LLVM 和 GCC 的语法分析算法

Joseph Qiu

2023 年 1 月 5 日

目录

1 序言

本文是一篇简单的调研综述型论文,通过源码和手册阅读,分析 LLVM/GCC 两个现代编译器框架使用的递归下降语法分析算法，并与其它方法进行比较。

1.1 语法分析理论简述

在编译器架构中，语法分析器接收词法分析器提供的记号串，根据文法规则将它们组合为表达式、声明、函数体等带有语法逻辑的结构，构建语法树。同时，应进行错误检查等工作。

常见的语法分析方法分为通用方法、自顶向下、自底向上三种。

1. 通用方法，如 Cocke-Younger-Kasami 算法和 Earley 算法等，可以分析任意文法，但构建的语法分析器效率较低。

2. 自顶向下方法，分为以下几类:

   - 一般递归下降分析，可能需要回溯。
   - 对 LL(k) 文法，可以使用预测分析器（即 LL 分析器)。
   - 若用显式状态栈取代隐式递归调用，可以实现非递归的 LL 分析器。

3. 自底向上方法，分为以下几类:

   - 一般移进规约分析法。
   - 对 LR(k) 文法，可以使用 LR(k) 分析法。具体可以使用 SLR、LALR 等方法。LR 语法分析器可以使用自动工具生成。

1.2 C 可以被哪些语法分析器分析

在探讨 LLVM 与 GCC 使用的语法分析算法之前，我们应当讨论一下有哪些选择是可行的。也就是说，C 语言可以使用哪些语法分析方法进行分析?

C 语言存在如下二义性:

```
Foo * Bar;
```

该语句可以被分析为：

- 对 `Bar` 的声明，其为指向 `Foo` 类型的指针。
- `Foo` 与 `Bar` 相乘。

因为这一类型/变量名二义性，C 语言无法被纯粹的 LL 或 LR 分析法分析，包括使用 `Yacc` 等工具生成的分析器。[2] 大多使用 LR 分析法的 C/C++ 语法分析器会加入针对特定规则的语义检查来消除歧义。针对上述的 `Foo * Bar` 歧义，许多 C/C++ 分析器在语法分析过程中同步生成符号表，从而当分析到 `Foo` 时，分析器便知道它是否是已经声明的类型。[9] 这类修改后的 LR 分析器可以正确解析 C/C++。

通用的递归下降分析当然可以被用于 C 语言分析。由于可能存在的回溯，递归下降分析的时间复杂度为 $O(k^n)$ （n 为输入串长度）。许多递归下降分析器通过引入预测集和记号预读避免回溯，可以在接近线性时间内完成解析，实际上已经接近 LL 分析器。[1]

此外，目前也有许多团队使用 GLR 等更强的 LR 分析器对 C 进行解析，而且无需进行对自动生成的分析器进行额外修改。GLR 分析器针对可能的歧义产生一个树形表示，并在语法分析完成后消除歧义。[3] 代表性工作有 Elsa 等。

1.3 本文分析对象

本文将分析现代两大主流编译框架 LLVM 和 GCC 的语法分析器。

对于 LLVM，我们分析作为 LLVM 基础设施前端的 Clang。LLVM 的模块化设计将所有功能分类组织为库，可运行部分则通过调用各类库函数完成工作。Clang 作为 LLVM 中较高层次的工具，引用多个前端相关的库，其中较为重要的包括进行词法分析的 `clangLex`，构建 AST 的 `clangAST`，进行语法分析的 `clangParse`，进行语义分析的 `clangSema`，完成 IR 代码生成的 `clangCodeGen` 等。

[1]这里存在一个问题：完全避免回溯的递归下降分析器是否就是 LL 分析器？但如果是，则意味着 LL 分析器分析了一个非 LL 文法。

从整体架构来看，我们可以将这些库分别抽象为一些功能单元，包括预处理器和词法分析器 Preprocessor/Lexer，语法分析器 Parser 和语义分析器 Sema，它们协同工作，将原始代码识别为记号流，识别语法结构生成 AST，最后产生 LLVM IR。

