MeKu Software

Как сделать front-end компилятора в домашних условиях

Кто мы такие?

А что это вообще такое?

- Статический анализатор для языка Solidity
- Solidity – язык для написания смарт-контрактов
- Используется в Ethereum и других проектах
- Имеет JS-подобный синтаксис

```
pragma solidity ^0.8.10;

contract SimpleStorage
{
  uint public num;
  function set(uint _num) public { num = _num; }
  function get() public view returns (uint)
  { return num; }
}
```

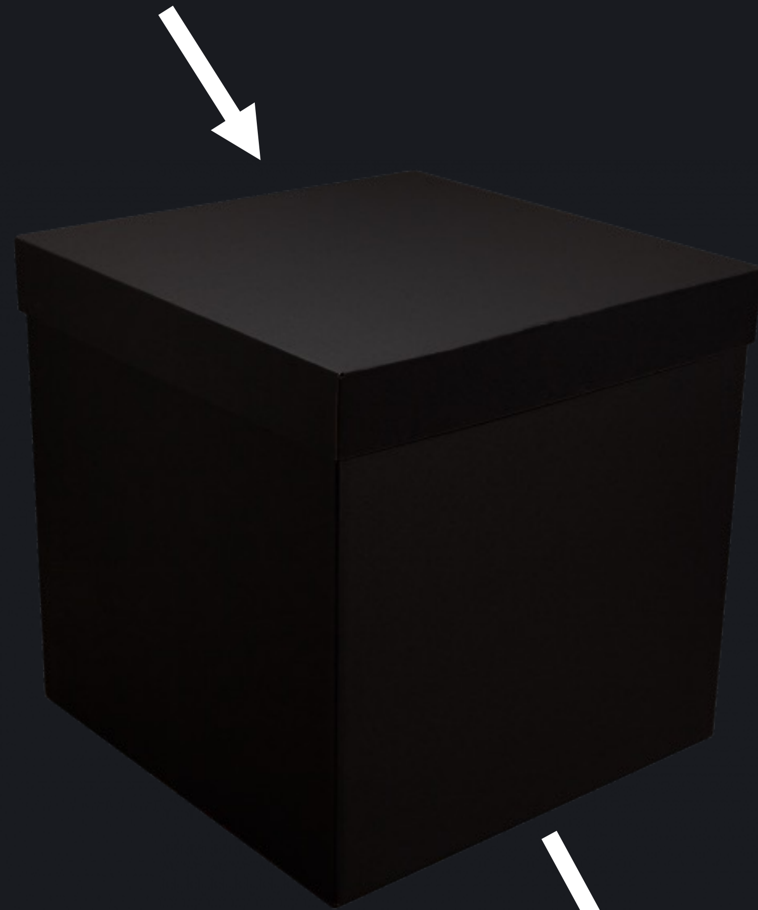
Почему не затянуть готовый фронт-энд?

- «Если хочешь что-то сделать хорошо – сделай это сам!»
- У Solidity множество версий, которые обратно-несовместимы
  - К примеру, в версиях после 0.5 отсутствует throw keyword
- Компилятор распространяется под GPL
- Требуется расширяемость нод дополнительной метаинформацией

