Таким образом, листья AST являются пустыми операторами и представляют только переменные и константы.

AST отличается от дерева разбора тем, что в нём отсутствуют узлы и рёбра для тех синтаксических правил, которые не влияют на семантику программы. Например:

- отсутствует информация о скобках – она задается структурой дерева
- вышеупомянутое упрощение $numterm$ – $-leafnode \rightarrow leafnode : val$

Обычно всё незначимая подцепочка просто заменяется на значение(я) из терминала(ов).

```
TranslationUnitDecl 0x58e128 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x58e9c0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x58e6c0 '__int128'
|-TypedefDecl 0x58ea30 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x58e6e0 'unsigned __int128'
|-TypedefDecl 0x58ed38 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x58eb10 'struct __NSConstantString_tag'
|   `-Record 0x58ea88 '__NSConstantString_tag'
|-TypedefDecl 0x58edd0 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x58ed90 'char *'
|   `-BuiltinType 0x58e1c0 'char'
|-TypedefDecl 0x58f0c8 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x58f070 'struct __va_list_tag [1]' 1
|   `-RecordType 0x58eeb0 'struct __va_list_tag'
|     `-Record 0x58ee28 '__va_list_tag'
`-FunctionDecl 0x5ebe50 <1.c:1:1, line:3:1> line:1:5 f 'int (int, int)'
 (|-ParmVarDecl 0x5ebcf8 <col:7, col:11> col:11 used a 'int'
  |-ParmVarDecl 0x5ebd78 <col:14, col:18> col:18 used b 'int')
{}`-CompoundStmt 0x5ec048 <col:21, line:3:1>
    `-ReturnStmt 0x5ec038 <line:2:2, col:15>
      `-BinaryOperator 0x5ec018 <col:9, col:15> 'int' '/'
        |-ParenExpr 0x5ebfd8 <col:9, col:13> 'int'
        | `-BinaryOperator 0x5ebfb8 <col:10, col:12> 'int' '+'
        |   |-ImplicitCastExpr 0x5ebf88 <col:10> 'int' <LValueToRValue>
        |   | `-DeclRefExpr 0x5ebf48 <col:10> 'int' lvalue ParmVar 0x5ebcf8 'a' 'int'
        |   `-ImplicitCastExpr 0x5ebfa0 <col:12> 'int' <LValueToRValue>
        |     `-DeclRefExpr 0x5ebf68 <col:12> 'int' lvalue ParmVar 0x5ebd78 'b' 'int'
        `-IntegerLiteral 0x5ebff8 <col:15> 'int' 2
```

Рис. 8: Clang AST для функции целочисленного осреднения 2 целых чисел

Понятно, что структура элементов дерева укладывается в иерархии. Здесь следует отметить 2 момента по программированию:

- У 2 разных корней (ноды разных категорий) может не быть общего предка, и на практике они наследованы от разных базовых классов. То есть иерархически по классам дерево получается не деревом, а лесом. И методы для каждого дерева из леса могут быть различными.
- Представим, мы находимся в вершине дерева A, и нам нужно сделать кодогенерацию для дочерней вершины B, которая может быть типов $C_1, C_2, C_3, .., C_n$. Следовательно, в функцию $CG :: GenerateCodeA$ придётся вставить switch-case на n элементов для каждой из альтернатив C_i. Но так придётся делать для каждой из функций!

Напрашивается способ, как решить вышеуказанные моменты изящно. Для этого служит паттерн ООП «Visitor», который мы рассматривать не будем. Любой модуль, использующий AST для своих целей (AST Consumer) реализует в себе такой «Visitor».

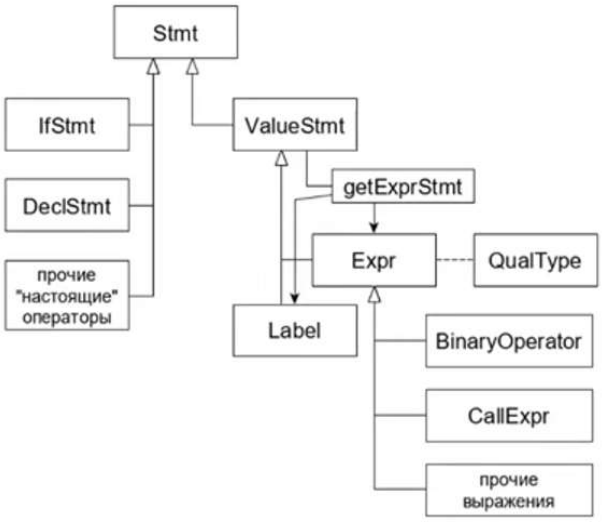

Рис. 9: Clang FE: Иерархия Stmt в Clang AST

7.2 Синтаксический разбор

Как правило, в компиляторах на данный момент доминируют 3 способа построения AST по входной программе:

