dismiss();
777         return std::move(S);
778     }
779
780     if (checkKind(TF, tok::kw_if)) {
781         auto If = make_unique<IfStmt>();
782         for (bool ElseBranch = false, First = true; !ElseBranch; First = false) {

```

```

783     AnnotatedToken *KW1, *KW2 = nullptr;
784     if (First && checkKind(TF, tok::kw_if)) {
785         KW1 = TF.next();
786     } else if (checkKind(TF, tok::kw_else)) {
787         KW1 = TF.next();
788         if (checkKind(TF, tok::kw_if))
789             KW2 = TF.next();
790         else
791             ElseBranch = true;
792     } else {
793         break;
794     }
795
796     std::unique_ptr<ASTElement> Cond;
797     AnnotatedToken *LPar = nullptr, *RPar = nullptr;
798
799     if (!ElseBranch) {
800         if (!checkKind(TF, tok::l_paren))
801             return {};
802         LPar = TF.next();
803
804         if (!(Cond = parseCond(TF)))
805             return {};
806
807         if (checkKind(TF, tok::r_paren))
808             RPar = TF.next();
809     }
810
811     auto Body = parseControlFlowBody(TF);
812
813     If->addBranch(KW1, KW2, LPar, std::move(Cond), RPar, std::move(Body));
814 }
815 Guard.dismiss();
816 return std::move(If);
817 }
818
819 if (checkKind(TF, tok::kw_for)) {
820     auto S = make_unique<ForStmt>();
821
822     S->setKeyword(TF.next());
823     if (!checkKind(TF, tok::l_paren))
824         return {};
825     S->setLeftParen(TF.next());
826
827     if (!checkKind(TF, tok::semi) &&
828         !(S->Init = parseCond(TF, /*ForLoopInit=*/true)))

```

```

829         return {};
830     if (!checkKind(TF, tok::semi))
831         return {};
832     S->setSemi1(TF.next());
833     if (!checkKind(TF, tok::semi) && !(S->Cond = parseCond(TF)))
834         return {};
835     if (!checkKind(TF, tok::semi))
836         return {};
837     S->setSemi2(TF.next());
838     if (!checkKind(TF, tok::r_paren) && !(S->Inc = parseExpr(TF)))
839         return {};
840
841     if (checkKind(TF, tok::r_paren))
842         S->setRightParen(TF.next());
843
844     S->Body = parseControlFlowBody(TF);
845
846     Guard.dismiss();
847     return std::move(S);
848 }
849
850 return {};
851 }
852
853 static bool parseClassScope(TokenFilter &TF, ClassDecl &C) {
854     if (!checkKind(TF, tok::l_brace))
855         return false;
856
857     C.setLeftBrace(TF.next());
858     if (!parseScope(TF, C, true))
859         return false;
860
861     if (checkKind(TF, tok::r_brace))
862         C.setRightBrace(TF.next());
863
864     if (checkKind(TF, tok::semi))
865         C.setSemi(TF.next());
866     // else: just pass
867
868     return true;
869 }
870
871 static std::unique_ptr<Stmt> parseNamespaceDecl(TokenFilter &TF) {
872     if (!checkKind(TF, tok::kw_namespace))
873         return {};
874     auto Guard = TF.guard();

```

```

875
876     AnnotatedToken *NSTok = TF.next(), *NameTok = nullptr;
877     if (checkKind(TF, tok::identifier))
878         NameTok = TF.next();
879
880     if (!checkKind(TF, tok::l_brace))
881         return {};
882
883     auto NS = make_unique<NamespaceDecl>();
884     NS->setNamespace(NSTok);
885     NS->setName(NameTok);
886     NS->setLeftBrace(TF.next());
887
888     parseScope(TF, *NS);
889
890     if (checkKind(TF, tok::r_brace))
891         NS->setRightBrace(TF.next());
892
893     Guard.dismiss();
894     return std::move(NS);
895 }
896
897 static std::unique_ptr<ClassDecl> parseClassDecl(TokenFilter &TF) {
898     if (!(checkKind(TF, tok::kw_class) || checkKind(TF, tok::kw_struct) ||
899         checkKind(TF, tok::kw_union) || checkKind(TF, tok::kw_enum)))
900         return {};
901
902     auto Guard = TF.guard();
903
904     auto C = make_unique<ClassDecl>();
905     C->setClass(TF.next());
906
907     if (!(C->Name = parseType(TF)))
908         return {};
909
910     if (checkKind(TF, tok::colon)) {
911         C->setColon(TF.next());
912         bool Skip = true;
913         for (;;) {
914             AnnotatedToken *Accessibility = nullptr;
915             if (checkKind(TF, tok::kw_private) || checkKind(TF, tok::kw_protected) ||
916                 checkKind(TF, tok::kw_public))
917                 Accessibility = TF.next();
918             auto T = parseType(TF, false);
919             if (!T)
920                 break;

```