MeKu Software

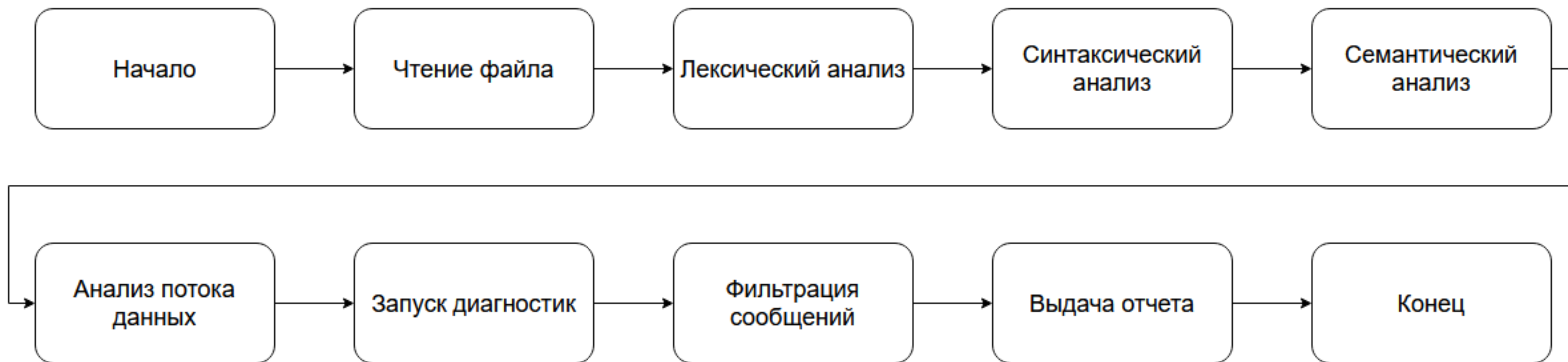
Структура фронт-энда

Файл с исходным кодом



Какой-то вывод

На самом деле "чёрных коробок" много



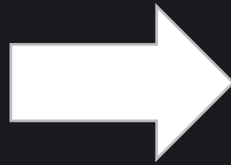
MeKu Software

Lexer

Зачем нужен лексический анализ?

- Помогает упростить итоговую грамматику парсера
- Другими словами, облегчает работу парсера тем, что анализирует поток символов, переводя его в поток токенов
- Токен – идентификатор, соответствующий ключевому слову/спецсимволу/идентификатору или любой другой символьной конструкции, имеющийся в языке

```
int foo = bar + 1;
```



```
tk_int,
tk_identifier "foo",
tk_eq,
tk_identifier "bar",
tk_add,
tk_int_literal "1",
tk_semicolon,
tk_eof
```

Lexer

- Написан руками
- Раздельная обработка спецсимволов и ключевых слов
- Есть кеш токенов
- Оптимизация разбора ключевых слов: префиксное дерево

```
enum class kind { .... };

struct token
{
  kind type;
  std::string_view pos;
};
```