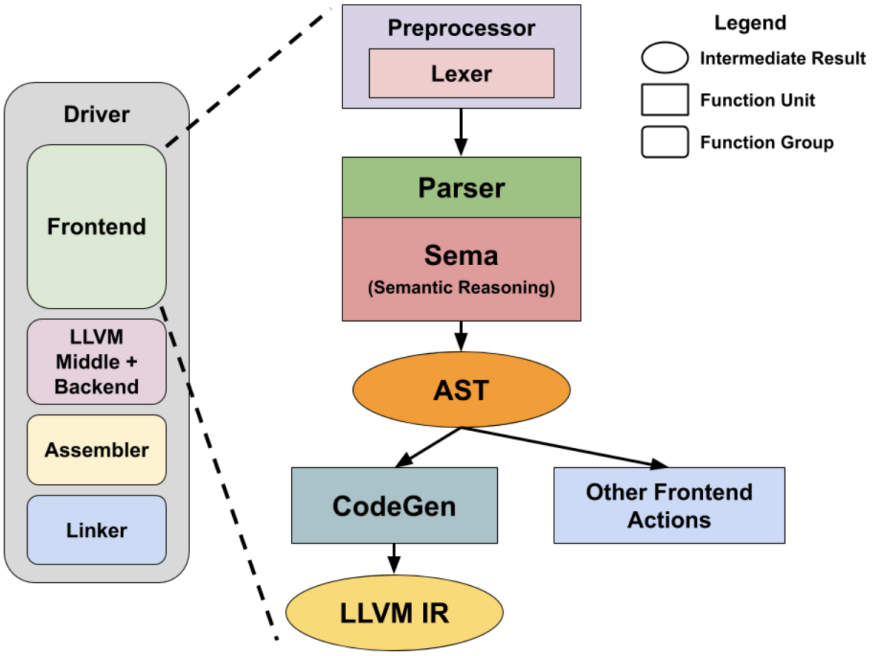

图 1: LLVM 前端 Clang 架构 [5]

本文分析 Clang 的 `clangParse` 库，即语法分析器 Parser 功能单元。进行阅读的源码为 LLVM 12.0，该库目录为 `clang/lib/Parse`。由于 Clang 的 Parser 同时兼容 C 与 C++ 解析，出于简洁性考虑，本文分析将侧重其对 C99 语法的解析过程。

对于 GCC，我们分析其 C 前端的语法分析部分，该过程所有可能行为都在 `gcc/c/c-parser.cc` 单个文件中定义。进行阅读的源码为 GCC 12.2。同样，本文分析将侧重其对 C99 语法的解析。

2 LLVM 使用的语法分析算法

根据 LLVM 文档说明，Clang 使用手写的递归下降语法分析器，这便于开发者理解代码，便利地加入临时规则（ad-hoc rules）和其他技巧。

递归下降语法分析法从根节点开始，从上到下、从左至右地为输入符号串建立一棵分析树。该算法表现为一个递归调用过程：对当前处理的节点，选择一个产生式展开。对展开部分，若节点为非终结符，对其进行递归调用的展开过程；若为终结符，则进行匹配。匹配失败，需要进行回溯。

正如 1.2 中所说，通用的递归下降算法具有指数时间复杂度，完全按照原始算法实现显然不可取，许多递归下降分析器也都使用了记号预读等方法避免回溯。下面我们通过 Clang Parser 的具体分析流程解释它对递归下降算法的具体实现。

2.1 分析部分与样例说明

我们分析 Clang 语法分析的起始部分，说明如下：

根据 C99 标准的文法规定，起始符号为 `translation-unit`，其两个产生式为：

```
translation-unit: [C99 6.9]
    external-declaration
    translation-unit external-declaration
```

外部声明 `external-declaration` 产生函数定义 `function-definition` 或声明 `declaration`：

```
external-declaration: [C99 6.9]
    function-definition
    declaration
```

这两者的产生式为：

```
function-definition: [C99 6.9.1]
    decl-specs declarator declaration-list[opt] compound-statement

function-definition: [C99 6.7.1]
    decl-specs[opt] declarator declaration-list[opt] compound-statement
```

```
declaration: [C99 6.7]
    declaration-specifiers init-declarator-list[opt] ';'
```