- LALR(1)-парсинг + модификации парсера для специфических операций типа «составление таблицы символов», etc. Использовался ранее в GCC до версии 3.X.X, затем был переработан во вручную написанный рекурсивный спуск.
- Рекурсивный спуск, написанный вручную. Используется в Clang, современном GCC, Rust C и др.
- GLR-анализ – обобщенный LR-разбор, как правило, использующий GLR-парсеры общего назначения, частично доработанные. Пример – Elsa C++ Parser.

Если смотреть по соотношению в индустрии, подход №2 с рекурсивным спуском существенно доминирует. Почему так? Этому есть, как минимум, 4 причины:

7.3 Лексический анализ C-подобных языков

Проблемы:

- Есть 2 типа токенов (или даже больше!) – Token и PreprocessingToken (для макроопределений)

```
int 'int'         [StartOfLine]  Loc=<1.c:1:1>
identifier 'f'    [LeadingSpace] Loc=<1.c:1:5>
l_paren '('              Loc=<1.c:1:6>
int 'int'                Loc=<1.c:1:7>
identifier 'a'    [LeadingSpace] Loc=<1.c:1:11>
comma ','                Loc=<1.c:1:12>
int 'int'         [LeadingSpace] Loc=<1.c:1:14>
identifier 'b'    [LeadingSpace] Loc=<1.c:1:18>
r_paren ')'              Loc=<1.c:1:19>
l_brace '{'       [LeadingSpace] Loc=<1.c:1:21>
return 'return'   [StartOfLine] [LeadingSpace]   Loc=<1.c:2:2>
l_paren '('       [LeadingSpace] Loc=<1.c:2:9>
identifier 'a'           Loc=<1.c:2:10>
plus '+'                 Loc=<1.c:2:11>
identifier 'b'           Loc=<1.c:2:12>
r_paren ')'              Loc=<1.c:2:13>
slash '/'                Loc=<1.c:2:14>
numeric_constant '2'              Loc=<1.c:2:15>
semi ';'                 Loc=<1.c:2:16>
r_brace '}'       [StartOfLine]  Loc=<1.c:3:1>
eof ''           Loc=<1.c:3:2>
```

Рис. 10: Токены для функции целочисленного осреднения 2 целых чисел

Интуитивно бы сделать 1 лексер на 2 класса токенов. Но в некоторых компиляторах, например в Clang, все с точностью до наоборот – 2 лексера (Lexer и TokenLexer) и один класс токенов (Token)! Как результат, при обработке, например, `include`, нужно поддерживать целый стек лексеров, какие-то из которых просто лексеры, а какие-то TokenLexer.

7.4 Взаимодействие компонент фронтенда

В учебниках по компиляторам часто пишут, что взаимодействие компонент выглядит как конвейер:

лексический –> синтаксический –> семантический анализ

Это крайне грубое представление о работе современных компиляторных фронтендов. В следующем подразделе мы покажем, что в деталях это совсем не так.

7.5 Clang как фронтенд

При вызове `clang -cc1` создаётся экземпляр класса `Clang::CompilerInstance` в методе `cc1_main`, в нём выставляется базовое действие, которое должен сделать фронтенд[14]. Действие активируется `Act`, после чего Clang его выполняет.