Prefix tree

```
constant
constructor
continue
```

Prefix tree

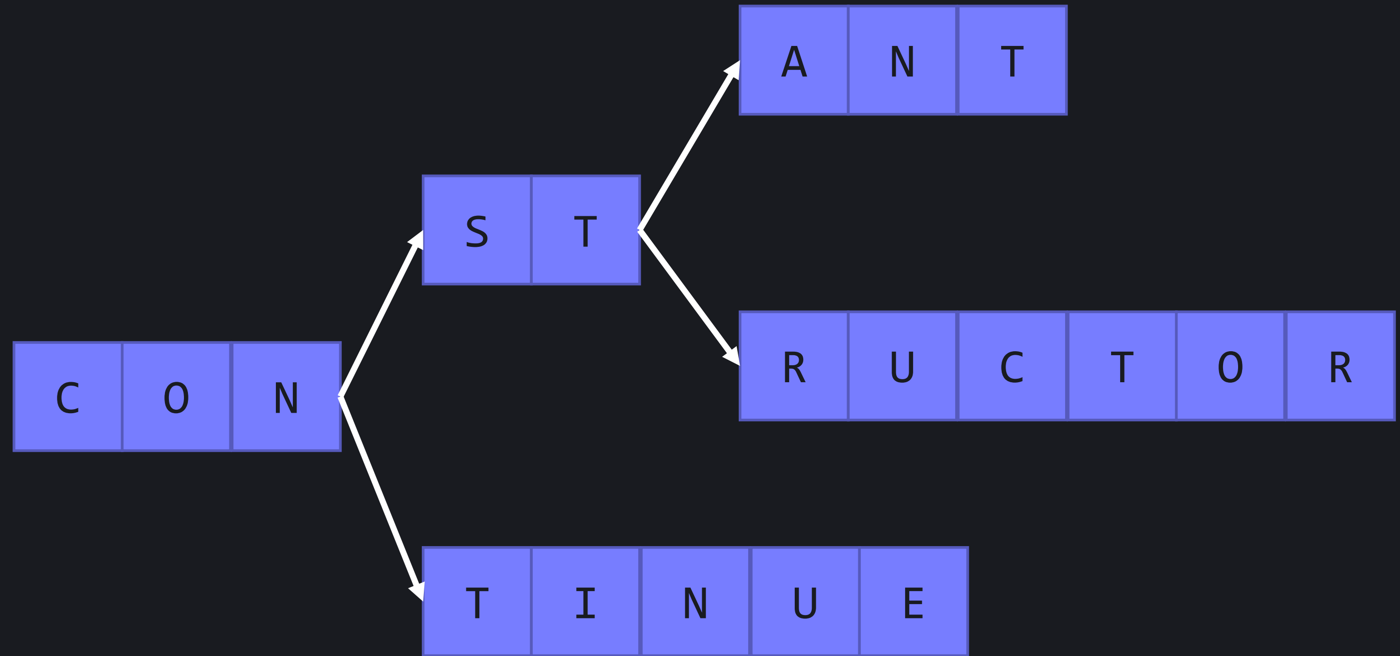
```
constant
constructor
continue
```

Prefix tree

constant

constructor

continue



AST

AST

- Предоставляет возможность работать с языковыми конструкциями
- Определяется грамматикой и семантикой целевого языка

AST

- Базовый класс узла – ast_node
- Дерево больше похоже на дерево разбора с «накрученой» семантикой
- Узел может быть терминальным и нетерминальным
  - Терминальные узлы имеют ссылку на позицию в файле
  - Нетерминальные содержат список дочерних узлов (std::vector)

Псевдо-контейнер параметров

- Позволяет работать со списком узлов
  - Указывает тип дочерних узлов
  - Указывает «шаг» (например, при наличии запятых)

```
template <class GetAs = ast::ast_node,
        size_t DerefCount = 2,
        class Node = ast::ast_node>
struct parameters;
/* foo{opt1: arg, opt2: val}() */
struct funcall_options_expression : ....
{
  ast_node &get_args() noexcept;

  ast::proxy::parameters<ast::named_argument>
      get_arg_list() noexcept { return get_args(); }

};
```

Pattern matching

- Основной интерфейс – функция try_get_as

```
expr.try_get_as<ast::member_expression>();
```

- Работает по типу std::visit
- Описана в прошлом докладе: https://www.youtube.com/watch?v=gTnd_l75938

MeKu
Software

Parser

Немного теории

- Язык программирования задается формальной грамматикой
- Формальная грамматика описывает язык с помощью двух видов символов:
  - Терминал – конкретная языковая конструкция, например, ключевое слово
  - Нетерминал – некая сущность языка (например, выражение), не имеющая конкретного символьного значения
- Символы используются в продукциях для описания конструкций языка

Немного теории

- Парсер выполняет синтаксический анализ заданного языка
- Может быть сгенерирован или написан вручную
- Парсеры можно охарактеризовать следующими параметрами:
  - Направление чтения входного потока: слева-направо (L), или справа-налево (R)
  - Производит наиболее левую продукцию (L), либо наиболее правую (R)
  - Количество символов предпросмотра (число k)
- Синтаксический анализатор на основе рекурсивного спуска является LL(k) анализатором, чаще всего k=1

Немного теории

- Не каждый парсер может разобрать заданную грамматику
- LL-анализатор не может разбирать грамматики с левой рекурсией:

```
<Exp> ::= <Exp> '+' <Term> | <Term>
```

- Такая проблема решается преобразованием грамматики с устранением рекурсии, например:

```
<Exp> ::= (<Term> '+')* <Term>
```

- Главное – сохранить семантику грамматики, не нарушив ассоциативность вычислений