其中包括声明区分符 declaration-specifiers、声明符 declarator。用两个简单的例子说明一下这些概念：

```
static const int Var;
~~~~~~~~~~~~ ~~ ';'
 (1)        (2) (3)

(1) 声明区分符 declaration-specifiers，包括 qualifiers 和 specifiers
(2) 声明符 declarator，这里经过 declarator -> direct-declarator ->
    identifier 派生出一个标识符 identifier (Var)
(3) semicolon
```

```
int func(int x, int y);
~~~ ~~~~ ~~~~~~~~~~~~';'
(1) (2)      (3)     (4)

(1) 声明区分符 declaration-specifiers
(2) 声明符 declarator，同样派生出 identifier
(3) (identifier-list)
(4) semicolon
```

2.2 递归下降的基本方法

根据递归下降算法，主要需要实现的两个动作为“派生”和“匹配”。

- 派生：根据当前非终结符，选择一条产生式进入相应分析路径，完成对应分析动作。最一般的实现方法即为对每条产生式设置一个分析函数。
- 匹配：若分析过程产生终结符，与当前输入串进行匹配，若一致，则通过读取下一记号，“吃掉”（eat）当前记号。

2.3 分析外部声明

在语法分析过程开始后，起始符号 translation-unit 的展开就存在左递归。显然，此处如果递归调用当前函数，本身就显得极不合理，会导致无限循环。我们也无需修改当前文法，先消除左递归，再针对新文法设计分析函数。translation-unit 的产生式意味着一个翻译单元可以派生出多个外部声明（C 规定至少一个），因此 Clang Parser 的程序入口并非对 translation-unit 的分析函数，而是 ParseFirstTopLevelDecl。Parser 一开始工作，便直接进入分析第一个顶层声明。这里的顶层声明 top-level-declaration 是 Clang 为支持 C++ 语法添加的，在分析 C 语言时，会跳过其余部分，进入 ParseExternalDeclaration 函数。

在分析外部声明 external-declaration 时，Clang Parser 并没有选择对两个产生式分别设计解析函数，因为能够区分两者的向前看字符数量是未知的，我们暂时无法确定当前解析的是函数定义还是声明。因此，Clang Parser 将两个产生式共同解析，也即先解析其产生式中的公共部分，包括声明区分符 declaration-specifiers 和声明符 declarator。实际上，这等价于一个对文法提取左因子的过程。

使用的函数为 ParseDeclOrFunctionDefInternal：

```
Parser::ParseDeclOrFunctionDefInternal(ParsedAttributesWithRange
    &attrs,
                                ParsingDeclSpec &DS,
                                AccessSpecifier AS) {
  // Parse the common declaration-specifiers piece.
  ParseDeclarationSpecifiers(DS, ParsedTemplateInfo(), AS,
                      DeclSpecContext::DSC_top_level);

    ...

  return ParseDeclGroup(DS, DeclaratorContext::File);
}
```

如上所述，该函数针对声明区分符和声明符，分别使用 ParseDeclarationSpecifiers 函数和 ParseDeclGroup 函数进行解析。在 ParseDeclGroup 解析完第一个声明符 declarator 后，我们可以判断当前外部声明是函数声明还是变量声明，从

而选择后续的解析方法。在这里，Clang Parser 通过利用分析过程中收集的上下文信息，给出确定的产生式选择，避免回溯。

2.4 分析直接声明符

在上述过程中，`ParseDeclGroup` 递归调用 `ParseDirectDeclarator` 解析直接声明符（非指针声明符）。`direct-declarator` 的产生式同样存在左递归，其目的是为了表述括号标识符 `ParenDeclarator`。

```
direct-declarator:
    identifier
    ( declarator )
    direct-declarator [ type-qualifier-listopt ]
    direct-declarator [ static type-qualifier-listopt
        assignment-expression ]
    direct-declarator [ type-qualifier-list static
        assignment-expression ]
    direct-declarator [ type-qualifier-listopt * ]
    direct-declarator ( parameter-type-list )
    direct-declarator ( identifier-listopt )
```