[14]только одно, поэтому clang не может одновременно, например, скомпилировать программу (-emit-obj) и сдампить AST (-ast-dump)

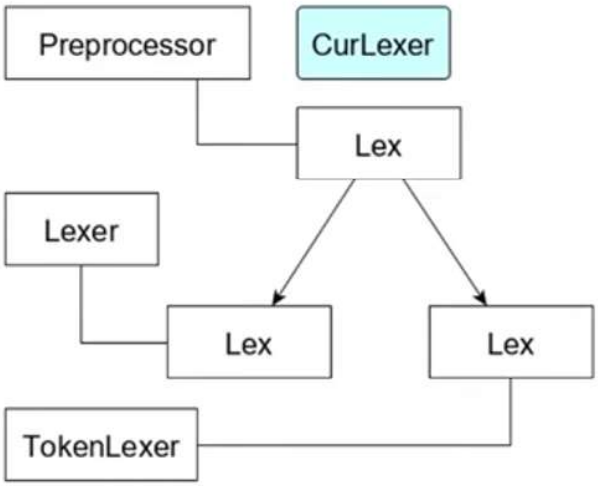

Рис. 11: 2 лексера в Clang

7.5.1 Иерархия базовых действий:

Как правило, мы хотим что-то выводить – у нас `Emit<actionname>Action`. Стоит отметить, что:

- Все такие действия наследуют от `CodeGenAction`
- CodeGenAction делает CodeGen консьюмером для AST
- ASTFrontendAction добавляет использование семантического анализа
- Точка входа при таких действиях: `ParseAST`.

7.5.2 Парсинг в Clang

Главный в парсинге – Parser. Его задача – подготовить AST, далее включаются все продьюсеры и потребляют AST. Лексер – однопроходный и не зависит от парсера. Это 2 лексера, описанных выше. Стек лексеров хранит объект класса Preprocess, он и является настоящим лексером в Clang. Семантический модуль Sema, по теории, не должен зависеть от лексера, но он от него зависит! (SIC).

Парсер – это рукописный рекурсивный спуск, как было сказано ранее. То есть написан набор методов `Parser:Parse<XYZ>`, по функции для каждого нетерминала. Если в ходе парсинга происходит ошибка, в принципе, предпостроенная часть AST имеет право на существование, а в месте, в котором возник затык, вставляется вершина с записью о возможной ошибке

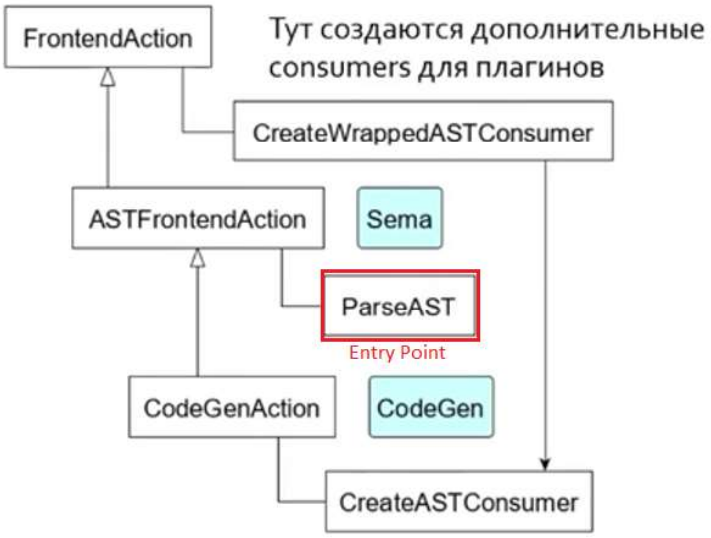

Рис. 12: Clang FE: Иерархия действий при парсинге

(ошибки вставляются по сопоставлению с большим enum'ом). Также есть опция `-fixit`, позволяющая исправлять простейшие синтаксические ошибки.

LALR(1) не используется, потому что C/C++ языки, для которых:

- Грамматика ну «почти» регулярная – довольно простая
- При этом язык на самом деле контекстно-зависимый
- Потенциально мало бектрекинга
- Довольно строгий стандарт
- Много особых случаев, которые гораздо проще прописывать вручную

Проблемы:

- Невозможность раннего определения идентификаторам категории (лексер даже не пытается). Поток токенов ну оооочень простой. Парсер должен по грамматике догадаться по грамматике, что это. А сам язык сложный.
- Бектрекинг может быть необходим при таком подходе!