Структура синтаксического дерева

- ast_node [Derived Types]
  - assembly_statement
  - ast_node_symbol
    - ast_node_scope
      - block
      - catch_clause
      - common_contract
        - contract_definition
        - interface_definition
        - library_definition
      - common_function
        - constructor_definition
        - fallback_function_definition
        - function_definition
        - modifier_definition
        - receive_function_definition
      - enum_definition
      - error_definition
      - event_definition
      - for_statement
      - import_directive
      - source_unit
      - struct_definition
    - common_variable
      - const_variable_declaration
      - error_parameter
      - event_parameter
      - parameter_declaration
      - state_variable_declaration
      - struct_member
      - variable_declaration
      - variable_declaration
    - enum_value_definition
    - user_defined_value_type_definition
  - break_statement
  - call_argument_list
  - continue_statement
  - do_while_statement
  - emit_statement
  - empty_statement
  - expression
    - array_expression
    - binary_expression
    - cast_expression
    - declaration_tuple_expression
    - expression_symref
      - funcall_expression
      - funcall_options_expression
      - literal_expression
      - member_expression
      - paren_expression
    - inline_array_expression
    - new_expression
    - payable_conversion_expression
    - postfix_expression
    - prefix_expression
    - ternary_expression
    - tuple_expression
    - type_metadata_expression
  - expression_statement
  - function_return_info
  - identifier_path
  - identifier_path
  - if_statement
  - import_aliases
  - inheritance_specifier
  - modifier_invocation
  - named_argument
  - override_specifier
  - parameter_list
  - path
  - pragma
  - return_statement
  - revert_statement
  - throw_statement
  - try_statement
  - type_name
    - address_payable
    - array_typename
    - function_typename
    - mapping_typename
    - tuple_type
    - type_name_identifier
  - unchecked_block
  - using_directive
  - variable_declaration_statement
  - variable_declaration_tuple
  - while_statement

Parser

- Выполняет синтаксический анализ
- Умеет разруливать неочевидности
  - отличает декларацию от выражения
  - есть костыли для спец. синтаксиса (abi.decode builtin)
- Самописный, на основе рекурсивного спуска
- Управляет памятью узлов дерева

abi.decode???

```
expression:
        ....
    | (
        identifier
        | literal
        | literalWithSubDenomination
        | elementaryTypeName[false]
      ) # PrimaryExpression
```

abi.decode???

```
expression:
        ....
    | (
        identifier
        | literal
        | literalWithSubDenomination
        | elementaryTypeName[false]
      ) # PrimaryExpression
```

abi.decode???

- Т.е. с точки зрения синтаксиса это OK:

```
uint x = uint + int256;
```

- Такая конструкция нужна для следующей builtin-функции, например:

```
abi.decode(data, (uint, address, uint[], MyStruct))
```

abi.decode???

- В парсере на основе рекурсивного спуска можно легко обработать этот случай:

```
/* parser::p_postfix_expression() */
case kind::tk_lparen:

if (is_abi_decode_expr(*expr))
{
  /*
   * abi.decode(....) call requires special parsing
   * sequence because of tuple-of-types syntax.
   */
  expr = p_abi_decode_call(*expr);
  break;
}

/* regular parsing continues... */
```

Decl or Expr?

- В Solidity объявления переменных и выражения можно смешивать, как и в любом другом современном языке программирования
- Например, вот так выглядит декларация переменной:

```
a.b.c[42] d;
```

- А вот так будет выглядеть выражение:

```
a.b.c[42] = d;
```

Decl or Expr?

- Различие лишь в том, что после идентификатора/массива/member expression всегда будет идти имя переменной, если это выражение – то будет что-либо еще
- Этот случай обрабатывается специальным алгоритмом внутри парсера на основе этого подхода

```
a.b.c[42] d;
a.b.c[42] = d;
```

Интерфейс синтаксического анализатора

```
class parser
{
public:
    parser(lexer &lexer, errors::error_dispatcher &disp) : m_err_disp { disp }, lex(lexer), pool() {}

    parser(const parser &) = delete;
    parser &operator=(const parser &) = delete;

    friend inline void* operator new(size_t count, parser& p) noexcept;
    friend inline void operator delete(void*, parser& p) noexcept;

    template <class Node, class ...Args>
    Node* make_node(Args &&...args) { ... }

    node_allocator<ast::ast_node *> get_allocator() noexcept { return node_allocator<ast::ast_node *> { pool }; }
    node_alloc &get_pool() noexcept { return pool; }
    errors::error_dispatcher &get_dispatcher() noexcept { return m_err_disp; }

    lexer& get_lexer();
    ast::source_unit* p_source_unit(std::string_view file);

private:
    ast::ast_node* p_pragma();
    ast::ast_node* p_import_aliases();
    ast::ast_node* p_import_directive();
    ast::ast_node* p_contract_body_element();
    std::optional<ast::inheritance_def> p_inheritance_specifier_list();
    ast::ast_node* p_contract_definition();
```

Интерфейс синтаксического анализатора

```cpp
    ast::source_unit* p_source_unit(std::string_view file);

private:
    ast::ast_node* p_pragma();
    ast::ast_node* p_import_aliases();
    ast::ast_node* p_import_directive();
    ast::ast_node* p_contract_body_element();
    std::optional<ast::inheritance_def> p_inheritance_specifier_list();
    ast::ast_node* p_contract_definition();
    ast::ast_node* p_interface_definition();
    ast::ast_node* p_library_definition();
    ast::ast_node* p_function_definition();
    ast::ast_node* p_constructor_definition();
    ast::ast_node* p_event_parameter();
    ast::ast_node* p_fallback_function_definition();
    ast::ast_node* p_modifier_definition();
    ast::ast_node* p_receive_function_definition();
    ast::ast_node* p_state_variable_declaration();
    ast::ast_node* p_constant_variable_declaration();
    ast::ast_node* p_event_definition();
    ast::ast_node* p_enum_definition();
    ast::ast_node* p_error_parameter();
    ast::ast_node* p_error_definition();
    ast::ast_node* p_struct_member();
    ast::ast_node* p_struct_definition();
    ast::ast_node* p_user_defined_value_type_definition();
    ast::ast_node* p_using_directive();
    ast::ast_node* p_modifier_invocation();
```

```cpp
    ast::ast_node* p_override_specifier();
    ast::ast_node* p_identifier_path(bool type_id = false);
    ast::ast_node* p_named_argument();
    ast::call_argument_list* p_call_argument_list();
    ast::type_name* p_elementary_typename(bool allow_address_payable = false);
    ast::type_name* p_mapping_key_type(bool allow_address_payable = false);
    ast::type_name* p_type_name2();
    ast::type_name* p_type_name();
    ast::ast_node* p_variable_declaration(bool parameter, ast::type_name* type_name = nullptr);
    ast::parameter_list* p_parameter_list();
    ast::function_return_info* p_function_returns();
    ast::ast_node* p_visibility();
    ast::ast_node* p_state_mutability();
    ast::ast_node* p_data_location();
    ast::expression* p_expression(bool parse_comma = true);
    ast::expression *p_bin_expression(unsigned prio = 14);
    ast::expression *p_abi_decode_call(ast::expression &expr);
    ast::expression *p_postfix_expression();
    ast::expression* p_unary_expression();
    ast::expression* p_primary_expression();

    // True if expression is parsed, false otherwise
    std::pair<ast::ast_node*, bool> p_decl_or_expr(bool parse_comma = true);
    ast::statement* p_statement();
    ast::statement* p_empty_statement();
    ast::block* p_block();
    ast::statement *p_throw_statement();
    ast::statement* p_if_statement();
```

Пример разбора грамматической конструкции

```
returnStatement: Return expression? Semicolon;
```

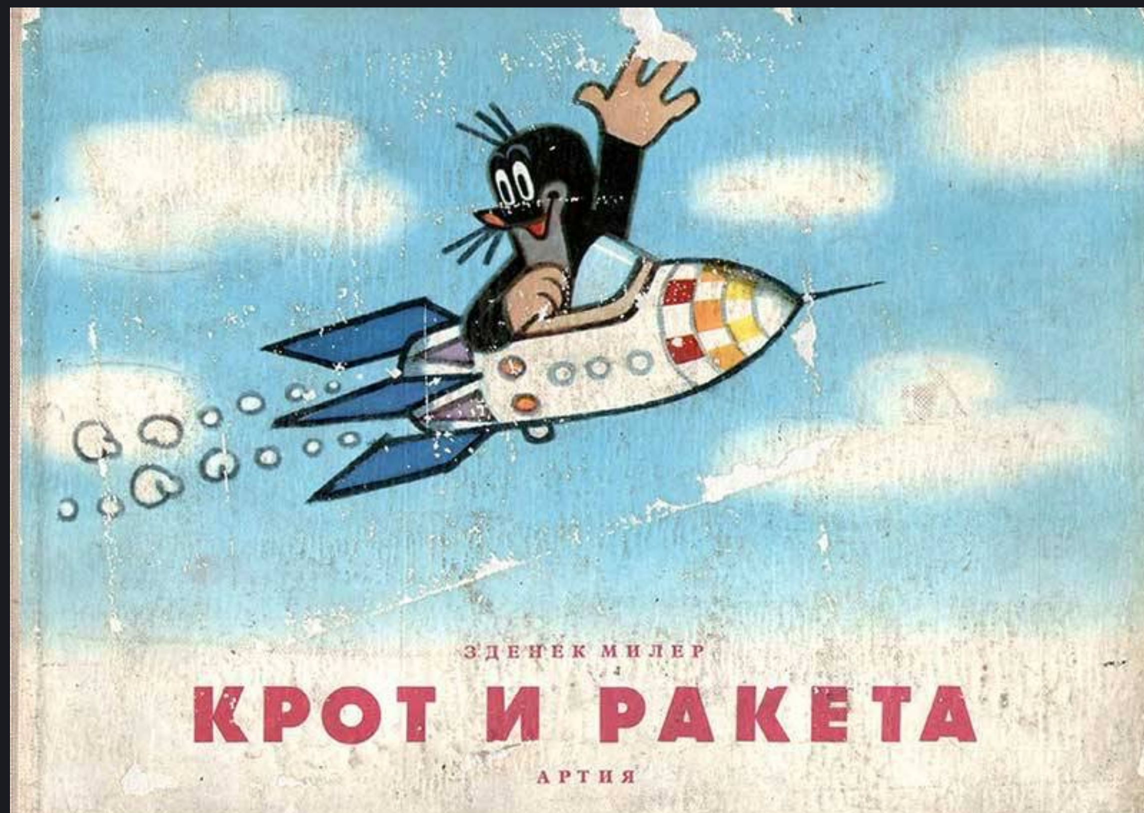
```
ast::statement *parser::p_return_statement()
{
    if (lex.look_ahead(0).get_type() != kind::tk_return)
    {
        //errors::parser_error("return_statement");
        return nullptr;
    }
    auto tkreturn = lex.get_token();

    if (lex.look_ahead(0).get_type() == kind::tk_semicolon)
    {
        return make_node<ast::return_statement>(*this, tkreturn, nullptr, lex.get_token());
    }

    auto expr = p_expression();
    if (!expr || lex.look_ahead(0).get_type() != kind::tk_semicolon)
    {
        //errors::parser_error("return_statement");
        return nullptr;
    }

    return make_node<ast::return_statement>(*this, tkreturn, expr, lex.get_token());
}
```

MeKu Software



Sema

Sema

- Семантический анализатор
- Содержит таблицу символов
- Система типов на основе узлов дерева
- Умеет выбирать перегрузки
- Считает константы произвольной точности
  - BeeNum – предоставляет 2 числовых типа – Bint и Brat – произвольного размера с автоматическим расширением (в отличие от llvm::APInt)

Symbol table

- Структура данных для связки идентификаторов с их определением
- На каждую область видимости создается отдельная таблица
- Каждая таблица имеет ссылку на следующую
  - Кроме глобальной области видимости
- Запись содержит информацию о символе
  - Имя
  - Тип
  - Значение
  - Декларация

Lookup

- Есть 2 класса: scope и LookupInfo
- Класс scope содержит таблицу символов и ссылку на следующий scope
- LookupInfo содержит состояние поиска:
  - Текущий scope
  - Базовые классы
  - Информация о виртуальных функциях
- Вся логика регистрации нового символа содержится в классе scope
- Логика поиска – в LookupInfo
  - Позволяет инкапсулировать состояние и избавиться от лишних параметров

Тестирование

- Содержание тестовой базы представляло следующий набор данных:
  - Файлы с ресурса solidity by example с примерами для новичков;
  - Реальные библиотеки для разработки смарт-контрактов (OpenZeppelin);
  - Известные смарт-контракты, работающие в реальной сети (Tether);
  - Вручную написаные тесты.

Тестирование

Microsoft Visual Studio Debug Console

```
[ RUN      ] rule_tests.rule_7_test
[       OK ] rule_tests.rule_7_test (4 ms)
[ RUN      ] rule_tests.rule_8_test
[       OK ] rule_tests.rule_8_test (3 ms)
[ RUN      ] rule_tests.rule_8_test2
[       OK ] rule_tests.rule_8_test2 (2 ms)
[ RUN      ] rule_tests.rule_9_test
[       OK ] rule_tests.rule_9_test (6 ms)
[ RUN      ] rule_tests.rule_10_test
[       OK ] rule_tests.rule_10_test (4 ms)
[ RUN      ] rule_tests.rule_11_test
[       OK ] rule_tests.rule_11_test (6 ms)
[ RUN      ] rule_tests.rule_12_test
[       OK ] rule_tests.rule_12_test (2 ms)
[ RUN      ] rule_tests.rule_51_test
[       OK ] rule_tests.rule_51_test (4 ms)
[ RUN      ] rule_tests.rule_54_test
[       OK ] rule_tests.rule_54_test (4 ms)
[ RUN      ] rule_tests.rule_55_test
[       OK ] rule_tests.rule_55_test (3 ms)
[ RUN      ] rule_tests.rule_56_test
[       OK ] rule_tests.rule_56_test (3 ms)
[----------] 17 tests from rule_tests (80 ms total)

[----------] Global test environment tear-down
[==========] 100 tests from 4 test suites ran. (721 ms total)
[  PASSED  ] 100 tests.

D:\Users\wolfreiser\Documents\karas6\build\Debug\karas_tests.exe (process 10264) exited with code 0.
Press any key to close this window . . .
```

MeKu Software

DataFlow/Codegen try1: LLVM

LLVM

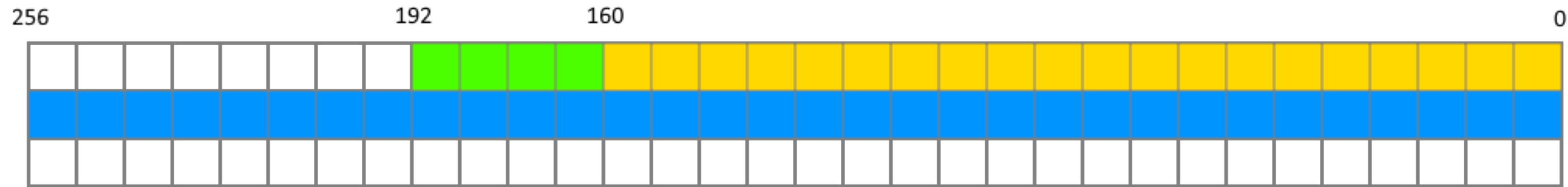
- Позволит заиспользовать уже имеющуюся архитектуру
- Уже есть кодогенератор – EVM-LLVM
- Отлаженый и известный проект
- Хорошая документация
  - Туториал по базовым вещам: https://llvm.org/docs/tutorial/
  - Полная информация об IR: https://llvm.org/docs/LangRef.html
  - Doxygen: https://llvm.org/doxygen/
- Множество остальных приемуществ :)

LLVM

- Не очень совместим с Solidity
- Несколько адресных пространств со своими особенностями
  - Memory
  - Storage
  - Calldata
  - Code
  - Stack

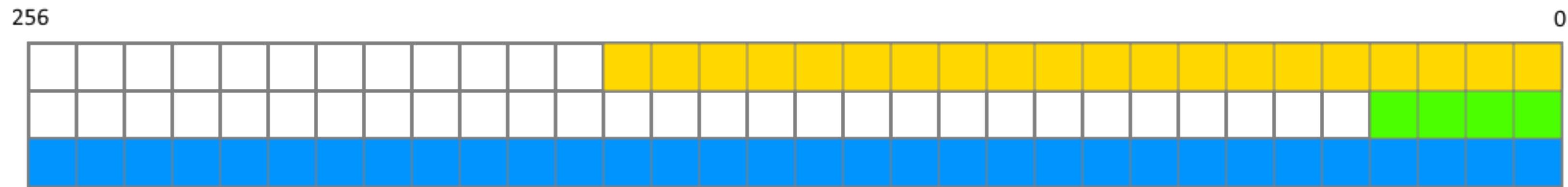
Storage slots

Упаковывается по 256-битным слотам без разделений



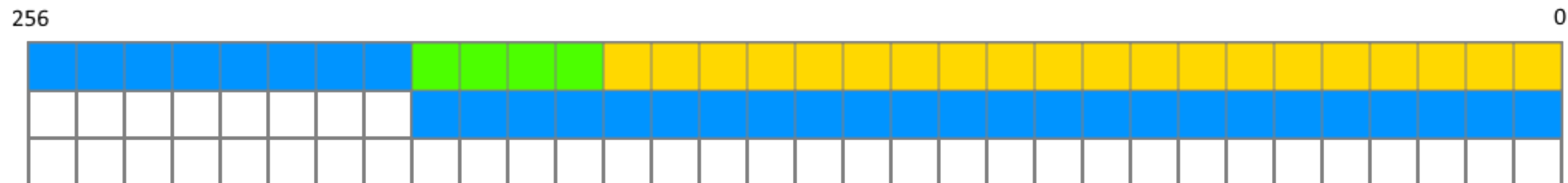
Memory

Может иметь любую структуру



Calldata

Тесно упаковано согласно спецификации ABI

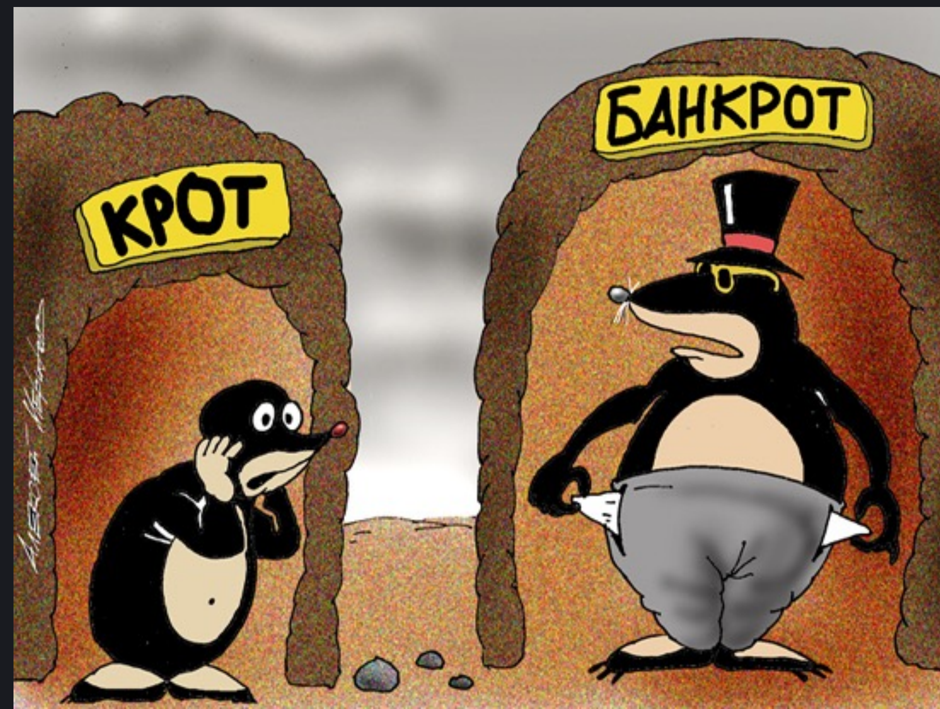


LLVM

- Отсутствие возможности расширения
  - Нельзя добавить ссылку на дерево AST
- После оптимизаций IR информация о позиции может быть утеряна
- Повышается сложность в сборке проекта
  - У разработчика требуется иметь установленный EVM-LLVM в систему

LLVM

- Отсутствие
  - Нельзя до
- После опти ции может быть утеряна
- Повышается
  - У разработ ный EVM-LLVM в систему



MeKu Software

DataFlow/Codegen try2: Own

Собственный DataFlow

- Простота в расширении и модификации
- Учет особенностей EVM и Solidity

- Требуется реализация инфраструктуры
  - Управление памятью
  - Структуры данных (intrusive list, directed graph)
  - Алгоритмы (Dominance, SSA)
- Все еще WIP

MeKu Software

SSA

Control Flow Graph

- Весь код, который содержит ветвления, можно представить в виде ориентированного графа, который называется графом потока управления (Control Flow Graph, CFG)
- Узлы в таком графе называются базовыми блоками. Базовые блоки содержат в себе код без ветвлений
- