Clang pseudo-parser

hokein@google.com, sammccall@google.com - 2021-11-05 - **PUBLIC**

UPDATE: We implemented parts of this (mainly the GLR parser) but were unable to prioritize completing it, so it will shortly be removed from llvm-project.

TL;DR
We propose a clang pseudo-parser, which can approximately parse C++ (including broken code).
It parses a file in isolation, without needing headers, compile flags etc. Ambiguities are resolved heuristically, like clang-format. Its output is a clang::syntax tree, which maps the token sequence onto the C++ grammar.

Introduction
Scope
Design overview
Design: Grammar-based parser
- GLR parser
  - enhancement: lazy brackets
  - enhancement: support variant start symbols
- Earley parser

Design: Error resilience and recovery
- What does "accurate" mean for broken code?
- Structure: brackets
- Structure: sequences
- Recovery: top-down fallback parser
- Recovery: bottom-up fallback parser

Design: Combining parsers
Design: Parse forest
Design: Resolving ambiguity
- Likely and unlikely patterns
- Ambiguities in identifiers
- False grammatical ambiguities
- Using the Clang AST
- Mechanics of disambiguation

Design: Forming Syntax trees
- Local differences
- Grammar annotations

Design: Preprocessor
- Comments
- Attributes
- Directives
- Conditional compilation with #if, #ifdef etc

        - Syntax tree representation
        - Pathological cases
    - Macros
        - Placing macros directly in the grammar
        - Recognizing macro tokens
        - Parsing macro usages
        - Understanding macro signatures
        - Macros that expand to empty

- Work plan

- Related work and history
    - Syntax trees
    - Syntax trees from ASTs
    - LibFuzzy
    - clang-format
    - tree-sitter

Introduction

Many clang-based tools are built on top of clang AST. For example, refactoring tools traverse it to find the code pattern they need to change, and later do the desired refactoring transformations.

This AST is built by the clang parser, which has some fundamental limitations due to its accuracy:

- **slow:** clang must parse all the transitive headers and perform semantic analysis.
  (Often 10+ seconds, generating headers takes minutes)
- **dependencies**: integration with build systems is required (compiled flags, generated files)
- **single configuration**: code in not-taken `#if` branches is not analyzed
- **broken-code support**: clang is optimized to surface nice diagnostics and fixits, but representing broken code is best-effort. There are hard performance and complexity tradeoffs here.
- **stability:** clang's internal model of broken code is poorly-defined and crash-prone. This is largely down to the complexity of semantic analysis, and to broken code not being a primary design goal.
- **incomplete modeling of syntax**: for size reasons, the clang AST provides locations of critical tokens only (e.g. semicolons and `const` are often missing). Lengths are missing (`>` vs `>>` and others). In places, it models semantics rather than syntax (`int x, y;`). This makes mutation very awkward.

Addressing these requires a different parser design. Trying to modify clang to achieve both goals would be unsustainably invasive and complex.

These limitations create hard-to-solve pain points for tool users, e.g. tools are non-functional without an accurate `compile_commands.json`; a long warmup time before any features are available; multi-configuration support is impossible; broken code is not handled well; obscure bugs and crashes are a fact of life.
A notable exception is `clang-format`. It uses a special-purpose pseudo-parser that infers just enough structure to allow formatting.

We propose a general-purpose C++ pseudo-parser under clang/Tooling/Syntax. It trades off correctness for performance and other usability concerns. It aims to provide suitable foundations for fast, error-tolerant syntax-based analysis and refactoring.
This will be a library separate from the clang front-end, depending on the clang lexer only. Like the rest of clang/Tooling, it will not be used in the compiler.

We've spent much of this year exploring the ideas in this document, and have built functional prototypes of many parts of it: preprocessing, bracket recovery, GLR and Earley's parsers with lazy-bracket behavior, forests and disambiguation.

Scope

Goals

- **operate on a single source file**
  This is a simple boundary that lets us make useful performance and usability guarantees.
- **parse real world C++ code accurately**
  This is not a research project. A parser that fails on macros, common language extensions etc is much less useful.
- **parse code containing errors**
  We want to be able to use this directly in IDEs.
- **parse at interactive latency (10MB/sec)**

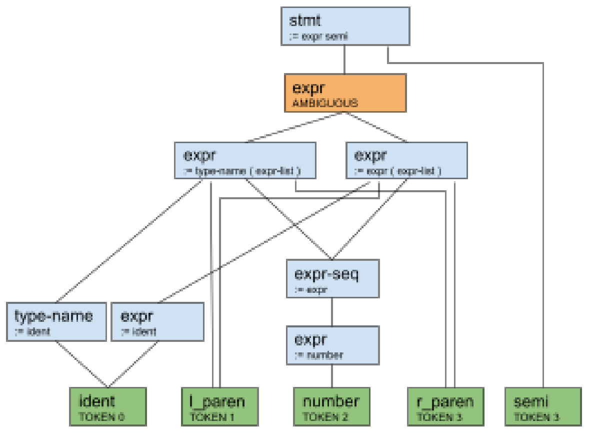
- **optionally consume clang AST, index etc to improve accuracy**

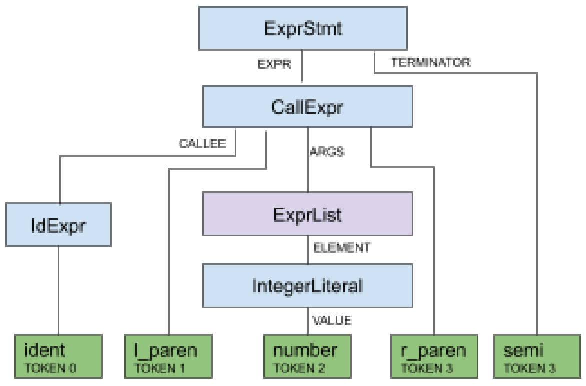
Non-goals
- **displace the clang parser**
- **resolve symbols**
  The parser can parse `x = 1;` as an assignment, but will not indicate what x refers to.
- **emit diagnostics and fixes**
  Semantic analysis is usually needed to understand intent. Without intent, diagnostics are poor.
- **ensure a correct parse in 100% of cases**
  The converse of "parse real world C++": pathological cases are less important if they're rare.
- **support non-Clang languages (C++ is the initial focus)**
  Generalizing increases the scope and complexity of the effort.
- **share code with clang-format (see "Related work")**
  clang-format is mature, specialized, and supports non-Clang languages.

Initially we will focus on building the foundation, and validate it with applications in clangd (see work plan below). We believe this project will enable many applications (e.g. smart-diff tools, fast linter replacing cpplint, syntactic search/replace tools) that are difficult or impossible to build with the current clang AST.