Бектрекинг в лексере: интерфейс (завёрнутый в `TentativeParsingAction`-объект)

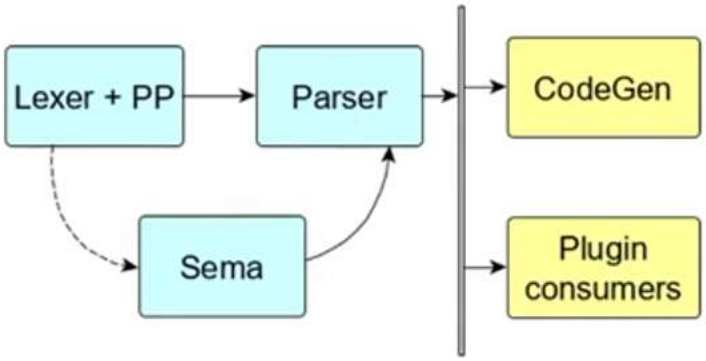

Рис. 13: Clang FE: Диаграмма зависимостей при парсинге / консьюминге AST

- EnableBacktrackAtThisPos[15] – запомнить точку отката
- CommitBacktrackedTokens – забыть
- Backtrack – откатиться

Следовательно: лексер поддерживает бектрекинг, после которого подпоследовательность снова считывается, и снова сопоставляется по грамматике парсером. Это довольно накладно по производительности, поэтому придумали ещё один тип токенов – аннотирующие.

Как правило, их используют для `typename, scope_identifiers`. Парсер внедряет этот токен в последовательность токенов для указания, что уже понял, что это за тип и т.д. (проверка условия: `if TryAnnotateTypeOrScopeToken()`, установка: `setTypeAnnotation(tok,ty)`).

Мало того, парсер может вставлять не только такие токены, а и вообще любые. `ExpectAndConsume`. Иногда это используется для обработки ошибок.

7.5.3 Семантический анализ

Семантический анализ выполняется модулем Sema по вызову из модуля Parser.

Семантический анализ происходит по схеме: Parse(XX) –> Sema::ActOn(XX) –> Ok –> change AST / No –> err, то есть AST строит именно семантический анализатор.

То есть Sema по сути решает 2 задачи:

- Ищет ошибки
- Строит AST

[15]Этот вызов может стэкаться

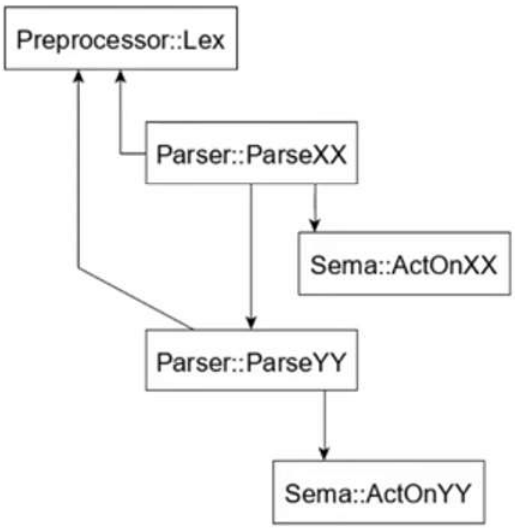

Рис. 14: Clang FE: Parser + Sema

8 Приложение

8.1 Необходимые определения из близких областей

8.1.1 Графы

В данном курсе мы будем рассматривать только конечные ориентированные помеченные графы, подразумевая под «графами» именно такие графы, если не указано противное.

Опр. 8.1 *Граф $G = (V, E, L)$, где V — конечное множество вершин, E — конечное множество рёбер, L*

Опр. 8.2 *Отношением достижимости на графе в смысле нашего определения называется двухместное,*

Опр. 8.3 *Транзитивным замыканием графа называется транзитивное замыкание отношения достижимости по всему графу.*

8.1.2 Матрицы

Список литературы

[1] Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений. Санкт-Петербург : Вильямс, 2008.

[2] http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_за_O(n%5E2)_с_построением_пар_различимых_состояний

[3] http://neerc.ifmo.ru/wiki/index.php?title=Минимизация_ДКА,_алгоритм_Хопкрофта_(сложность_O(n_log_n))

[4] http://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Бржозовского

[5] https://habr.com/ru/post/171667, 2013 (перевод, оригинал тоже гуглится).

[6] Compilers: Principles, Techniques, and Tools is a computer science textbook by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D.