```

921         if (checkKind(TF, tok::l_brace)) {
922             C->addBaseClass(Accessibility, std::move(T), nullptr);
923             Skip = false;
924             break;
925         }
926         if (!checkKind(TF, tok::comma))
927             break;
928         C->addBaseClass(Accessibility, std::move(T), TF.next());
929     }
930     if (Skip) {
931         while (!checkKind(TF, tok::l_brace))
932             TF.next();
933     }
934 }
935
936 if (checkKind(TF, tok::semi))
937     C->setSemi(TF.next());
938 else
939     parseClassScope(TF, *C);
940
941 Guard.dismiss();
942 return C;
943 }
944
945 static std::unique_ptr<TemplateParameterType>
946 parseTemplateParameterType(TokenFilter &TF) {
947     if (!checkKind(TF, tok::kw_typename) || checkKind(TF, tok::kw_class))
948         return {};
949     auto Guard = TF.guard();
950
951     auto TPT = make_unique<TemplateParameterType>();
952     TPT->setKeyword(TF.next());
953     if (!checkKind(TF, tok::identifier))
954         return {};
955     TPT->setName(TF.next());
956
957     if (checkKind(TF, tok::equal)) {
958         TPT->setEqual(TF.next());
959         if (!(TPT->DefaultType = parseType(TF)))
960             return {};
961     }
962
963     Guard.dismiss();
964     return TPT;
965 }
966 static std::unique_ptr<TemplateDecl> parseTemplateDecl(TokenFilter &TF) {

```

```

967     if (!checkKind(TF, tok::kw_template))
968         return {};
969
970     auto Guard = TF.guard();
971     auto T = make_unique<TemplateDecl>();
972     T->setKeyword(TF.next());
973
974     if (!checkKind(TF, tok::less))
975         return {};
976     T->setLess(TF.next());
977
978     while (!checkKind(TF, tok::greater)) {
979         if (auto D = parseVarDecl(TF, /*TypeName=*/0, /*NameOptional=*/ false,
980                                 /*StopAtGreater=*/true))
981             T->addParam(std::move(D));
982         else if (auto TPT = parseTemplateParameterType(TF))
983             T->addParam(std::move(TPT));
984         else
985             return {};
986
987         if (checkKind(TF, tok::comma))
988             T->addComma(TF.next());
989         else if (!checkKind(TF, tok::greater))
990             return {};
991     }
992
993     assert(checkKind(TF, tok::greater));
994     T->setGreater(TF.next());
995
996     if (auto F = parseFunctionDecl(TF))
997         T->Templated = std::move(F);
998     else if (auto C = parseClassDecl(TF))
999         T->Templated = std::move(C);
1000     else
1001         return {};
1002
1003     Guard.dismiss();
1004     return T;
1005 }
1006
1007 static std::unique_ptr<Stmt> parseAny(TokenFilter &TF, bool SkipUnparsable,
1008                                     bool NameOptional) {
1009     if (auto S = parseDeclStmt(TF))
1010         return S;
1011     if (auto S = parseReturnStmt(TF))
1012         return S;

```

```

1013     if (auto S = parseLabelStmt(TF))
1014         return S;
1015     if (auto S = parseControlFlowStmt(TF))
1016         return S;
1017     if (auto S = parseTemplateDecl(TF))
1018         return std::move(S);
1019     if (auto S = parseFunctionDecl(TF, NameOptional)) {
1020         if (checkKind(TF, tok::semi))
1021             S->setSemi(TF.next());
1022         else if (checkKind(TF, tok::l_brace)) {
1023             S->Body = parseCompoundStmt(TF);
1024         }
1025         return std::move(S);
1026     }
1027     if (auto S = parseNamespaceDecl(TF))
1028         return S;
1029
1030     if (auto S = parseClassDecl(TF)) {
1031         if (checkKind(TF, tok::semi))
1032             S->setSemi(TF.next());
1033         else if (checkKind(TF, tok::l_brace)) {
1034             parseClassScope(TF, *S);
1035         }
1036         return std::move(S);
1037     }
1038     {
1039         auto Guard = TF.guard();
1040         if (auto E = parseExpr(TF)) {
1041             if (checkKind(TF, tok::semi)) {
1042                 Guard.dismiss();
1043                 return make_unique<ExprLineStmt>(std::move(E), TF.next());
1044             }
1045         }
1046     }
1047     return SkipUnparsable ? skipUnparsable(TF) : std::unique_ptr<Stmt>();
1048 }
1049
1050 TranslationUnit fuzzyparse(AnnotatedToken *first, AnnotatedToken *last) {
1051     TranslationUnit TU;
1052     {
1053         BasicTokenFilter<false> TF(first, last);
1054         while (!TF.eof()) {
1055             if (TF.peekKind() == tok::hash && TF.peek()->Tok().isAtStartOfLine())
1056                 TU.addPPDirective(parsePPDirective(TF));
1057             TF.next();
1058         }
1059     }

```

```
1059     }
1060     {
1061         TokenFilter TF(first, last);
1062         while (!TF.eof())
1063             TU.addStmt(parseAny(TF));
1064     }
1065     return TU;
1066 }
1067
1068 } // end namespace fuzzy
1069 } // end namespace clang
```