类似于此前提及的提取左因子方法，`ParseDirectDeclarator` 首先处理后续输入串开头的标识符 `identifier` 或 `'('`。若为标识符 `identifier`，设置该声明符 `D` 的标识符和位置信息。若为 `'('`，调用 `ParseParenDeclarator` 进行括号标识符解析。

无论该声明中是否使用了括号标识符（即存在左递归情形），所有情况最终都会完成对一个标识符 `identifier` 的解析。解析完成后，使用 `goto` 语句前往 `PastIdentifier` 标签，根据后续记号是 `'['` 或 `'('` 进行相应选择继续解析即可。

2.5 小结

可以看到，Clang Parser 的语法分析算法并非严格形式化的，与其说它遵循递归下降算法范式，不如说它更接近人类手动解析行为：自左向右扫描输入串，通过递归调用非终结符对应的分析函数进行展开；通过记号预

读，结合分析过程中收集的上下文信息，在大多数情况下可以给出确定的产生式选择；若无法确定，一些情形下可以先继续解析产生式的公共部分（左因子），后续再进行选择。若文法产生式存在左递归情形，消除左递归也无需通过复杂的形式化方法改写文法。由于左递归的使用常常只是为了便于表示一些特殊语法结构（如括号标识符），在设计语法分析器时只需针对这些特殊情形作相应考虑即可。

2.6 处理类型/变量名二义性

最后我们简单看一下 Clang 是如何处理类型/变量名二义性的。

Clang 的词法分析器 Lexer 并不区分用户定义类型和其它标识符。在它输出的记号流中，用户定义类型也被标记为标识符 `identifier`。Clang 通过维护符号表实现用户定义类型的识别。[1][2]

在 Parser 分析完 `typedef` 定义部分时，相应的符号表注册工作就开始进行。通过 Parser 的 `Actions` 接口，它调用 `Actions.ActOnDeclarator`，由 Sema 将该定义加入符号表。

在 Parser 调用到 `ParseDeclarationSpecifiers` 分析声明区分符时，它需要分辨一个标识符是否是上文定义的类型。，它调用 `Actions.getTypeName` 向 Sema 查询该标识符信息。`Sema::getTypeName` 调用 `LookupName` 在符号表中完成查找。

3 GCC 使用的语法分析算法

GCC 在旧版本中使用 Yacc 语法分析器，而在 3.x 版本的一次更新中替换为手写递归下降分析器。由于其算法和实现与 Clang Parser 基本一致，这里仅作简要分析。

在分析过程中，GCC 定义了唯一一个 `c_parser` 结构体用于保存语法分析状态、上下文信息、预读词法符号等。程序入口为 `c_parse_file` 函数，其中

[2]虽然概念上 Parser 和 Sema 是紧密耦合的，Clang 设计的 Actions 界面将两者在代码层面进行了清晰的分离。Parser 负责驱动语法分析，Sema 则处理语义信息。在这里符号表属于语义信息，因此由 Sema 负责。

处理可能的 PCH 预处理制导，初始化 `parser`，然后进入 `c_parser_translation_unit` 进行顶层翻译单元的解析。

GCC Parser 和 LLVM Parser 使用相同方法处理简单左递归情形：该左递归用于表示 `translation-unit` 可能由多个外部声明组成，只需循环解析外部声明即可。GCC Parser 循环调用 `c_parser_external_declaration`，直到遇到 EOF。

解析外部声明时同样面临产生式选择问题，GCC Parser 也使用相同方法，为函数定义与声明设计统一的解析函数 `c_parser_declaration_or_fndef`，并在解析完声明区分符 `specifiers` 与第一个声明符 `declarator` 后确定当前解析的是函数定义还是声明。同样，先解析两个产生式的公共部分（声明区别符和声明符）。解析完声明符后，当下一记号为赋值符，`comma` 或 `semicolon` 时，可以认为该部分不是函数定义，接下来只需解析剩余声明部分即可。

可以发现，GCC Parser 使用的递归下降分析基本思路与 Clang 一致，完整流程由递归的函数调用组成，解析函数的选择也通过有限预读避免不确定情形。

4 使用递归下降分析的原因

虽然编译理论为编译器开发者提供了多种语法分析算法，但 LLVM 和 GCC 两大编译器框架在工程实现上最终都选用了手写递归下降语法分析器，而没有使用 LR 等支持根据语法自动生成解析动作的语法分析算法，其中的设计考虑值得分析。

4.1 自由度与可定制性

手写的递归下降分析使用户可以自定义所有的分析方法，而无需遵循 LR 等形式化规则。我们可以根据语言的语法特性或一些边界情况，设计特别的优化分析方法，很多时候可以实现比形式化方法更快的分析速度。

例如 Clang Parser 再分析 C 语言数组声明时，会针对两种最常见的情形（空数组、常量表达式数组）进行特别处理，避免较深的函数调用：

```
void Parser::ParseBracketDeclarator(Declarator &D) {
```

```
...

  // C array syntax has many features, but by-far the most common is
      [] and [4].
  // This code does a fast path to handle some of the most obvious
      cases.
  if (Tok.getKind() == tok::r_square) {
    T.consumeClose();
    ParsedAttributes attrs(AttrFactory);
    MaybeParseCXX11Attributes(attrs);

    // Remember that we parsed the empty array type.
    D.AddTypeInfo(DeclaratorChunk::getArray(0, false, false, nullptr,
                                    T.getOpenLocation(),
                                    T.getCloseLocation()),
             std::move(attrs), T.getCloseLocation());
    return;
  } else if (Tok.getKind() == tok::numeric_constant &&
             GetLookAheadToken(1).is(tok::r_square)) {
    // [4] is very common. Parse the numeric constant expression.
    ExprResult ExprRes(Actions.ActOnNumericConstant(Tok,
        getCurScope()));
    ConsumeToken();

    T.consumeClose();
    ParsedAttributes attrs(AttrFactory);
    MaybeParseCXX11Attributes(attrs);

    // Remember that we parsed a array type, and remember its features.
    D.AddTypeInfo(DeclaratorChunk::getArray(0, false, false,
        ExprRes.get(),
                                    T.getOpenLocation(),
                                    T.getCloseLocation()),
                  std::move(attrs), T.getCloseLocation());
    return;
  }

  ...
```

```
}
```

4.2 工作量

手写分析器的工作量无疑大于借助工具自动生成。但从编译器整体架构角度看，语法分析部分并不作为现代编译器设计的核心。从难度上看，语义分析部分的复杂度比它大很多 [3]；从发展趋势看，现代编译器设计对中间代码优化、添加并行性支持等组件给予更多重视。因此，追求语法分析器设计的形式化与高效显得并没有那么重要。能够进行正确分析的递归下降分析器对大多数编译器项目来说已经足够好用。

4.3 可解释性

在工程实践中，自动的分析器生成工具常常会产生缺乏可解释性的行为，无法按照使用者意图生成分析器。例如文法无二义性或可以消除二义性时，生成工具有时仍然会报告错误。生成工具的使用体验常常给开发人员带来困惑与痛苦。

4.4 性能

GCC 在转用递归下降分析器后，运行耗时得到 1.5% 的提升 [7]，并不明显。事实上，语法分析程序的行为与此前相比并没有明显区别，只是此前的显式移进、规约与状态转换过程被函数递归调用替代。[3] 但根据 4.1 一节，递归下降分析器对后续优化添加的支持无疑是更友好的。

4.5 对错误检测的支持

递归下降分析器提供的开放性与可定制性也使得错误检测变得容易。用户可以针对特殊情形设计具体细致的错误检测方法，生成详细的错误诊断并提供错误恢复。相比之下，形式化的 LR 语法分析器自动产生的错误信息并没有很好的可阅读性。

4.6 可扩展性与可维护性

当程序设计语言进行版本迭代引入新特性、新语法时，递归下降分析器更易于修改，尤其是在已有程序基础上添加临时规则与补丁。自动生成的 LR 分析器具有较差的可扩展性，新添加的语言特性往往会让已有的 LR 分析器产生错误，需要重新调整、生成。