Design overview

The code is represented in different ways as the parser runs:

Source code The input string is a full source file exactly as it might exist on disk or in an editor buffer.	`a(2);`
Token stream This is an immutable flat list of tokens. The initial token stream is produced by the clang **lexer**. A simple **preprocessor** strips directives and comments, and picks a single path through #ifs, to produce a derived token stream for the initial parse. **Bracket pairs are marked** in the stream, and missing brackets are inserted before the parser runs.	a ident (l_paren 2 number) r_paren ; semi
Parse forest This is a DAG encoding how the code was parsed according to the language grammar. There is one sequence node per grammar rule matched. If multiple parses are possible, ambiguous nodes store the alternatives. Common subtrees are shared. (The representation is a DAG, but it's conventionally called "forest" because it captures all parse trees). The forest is built mostly with a **grammar-based parser** (GLR/Earley). The algorithm is extended to provide **error isolation** by understanding brackets and sequences. **Fallback parsers** are used to recover structure within	 stmt := expr semi expr AMBIGUOUS expr := type-name (expr-list) expr := expr (expr-list) expr-seq := expr type-name := ident expr := ident expr := number ident TOKEN 0 l_paren TOKEN 1 number TOKEN 2 r_paren TOKEN 3 semi TOKEN 3 (grammar simplified for example!)

broken code. Once the forest is built, it is **disambiguated** by eliminating impossible parses, and by heuristically scoring the alternatives for remaining ambiguous nodes.	
Syntax tree This is the output exposed by the parser API. It is easier to consume than the parse forest: ● No ambiguity ● Typed nodes with typed accessors ● Fewer intermediate nodes (flatter) ● Sequences are flat lists We **form the syntax tree from the parse tree** according to recipes associated with each grammar rule. Comments are reattached at this point.	 ExprStmt EXPR TERMINATOR CallExpr CALLEE ARGS IdExpr ExprList ELEMENT IntegerLiteral VALUE ident TOKEN 0 l_paren TOKEN 1 number TOKEN 2 r_paren TOKEN 3 semi TOKEN 3

Design: Grammar-based parser

C++ is a syntactically complicated language. While hand-written recursive-descent parsers are the standard approach for C++ compilers, this has several downsides:

- a huge surface area to build before real-world code can be parsed
- the "direct" control flow of a recursive-descent implementation is undercut by ambiguity. clang's parser has some tentative parsing but we'll have much more because we can't resolve identifiers with certainty up-front
- error-correction tends to be duplicated throughout the parser, so the cost of improving it is high
- the large gap between the specification and the implementation makes it hard to maintain

The C++ standard defines a formal C++ language grammar. Constructing a parser from the grammar gives us some hope of supporting the full language with reasonable effort, because basic support for "minor" language features is cheap to add.

The grammar does not fully specify C++. Formally, the grammar is context-free but ambiguous, while C++ relies on context to forbid certain interpretations and is thus unambiguous. This means we can match the code against the grammar, and then inspect/guess the context as needed to choose the best interpretation.

Deterministic generated parsers like LL, LR, LALR are efficient (linear time), but cannot handle ambiguous grammars, limit lookahead etc. We must use "general" parsing algorithms which handle arbitrary context-free grammars, and can produce multiple parse trees. We consider using GLR or Earley parser in the pseudo-parser (we're still evaluating the final decision).

GLR parser

In LR parsing, ambiguity within the lookahead window produces conflicts in the state machine. Generalized LR modifies the algorithm to allow LR state machines with conflicts. When a conflict occurs, it copies the parse stack and explores both options. (Breadth-first-search of all parses). A simple, weak parser

like LR(0) is used as we need not avoid conflicts.
The cleverness is in arranging data structures and sequencing work so that subproblems are still shared.
The parser produces the parse forest as a side-effect.

GLR achieves O(n^3) (n is the number of tokens) in the theoretical worst case (highly-ambiguous), practical performance is near-linear in most real-world cases.

Details of a standard GLR parser can be found here.

enhancement: lazy brackets

The GLR parser should be able to skip contents in brackets, and continue to parse the rest. (See error-recovery for details and motivations).

Implementation:

- When we construct a LR state from a kernel item “`A := { • body [lazy=brackets] }`”, we don’t expand the grammar rules of `body`. Instead, we record `(body, new state)` in that state's lazy-transitions list.
- During parsing, after shifting an open-bracket, we process this list.
  For each, we create an opaque node interpreting the bracket contents as e.g. `body`, and push a new state onto the stack as if we shifted `body` as a terminal.
- New states may be further ahead in the token stream vs others at the top of the (forked) stack. These are ignored until all other states have caught up.

More details can be found in this doc.

enhancement: support variant start symbols

An LR state machine has a designated start state, corresponding to “`start := • translation-unit”.`
We wish to be able to parse smaller chunks though, e.g. lazily-parsed function bodies, or expressions isolated after error-recovery.
We have several options:

- create separate states for `` `start := • expression` `` etc, and begin parsing at the appropriate one(s). (**Planned**)
- create a single state containing all start symbols. This means we look at all interpretations in parallel, and pick the one we want after. The main downside is the extra ambiguities introduced we must track.
- create a dedicated parser for each start symbol. This is wasteful if there is much grammar shared between them, which we expect to be the case (e.g. expressions and declarators appear often).

Earley parser

The Earley parser is another general CFG parsing technique. It moves left-to-right through the tokens, keeping track at each point of all the partially-applied grammar rules that may contribute to a parse.
After the input token stream has been processed, the chart tells us whether we succeeded, and we traverse the sequence of states to build a parse forest. More details of the algorithm can be found here.

Unlike the GLR parser which needs to build a LR automaton upfront, the Earley algorithm can be dynamic -- states are generated at runtime.

Operating directly on grammar rules and tokens rather than compiled states makes the parsing stage easy to customize:

- supporting variant start symbols is simple, just by adding the given start symbol to the initial state.
- lazy-brackets is similarly simple, a stack keeps track of states to reinject when brackets close.
- under "Structure: brackets" we mention a grammar-driven bracket-repair strategy that can be built on an Earley parser.

However, reconstructing the parse forest is more intricate, and needs to account for these customizations.

The Earley algorithm is O(n^3) for arbitrary grammars; In practice, it is much better O(n^2) for unambiguous grammar, O(n) for LR(k) grammar with some optimizations.
Unlike GLR, the constant factor depends on the size of the grammar: considering hundreds of states per token is typical, and this consumes a lot of memory for large inputs. It's possible to compress the itemsets using LR-like techniques.

Design: Error resilience and recovery

We should be able to accurately parse code with errors.

The most interesting errors are:

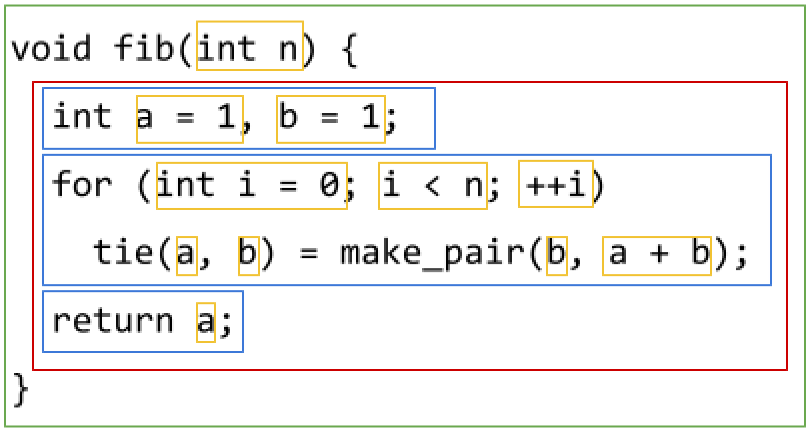
- code that is **incomplete** as the user hasn't finished editing yet.
  This tends to be some truncated construct, like `` `for (int x =) {` ``
  Motivation: IDEs see a lot of code in this state. And if batch tools like formatters/linters are fast and resilient to these errors, they can be used in IDE-like interactive workflows too (clang-format is an example).
- code that has **simple mistakes** and won't compile, but whose intent is fairly clear.
  This is often missing semicolons or brackets, typos, unresolved names etc.
  Motivation: failures due to errors that haven't been diagnosed yet is a poor experience. The pseudo-parser will not emit diagnostics and should be able to run faster than a diagnostic cycle.
- code that we **misinterpreted** due to limitations of the parser itself.
  For example, if the parser doesn't examine headers, it may not know about some macros used in the code.

As a rule of thumb, we'd like to handle ~2 incomplete constructs or other simple mistakes per line.

Incomplete code is particularly challenging to parse without context. An alternative would be to assume we have access to a recent version of the code that is fairly well-formed, and try to understand what changed. However such requirements make the parser much harder to deploy in practice.

What does "accurate" mean for broken code?

This is a slippery concept. Humans see hierarchical structure in code:



```
void fib(int n) {
  int a = 1, b = 1;
  for (int i = 0; i < n; ++i)
    tie(a, b) = make_pair(b, a + b);
  return a;
}
```

This structure is related to the language grammar, but is not purely the grammar. (For example, a function body is a flat list of statements, whether *statement-seq* is left- or right- recursive). It's much closer to the Syntax tree representation.

Accurately parsing broken code means something like:

> ***If you break code in one box, the parse of other boxes should not change.***

This is not *quite* right, it's easy to come up with counterexamples. The real definition is something like

> ***The parse should reflect the structure a human would see in the broken code.***
> ***When all interpretations break rules, the parser should ignore the things people ignore.***

We'll use both definitions to motivate and evaluate the design.

Structure: brackets

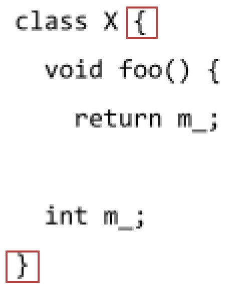
People understand code from the outside in, recognizing first functions, then loops, then statements, then expressions.
In C++, this mostly follows brace structure. The language is designed this way: we never need to look at what's inside the braces to parse the outer scope. (This is **sometimes** true for other brackets). This parser will read code in the same way.

Parsing of code in braces never affects the parsing of code outside them. This is also true for other brackets when the interpretation is forced, like `` `alignof( <expr> )` ``. These are known as **lazy** brackets since we may delay parsing their contents.
Rules are annotated in our grammar:

```
enum-specifier := enum-head { enum-body_opt [lazy=brackets] }
```

At a high level, when a parse head consumes **{**, it pins a parse state to be restored when it reaches **}**. Details will depend on the parsing algorithm.

Incorrectly matching brackets will cause a catastrophic parse failure (in humans too!)
We first detect and repair bracket problems by looking at indentation.



```
class X {
  void foo() {
    return m_;


  int m_;
}
```

Step 1 is to maximally match existing brackets, optimizing a cost function: the number of brackets left unmatched plus an indentation-sensitive penalty for each pair.
Step 2 is to choose an optimal insertion point for each unmatched bracket, or decide to delete it if all are bad. This uses indentation information and simple token-sequence rules.

Alternative: Repair brackets during parsing
Bracket repair can greatly benefit from understanding the grammatical structure of the surrounding code, only inserting brackets where they would help complete a grammar rule. Example:

```
  if (x[m] { … }
  call([m] { … }
```

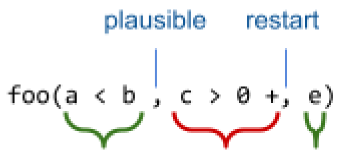
The token sequence is very similar; simple heuristics will not distinguish these cases.
If we let the parser consume an invisible) at any point, we can model different repairs as ambiguous parses.

This requires intrusive modifications to parsing algorithms. It's not clear the performance or code-complexity are acceptable, this needs prototyping. A design sketch is here.

If the outer scope is so broken we can't assess an inner scope, we make an informed guess.
This may mean parsing the scope a couple of ways and picking the best fit, or scanning for keywords near the opening bracket, etc.

Structure: sequences

Many bracketed scopes consist of sequences of repeated declarations, statements, expressions etc.
Errors in one sequence element should not affect the parsing of others.
To achieve this we need concepts of plausible boundary points, and restart points.



We set up the element parser at the start of the scope, and then reset the parser state after each element.
We accept the longest parse that ends at a plausible boundary point. (In the example, we reject the template id-expression `a<b,c>`).
If there's no successful parse then we consider this element broken and advance to the next restart point.

These boundary points are identified with heuristics:

- For delimited lists, the boundary is the next comma that's not in brackets. (With heuristics for <>).
- For sequences of statements and declarations, we must guess when } terminates a statement, infer missing semicolons etc.

The plausible boundary points affect correct code, so they must include all points where an element may end.
The restart points affect broken code only, so they can use aggressive heuristics (e.g. a newline without a change in indentation may indicate a new statement).

Recovery: top-down fallback parser

In broken code, the above techniques will often isolate a broken e.g. expression and its bounds.

We should identify the type of expression and its subexpressions (which may be valid) to make further progress. We don't know yet exactly what rules will work best here.

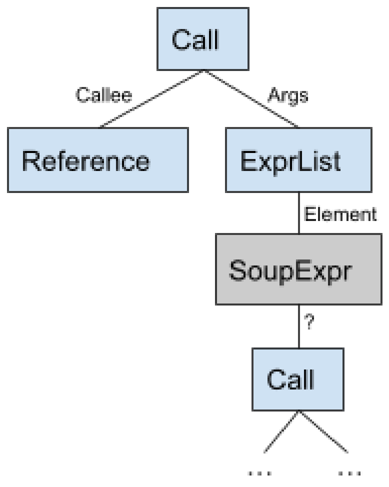
Some heuristics may use the partial results from the grammar-based parser:

- If we parsed a prefix of the code as an expression, accept that and discard the remaining tokens
  e.g. `a + b +`
- If the parser was still parsing at the end of tokens, we can force symbols to match the empty string.
  e.g. `a ? b :`
  In particular, this can recover statements that are missing semicolons.
- We may even attempt to run the parser forwards and backwards, and "sandwich" a broken symbol.
  e.g. `dynamic_cast<4+2>(e)`

Ad-hoc heuristics are also possible.
For example, an expression containing an unbracketed = is probably an assignment LHS = RHS, whether each side is well-formed or not, so we simply split, e.g: `a+ = b()`
This gains some of the flexibility of a hand-written parser. We only need rules for common cases that affect recovery, and they need not be strictly correct as it only triggers on broken code.

Recovery: bottom-up fallback parser

In the worst case, we cannot recognize the top-level structure of the broken expression using the grammar or heuristics. It may still be possible to recognize substrings of it. For example, within an expression `foo(...)` is probably a function call or functional-style cast.
We can recognize these by scanning for local token patterns, and represent them in the parse forest and eventually syntax tree as a special type of "token soup" node.



Even though the hierarchy is broken, representing local properties of the call can still be useful. (Consider semantic highlighting, or indexing references).

Design: Combining parsers

From the above descriptions, the lazy-parsing technique allows the parser to skip (or delay parsing) contents in brackets when parsing outer-scope code. And we have two major parsers: the grammar-based parser (for parsing variant symbols), the fallback parser (for parsing chunks of broken code). We need to combine them.

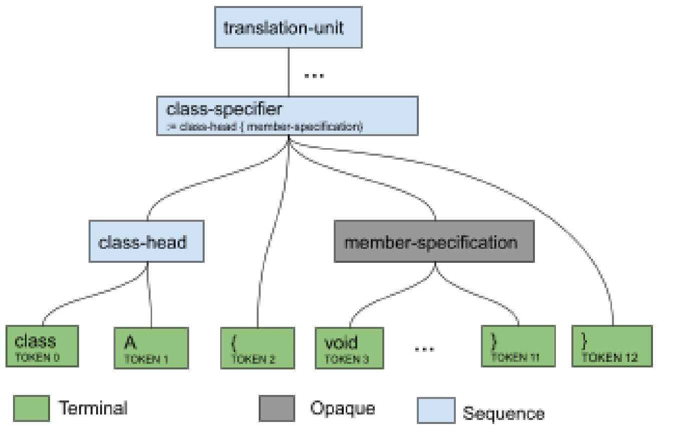
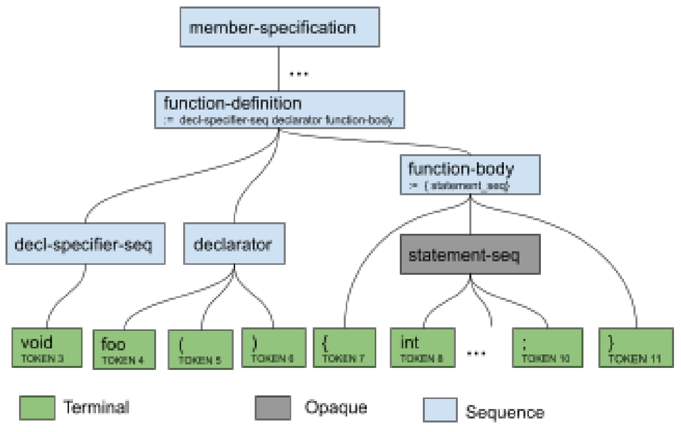
One implementation is to use a hierarchical parsing algorithm:

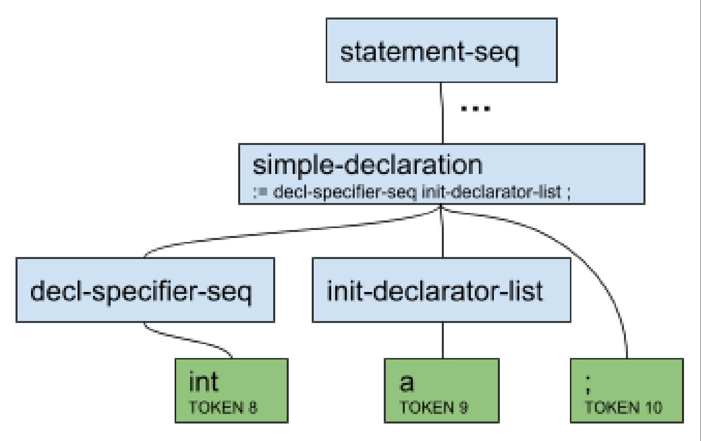
- we run a grammar-based parser at one time for one scope;
  ```
  Forest::Node* parse(TokenRange scope_range, grammar::Symbol start_symbol);
  ```
- skipped contents are modeled as an opaque node in the parse forest; The opaque node will be replaced by a concrete node when the inner-scope code is parsed;
- we parse outer-scope code first, then descend recursively into inner scopes;

Consider the following **correct** code:

```
class A {
   void foo() {
     int a;
   }
}
```

It will be parsed hierarchically from outermost scope to innermost scope:

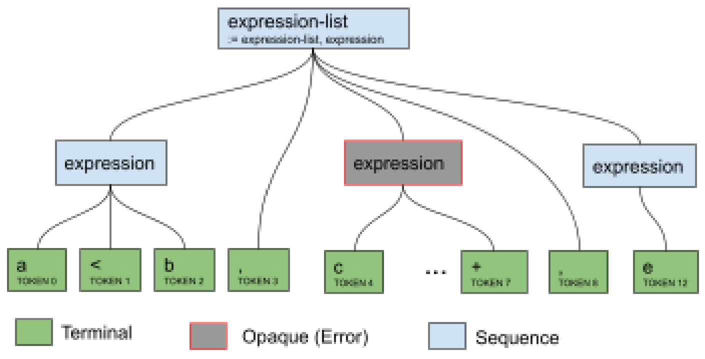
Step 1 -- parse the top-level TU-scope: class A { … } The resulting parse forest is shown as right. The parser skipped tokens in the class body, and emitted an opaque node. The opaque node presents the skipped tokens (tokens[3-11]), and has a grammar symbol member-specification, which guides how we should parse these tokens. After the forest is built, we traverse the forest and find all opaque nodes. For each opaque node, we parse their associated tokens independently.	 translation-unit … class-specifier := class-head { member-specification} class-head member-specification class TOKEN 0 A TOKEN 1 { TOKEN 2 void TOKEN 3 … } TOKEN 11 } TOKEN 12 Terminal Opaque Sequence
Step 2 -- parse the class-scope: void foo() { … } We will get a concrete member-specification forest node. And it will replace the opaque node generated in step1. In this node, there is an opaque node “statement-seq”, so we continue to parse its associated tokens (tokens[8-10])	 member-specification … function-definition := decl-specifier-seq declarator function-body function-body := { statement_seq} decl-specifier-seq declarator statement-seq void TOKEN 3 foo TOKEN 4 (TOKEN 5) TOKEN 6 { TOKEN 7 int TOKEN 8 … ; TOKEN 10 } TOKEN 11 Terminal Opaque Sequence

Step 3 -- parse the function-scope: int a; We end up with a simple-declaration node which has no opaque node inside, so the whole parsing is finished.	 statement-seq ... simple-declaration := decl-specifier-seq init-declarator-list ; decl-specifier-seq init-declarator-list int TOKEN 8 a TOKEN 9 ; TOKEN 10

In presence of **errors**, consider the example from the above “Structure: sequence” section:

```
foo(a < b, c > 0 +, e);
```

Our parser will hit an error when parsing the argument list, particularly at the second expression. When the lookahead reaches the “+” token, there is no valid action. The parser will apply heuristics mentioned above to determine the range of the broken code, move forward to the restart point, and continue parsing.
We end up with a parse tree:



The broken expression is modeled as an **opaque** node.
In the later step, we invoke a fallback parser to parse the “broken-expression” tokens. One strategy is to discard the extra trailing “+” token, and we get a valid expression `c > 0`.

An alternative implementation is to use a switch-back-and-forth strategy, e.g. when the main parser hits a start bracket, it switches a parser to parse the contents in the bracket, wait for it to finish and continue. We will get a complete forest in one run without paying a cost of traversing forest to find opaque nodes. But it is less intuitive and might be harder for debugging.

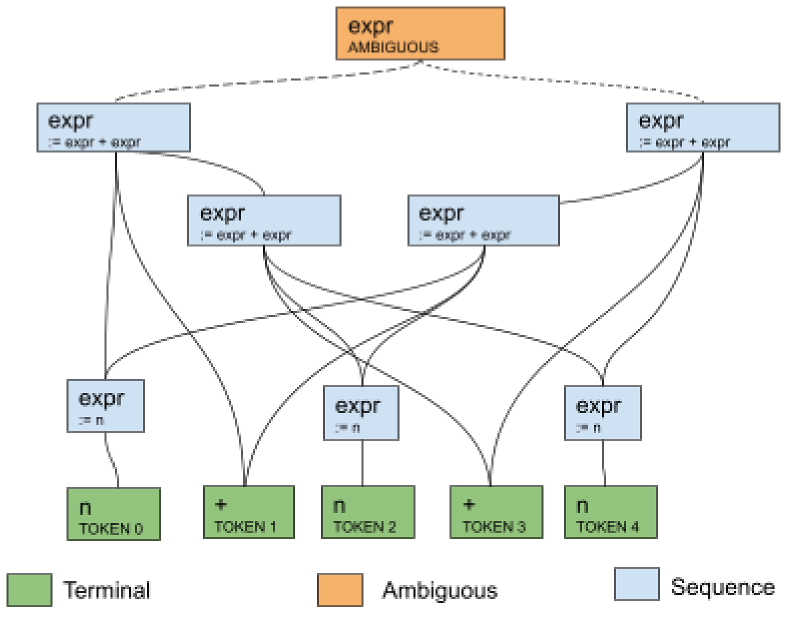
Design: Parse forest

Parse forest is the output of a grammar-based parser. It is a DAG which presents all possible parse trees without duplicating structures.
Common subtrees are shared: if several trees treat the token range [4, 10) as an expression then there is a single shared node representing that subparse.
This avoids exponential space requirements, and allows analysis (e.g. disambiguating internals of that expression) to be reused.

Consider a simple grammar for addition:

```
    expr := expr + expr
    expr := n
```

It is ambiguous because associativity is unspecified.
For the input n + n + n, there are two parse trees: (n+n)+n, n+(n+n). Here's the forest:



Each node binds a token range to a particular grammar symbol. There are four kinds:

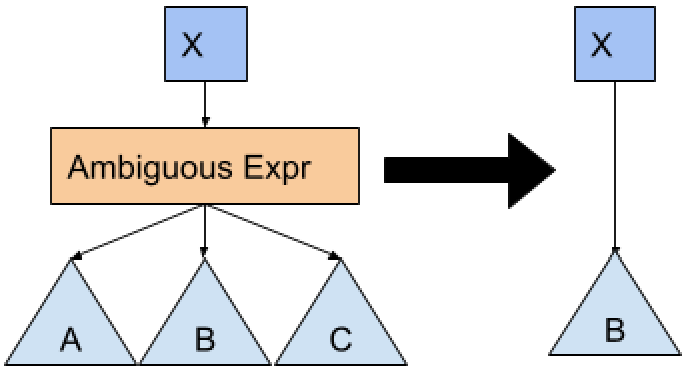
- **terminal**: one token in the token stream matches a terminal in the grammar.
- **sequence**: a non-terminal symbol is parsed from a grammar rule.
  If the rule is `A := X Y Z`, the node is for `A` and has 3 children `X, Y, Z`.
- **ambiguous**: the children are alternative parses of the same symbol
- **opaque**: a placeholder; we don't know *how* the tokens should be parsed.
  These are used for **lazy-parsing** (not parsed yet), or **error-recovery** (invalid).

Forest nodes are compact: 8 bytes per node, plus 8 per edge. They are allocated on an arena with edges as trailing objects. Based on an estimate of an average 10 nodes per token, 1.5 child per node, and 4 characters per token, the forest will be 50x bigger than the preprocessed source code, which is acceptable. (1MB is a large source file).

Probably we can live without mutability. Carefully ordering operations in the parsers lets gather all ambiguous alternatives as a batch.
After building, the main operations we will perform on the forests are scoring, disambiguation, and conversion to syntax trees. These are naturally expressed as bottom-up graph traversals, with disambiguation copying the selected pieces of the forest.

Design: Resolving ambiguity

The parse forest describes many interpretations, we need to pick one.
We do this by removing ambiguous nodes, and replacing them with a single alternative.



The chosen alternative should yield the best interpretation, so the B subtree should be the best interpretation of that range of tokens as an Expression.
(For correct code, the clang parser can provide ground truth for "best interpretation", and we can evaluate our algorithm in this way).

Likely and unlikely patterns

Some patterns are ambiguous with the information we're likely to have, and we must make a pure guess.

`aaa bbb(ccc);` // vexing parse: declaring a function or variable?
At function scope, this is probably a variable. At file scope, probably a function. We can assign scores to these alternatives. (If we know what `ccc` is we can disambiguate better, see below.)

`x<y, z>::w;`
This expression could be a pair of comparisons, but things that look like templates usually are.

`foo(m);`
This could be calling a function (likely), a constructor (less likely), or declaring `m` (very unlikely).
These will be encoded as ad-hoc rules, informed by gathering statistics on a large codebase.

Such common ambiguities can be identified by ad-hoc pattern-matching the children of ambiguous nodes.

Ambiguities in identifiers

Many ambiguities in the C++ grammar start with interpretation of identifiers, e.g:

```
namespace-name := identifier
class-name := identifier
template-name := identifier
id-expression := unqualified-id := identifier
```

In a typical parser, the "lexer hack" resolves this: the lexer can look up an identifier to classify it. (This uses the results of semantic analysis and so violates layering.)
But in the pseudo-parser, names are often defined in headers we don't see.

We have some tricks for resolving these ambiguities though:

- **multiple occurrences** of the same identifier often refer to the same thing
  We can use less-ambiguous occurrences to disambiguate others.
  Unlike the lexer hack, these hints can flow *forwards or backwards* in the code!

- identifiers often have **textual features** (case, underscores) that hint at their role
  We can infer these style rules from unambiguous parts of the file.
- it's often easy to build and look up **indexes** of identifiers
  Even for standalone tools, knowledge of the standard library helps.

For each occurrence of an identifier we combine possible interpretations to form a probability vector:
`Foo(42); (class-name=0.3, namespace-name=0, id-expression=0.7, …)`
Interpretation: what does the grammatical context indicate about this identifier?
And for each identifier, we combine occurrences and other signals (case, indexes, baseline distribution):
`Foo (class-name=0.1, namespace-name=0, id-expression=0.9, …)`
Interpretation: how should we interpret a random occurrence of `Foo` in this file, out of context?
Finally, we combine the contextual and global information to score an interpretation for the identifier.
This process lets us combine ambiguous information to make a decision without choosing an ambiguity to resolve "first", and encouraging but not forcing a consistent interpretation of each identifier.

False grammatical ambiguities

The formal grammar does not encode all rules of C++: it allows some parses that are semantically invalid.
For example: `` `static static int x;` `` is *grammatically* OK: the idea that only certain `decl-specifier`s can be combined is expressed in spec text, not the `decl-specifier-seq` grammar rules.

This is fairly harmless except where it causes ambiguity. Types may be repeated in a `decl-specifier-seq` for the same reason. The declaration `` `x y;` `` can parse as a declaration with two types, declaring no values!

We identify these bad interpretations with custom tree walks that e.g. count types to prove that parsing `` `x y` `` as a `decl-specifier-seq` cannot be valid.
Later we will attempt to eliminate such nodes when we resolve ambiguities.

Two cases this doesn't handle well:

- There's one grammatical parse, but it's semantically incorrect.
  We'll use it anyway, where ideally maybe we'd start error-recovery.
- There's too much ambiguity to prove that semantic rules are violated up-front.
  This would require carefully sequenced disambiguation to untangle.

Using the Clang AST

In some cases we may have a clang AST we can use for disambiguation. e.g.:

- latency-insensitive batch refactorings that don't need fast parsing.
  We can run the clang parser to improve accuracy, but still use the mutable syntax tree.
- stale state (like a cached AST in clangd, or a kythe index extracted from an AST)
  This is likely to be accurate for unmodified sections of the file.

Information about the boundaries and kinds of AST nodes in the clang AST can be translated into hints for the parse forest.

Mechanics of disambiguation

All the above techniques let us associate penalties/rewards with certain nodes in the forest.

We want to choose the tree that is as positive as possible.

To do this we simply walk the forest bottom-up, aggregating scores of nodes. When we reach an ambiguous node, we choose the child with the highest score.

If scores are equal, we can pick arbitrarily. In practice this probably means that no pattern heuristics matched at all, so we should log this in debug mode and attempt to add more.

Design: Forming Syntax trees

With ambiguities eliminated, the parse tree consists of sequence nodes and terminals.
We walk the tree bottom-up, constructing a syntax tree to describe each sequence node:

```
    Syntax::Tree* buildTree(grammar::RuleID rule, vector<Syntax::Tree*> RHS)
```

Local differences

There are several differences between the syntax tree representation and the grammar-based parse forest for the same code.

Syntax trees name the syntactic constructs and relationships.
Consider the rule

```
    postfix-expression := postfix-expression ( expression-list_opt )
```

A human can understand that this is a **call** expression, and that the expression-list is the **arguments**. The syntax tree exposes these as node *kinds* and edge *roles*.

Syntax trees omit nodes for uninteresting grammar rules.
An example here is trivial rules for the cascade of expression precedence.

```
    postfix-expression := primary-expression
```

If the RHS forms an IDExpression, then it also represents the `postfix-expression`.

Syntax trees have flat lists, rather than recursive ones.

```
    decl-specifier-seq := decl-specifier
    decl-specifier-seq := decl-specifier decl-specifier-seq
```

Grammar annotations

The simplest way to describe these conversions is to annotate the grammar.
Particularly as the conversion logic attaches to grammar rules, which have no names!

We can encode kinds/roles in the grammar with annotations on the symbols:

```
    postfix-expression [kind=CallExpr] :=
        postfix-expression [role=Callee]
        ( expression-list_opt [role=Args] )
```

Omitted nodes are a special case on the LHS:

```
    postfix-expression [kind=inline] := primary-expression
```

And lists are a special case on the RHS:

```
    decl-specifier-seq [kind=DSList] := decl-specifier [role=Element]
    decl-specifier-seq [kind=DSList] :=
        decl-specifier [role=Element] decl-specifier-seq [role=Rest]
```

Design: Preprocessor

C++ source code mixes two languages: core C++ and the preprocessor.
The main preprocessor features we have to understand are:

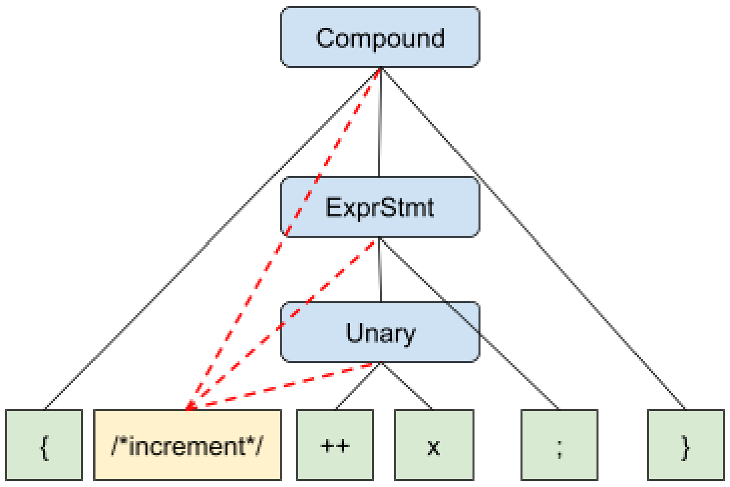
- #include and other directives
- conditional compilation with #if
- macro uses
- comments

We need to be able to handle each of these in the input, even though they're not part of the C++ grammar.
We need to represent them in the output syntax tree, without disrupting its structure too much.

The standard approach is to preprocess first, and parse C++ afterwards. This requires more information than we have (macro definitions), and the output describes only a single configuration with poor representation of preprocessor structure.
Nevertheless we try a modified version: we heuristically derive a (mostly) preprocessed token stream, parse it, and then try to layer the things we dropped back on top.

Comments

For parsing, comments are simply dropped from the derived stream.
Once the syntax tree is built, we attach them at the appropriate place, guided by their left/right neighbors.
There may be a choice of which node to attach comments to:



Applications care which comments document which entities, so we use heuristics.
We prefer to attach to the beginning of a node. We prefer certain node types that humans understand (e.g. Type) over others that are more abstract (DeclSpecList). We prefer to attach further up the tree.

Attributes

While not actually part of the preprocessor language, attributes have much in common with comments.

Attributes can be used almost anywhere in a C++ program, and attached to many constructs: types, functions, variables etc. Their contents are largely left unparsed.

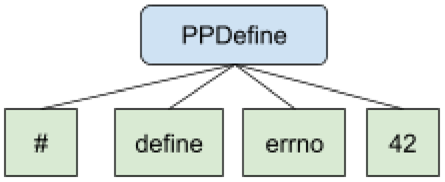
Attributes have a variety of different syntaxes (standard C++11/C2x attributes, GNU __attribute__, MSVC __declspec).
We could add these to the grammar, but:

- it adds a significant number of grammar rules to maintain
- it adds a significant branching factor and subgrammar (runtime size/complexity) to the grammar
- much real-world attribute use is non-standard, so we'd need to deviate (quite a bit) from the C++ standard grammar
- we'd rather **accept** code that puts attributes in the wrong place

Attributes can mostly be recognized lexically, and are always optional. We plan to treat them like comments: strip them from the token stream before parsing, and then reattach them afterwards.
The arguments of attributes will not be parsed, they're just tokens.

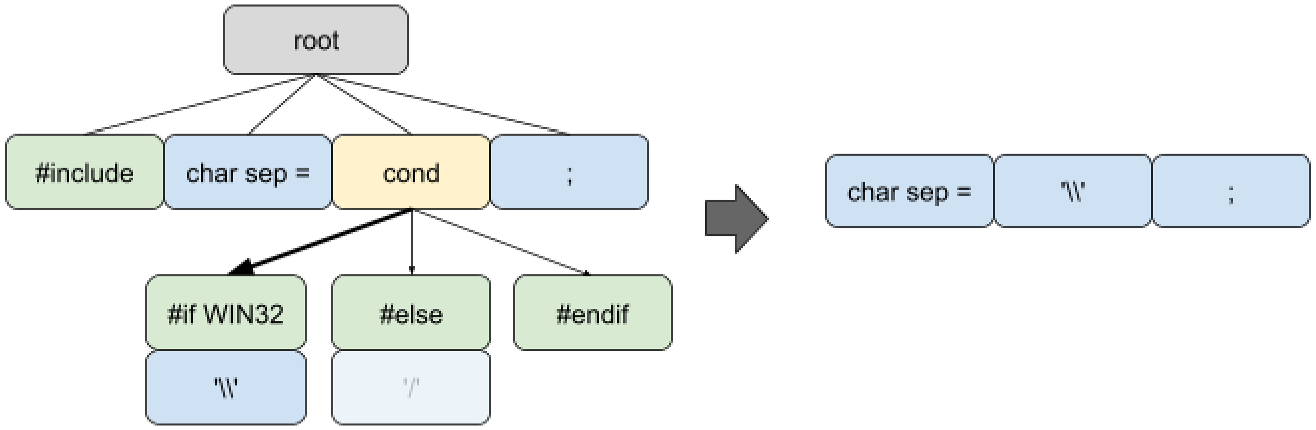
Directives

Most directives (#include, #define) etc are stripped and reattached like comments.
These are represented as nonterminals:

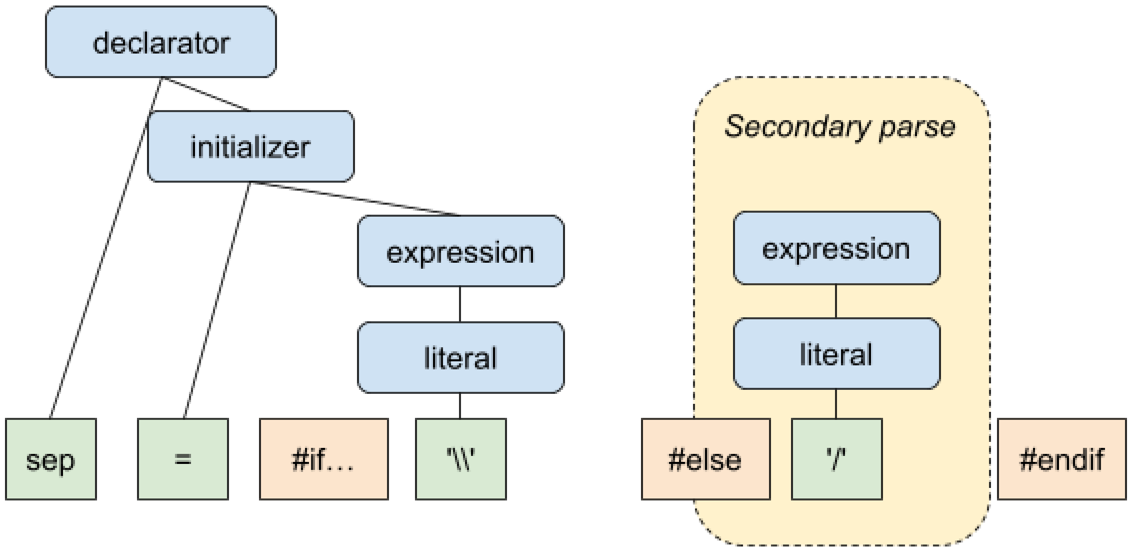


Conditional compilation with #if, #ifdef etc

Conditional directives (#if…#else…#end) are different. They break a region of code into alternatives, and we only get valid C++ by selecting one. The model is a tree:



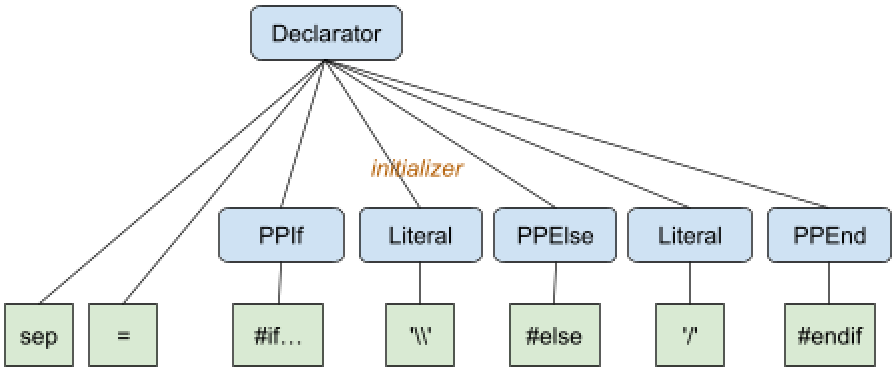
Initially, we select conditional chunks arbitrarily to produce a primary token stream, which we parse.



Later we find a chunk that was not selected (`'/'`). In the primary parse, its alternative (`'\\'`) was parsed as an `expression`, so we parse this chunk as an `expression` also. (This secondary parse cannot affect the interpretation of code outside the #if - only the primary parse gets to do that.)

Syntax tree representation

When the syntax tree is formed, these get stitched together and the directives attached:

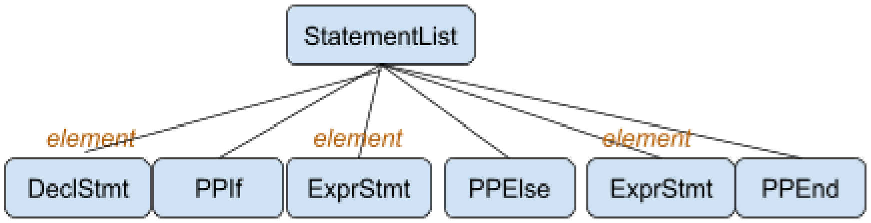


Here only the first literal gets to be the *initializer* of the declarator. Syntax trees require that only a single child of a sequence node like declarator has a particular role. The return type of Declarator::getInitializer() is Expression*, so we really have to pick one.

Alternative: represent conditional blocks as nodes
The case above would be neatly modeled by having a PPConditionalExpr node in the tree, which would be the declarator's initializer and contain all the PPIf… PPEnd as children.
However this can't easily model `#if` around elements in a list, and this is the common case.

Usually an `#if` section is in a list like StatementList, and all children can retain the *element* role.

```
void foo() {
  char* file = "temp.txt";
  #if WIN32
  DeleteFile(file);
  #else
  unlink(file);
  #endif
}
```



TBD: for applications that want to understand preprocessing structure, is the presence of directives in the syntax tree sufficient? Options include:

- exposing the directive tree itself
- providing an API to get the PP branch path of a token/node, compare paths etc

It's not clear how to keep these data structures up to date after syntax tree mutations.

Pathological cases

Things can go wrong if our assumptions fail.

The nesting structure of #if…#end can be malformed
For now we won't attempt any clever repair. #else and #endif outside #if will be ignored, and missing #endif will be inferred at the end of the file.

The conditional section may not cover a grammar rule cleanly

```
remove(
#if WIN32
  "c:\temp");
#else
  "/tmp");
#endif
```

In this case we'll end up unable to perform a secondary parse, and the tokens will just be "token soup" in the parent scope.

The primary path through the conditional blocks may produce unparseable garbage
Similar to the previous case, we'll be unable to perform a secondary parse.

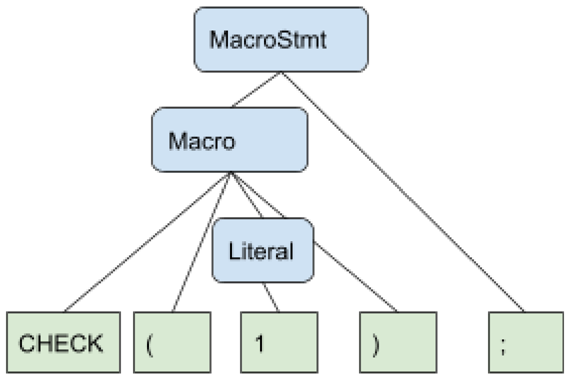
Macros

Unlike directives and comments, usages of macros can't be completely recognized and handled before parsing. This is because:

- we don't always know which identifiers refer to macros and whether they have arguments
- we don't always know what they expand to
- if we expand macros and parse the result, it's hard to relate the parse tree to the source code

Instead, we attempt to handle macros in the parser. Although any identifier could potentially be a macro, and a macro could potentially appear in any context and expand to anything, most uses follow common patterns we can infer.

Our goal is to produce syntax tree nodes that model the macro expansions themselves with as rich a structure as possible:



Placing macros directly in the grammar

The simplest possible approach is to treat hardcoded macros as if they were keywords, and refer to them in grammar rules. e.g.

```
decl := TEST (identifier , identifier ) compound-stmt
```

This obviously doesn't scale well but will enable us to parse real-world code in early stages of the project, before a more general handling of macros. **It's likely we won't get past this stage for some time** (see work plan).
It also allows us to handle macros that have a strange grammatical structure, like gtest's TEST.

Recognizing macro tokens

Precisely because macros are hard to recognize, using `ALL_CAPS_NAMES` is a strong convention. We consider any such name a likely macro. (The weakness of this heuristic is a single `WORD`).

`assert()` is an example of a macro that defies this convention. We need not recognize it as a macro at all, it resembles a function and we can parse it that way. This is not a coincidence.

We can also scan #defines from the current file, consult an index, or a clang AST/Preprocessor if available. This is probably not important, the naming convention is likely sufficient.

Parsing macro usages

Armed with logic to recognize which tokens may be macros, we can add grammar rules:

```
macro := identifier [guard=LikelyMacro]
macro := identifier [guard=LikelyMacro] ( macro-arg-list_opt )
macro-arg-list := macro-arg
macro-arg-list := macro-arg , macro-arg-list

expr := macro
stmt := macro ;
…
macro-arg := expression
macro-arg := identifier
…
```

This is likely to generate many ambiguous interpretations each time we see a macro, which reflects reality! Choosing one requires us to understand the macro's structure.

Understanding macro signatures

The signature of a macro is:

- whether a macro has arguments (function-like vs object-like)
- what grammatical symbol each argument is (e.g. expression)
- what grammatical symbol the macro expands to

Example:

```
offsetof(expression, identifier) → expression
```

In principle macros arguments or results don't need to be whole (or consistent) grammatical constructs. In practice, they usually are, or can be interpreted that way.

If we can obtain a macro's signature, we can use this in disambiguation. (**Alternative**: Using the macro signature during parsing would be more efficient, though complex).

Possible sources for macro signatures are:

- a configured or hardcoded list (e.g. `EXPECT_EQ` or `_Atomic`)
- #defines from current file (partial info, e.g. number of arguments)
- correlating multiple uses of the same macro
- an index which correlates #defines and usages seen in multiple files
- an index built from examining clang ASTs, which contains uses of the macros

Macros that expand to empty

Some macros expand to nothing, or are best treated that way.
For example, macros like `ALWAYS_INLINE` that conditionally expand to attributes.

These need special support in the parser to allow skipping over them. This is simple to implement in a parser that handles ambiguity; it requires looking ahead 1-2 tokens.

If such a parse "wins", it is represented in the syntax tree by a NullMacro node.

Work plan

Milestones 1-5 are essential to this being generally usable at all.
Milestone 6 is key to accuracy. Milestones 7 and 8 are nice-to-have.

	Milestone	**application enabled (clangd)**
M1	Project design doc & RFC accepted in LLVM	
M2	Token stream Bracket matching Simple preprocessor (strip directives, heuristically pick one #if branch)	LSP Folding ranges improve clang bracket recovery
M3	Parse correct (preprocessed) code → ambiguous forest Dummy disambiguation (pick first alternative)	
M4	Baseline error isolation: - brackets - lazy parsing - sequences Placeholder for fallback parsing	
M5	Forming syntax trees	file outline without AST fast indexing
M6	Disambiguation based on index/AST/heuristics	large accuracy boost more features available before first parse

M7	Advanced error recovery: - fallback parsing - parser-based bracket repair	improve broken-code handling
M8	Parsing macro uses Including #if branches Comments, attributes	

Related work and history

There are 3rd-party pseudo parsers (e.g. tree-sitter) which share similar goals. An alternative is to use them in LLVM. Our community is conservative in introducing and using external dependencies in LLVM unless there is a strong reason. Using 3rd-party code in LLVM might cause licensing problems; 3rd-party is less well integrated with LLVM (code style, ADT data-structure, clang-syntax tree); customization is needed to fit our use cases (high maintenance cost). Therefore, we decided to write our own parser.

Syntax trees

Syntax trees was an attempt to define a regular syntax-oriented representation of source code that could support such edits. It has not been completed, but we expect to complete the library as part of this work.

This idea came out of the need to produce source edits based on clang ASTs.
Since the dawn of man, tool authors have been refactoring code by inspecting the AST and trying to make logical changes by pasting strings together. (see clang::tooling::Replacement). Our own experience is with clang-tidy and clangd.

Syntax trees from ASTs

The original plan was to translate the clang AST into Syntax Trees which would then be mutable.
This worked somewhat: by combining the relationships between AST nodes, the token locations stored in them, and the token sequence, we could infer the syntactic constructs and their ranges. (It did involve a bit of ad-hoc parsing, hunting for the locations of const qualifiers and semicolons that the AST merely implied).

However this approach failed on broken code, which is important to clangd. Big chunks of code were not represented in the AST at all because the **semantic** invariants of their nodes were not satisfied. Clang would infer missing punctuation and keep going, but this info is not stored in the AST. Since the AST didn't fully correspond to the token sequence, we couldn't use it directly to interpret the tokens.

The next idea was to parse the code using the ASTs as a rough guide only. (Is this word a class or a variable?). But if we were going to build a whole parser, it makes sense for this guide to be optional. Clangd has a long warmup time (building an AST) when it'd be great to offer partial functionality. clang-format shows that many C++ ambiguities can be resolved heuristically in practical code, and that speed is a feature. So this led to the current project.

LibFuzzy

LibFuzzy is a C++ pseudo-parser project from 2014.
It shares many of the same high-level goals: single file parse, detail down to the token level.
There was an RFC on adding this to the clang tree, and sentiment that this would be useful.
However the review stalled, and the implementation is very incomplete.

It is written in recursive-descent style, and so far covers only the most common nodes and has limited error-recovery. Our sense is that this is a good design for the 80% point. The cost of completing and maintaining a second hand-written parser that covers the long tail of C++ syntax and error-recovery would be high, and comparable to the clang parser itself. For our purposes we'd like to have a parser that can represent all real-world code, and a design that scales better to that.

clang-format

In order to format code, clang-format needs to understand much of its structure, both macro (for mandatory line breaks and indentation) and micro (for optimal line-wrapping and spacing within lines).
Therefore clang-format does have a form of pseudo-parser, built atop the clang Lexer. For us, this was the strongest evidence that single-file parsing works in real-world workflows.

However this parser is itself not suitable for our purpose:

- it is special-purpose, obtaining only information that influences formatting. (Roughly: block/line structure, plus role annotations on individual tokens)
- its implementation is coupled to clang-format internals in a way that is hard to extract. (And risky in a mature product).
- it has support for various non-clang languages (Java, C#, Javascript, Protobuf) that we don't wish to build a general-purpose AST for but cannot drop from clang-format.

The last point is particularly painful, because it suggests that even if successful this pseudo-parser will never replace that of clang-format.

tree-sitter

Tree-sitter is a GLR-based parsing library for general programming languages. It is developed by GitHub to improve features (code-folding, outline, syntax-highlighting) in Atom editor, power code navigation in GitHub website etc.

Tree-sitter shares most of the high-level goals: parse a single file, robust on error code. It supports C++, C and other programming languages. We investigated it before starting writing our own parser, it is a nice framework, it gives us strong evidence that GLR can parse real-world C++ well. But at the same time, it has some weak points:

- its C++ grammar is built on top of the C grammar, not following the one from C++ specification. While it works for 95% of real-world code, it is not general and requires ad-hoc patching on the grammar file to fix missing cases;
- Ambiguities are simply resolved by (dynamic/static) precedence which is hard-coded in the grammar. While it is a good and simple model for general purposes, it doesn’t suit our purpose. We'd like to parse the code as precisely as possible, therefore we need a more powerful mechanism which can allow us to implement/inject different strategies during the ambiguity resolution;

- limited support of preprocessor elements: only top-level directives are supported by adding preprocessor-specific grammar rules;
- the generated parser is C code, and requires some runtime code to use, checking a bunch of third-party code in LLVM repo is not a common practice in LLVM;