递归下降分析器的代码往往也具有更好的可读性，便于维护。

4.7 The Lexer Hack

在处理类型/变量名二义性问题上，如果自动生成的 LR 分析器像 2.6 中 Clang Parser 使用的方法一样，将自定义类型与标识符不作区分，会产生大量无法解决的移进-移进冲突，因此必须设置特殊的记号名，这意味着第二次遇到该字符串时，在词法分析阶段就需要得到该记号的类型信息。LR 分析器使用 The Lexer Hack[4] 技巧，将语义分析阶段的信息前馈给词法分析阶段。由于混合了词法分析及语义分析，The Lexer Hack 通常被认为并不优雅。相比之下，手写分析器允许自定义类型同样使用 identifier 作为记号名，到语法分析阶段再通过 Sema 接口查询符号表信息，实现更加清晰。

4.8 小结

手写递归下降分析在性能、可定制性、可维护性等各项指标上达到了相对的平衡，同时便于进行扩展、优化，提供更好的错误检测支持。尽管自动生成的 LR 分析器也有其优势，但目前主流编译器框架的选择已经说明，至少在工程实现上，递归下降是更优的一方。

5 结语

用一篇调研论文为这学期的编译课程收尾似乎是个不错的选择。虽然课程学习过程中已经花了很多精力理解各类语法分析理论，从 Clang/GCC 源码入手分析主流编译框架的语法分析实现依然收获颇丰。它们的语法分析器实现并没有使用 LR 等形式化方法，而是采取自由度更高的手写递归下降策略。源码阅读过程中，我也从针对特定语法、边界情形的优化中，更

深刻地理解了“手写”的含义：那就是把分析函数核心逻辑的控制权交给用户，而非使用自动化工具。在这一层面上，手写递归下降分析牺牲便利换取自由。当然这只是语法分析方法选择的 tradeoff 之一，其余如错误诊断、可解释性等方面，本文并没有做特别深入具体的探究，这些话题都是值得探索的。

这次的调研再次反映了编译原理的深度与广度。在理论层面，它涉及的语言、文法等计算理论可能是计算机科学最形式化、抽象层次最高的领域。在工程实现层面，编译器框架涉及组件模块数量之多，使得模块化设计、面向对象设计等软件工程开发方法被广泛应用。高山仰止，也只能感叹自己才疏学浅，道阻且长。

最后感谢冯老师以及教学团队的指导！

参考文献

[1] Eli Bendersky. How Clang handles the type / variable name ambiguity of C/C++. `https://eli.thegreenplace.net/2012/07/05/how-clang-handles-the-type-variable-name-ambiguity-of-cc`, 2017. [Online, accessed 05-January-2023].

[2] Stackoverflow contributors. Why can't C++ be parsed with a LR(1) parser? `https://stackoverflow.com/questions/243383/why-cant-c-be-parsed-with-a-lr1-parser`, 2008. [Online, accessed 05-January-2023].

[3] Stackoverflow contributors. Are GCC and Clang parsers really handwritten? `https://stackoverflow.com/questions/6319086/are-gcc-and-clang-parsers-really-handwritten`, 2011. [Online, accessed 05-January-2023].

[4] Wikipedia contributors. Lexer hack. `https://en.wikipedia.org/wiki/Lexer_hack`, 2023. [Online; accessed 05-January-2023].

[5] Min-Yih Hsu. *LLVM Techniques, Tips, and Best Practices Clang and Middle-End Libraries: Design Powerful and Reliable Compilers Using the Latest Libraries and Tools from LLVM*. Packt, 2021.

[6] Edaqa Mortoray. Why I don't use a Parser Generator. `https://mortoray.com/why-i-dont-use-a-parser-generator/`, 2012. [Online; accessed 05-January-2023].

[7] Joseph S. Myers. New C Parser. `https://gcc.gnu.org/wiki/New_C_Parser`, 2004. [Online, accessed 05-January-2023].

[8] LLVM Project. Clang - Features and Goals. `https://clang.llvm.org/features.html#expressivediags`. [Online, accessed 05-January-2023].

[9] Jim Roskind. Jim Roskind on C ambiguity. `https://pdos.csail.mit.edu/archive/l/c/roskind.html`, 1992. [Online, accessed 05-January-2023].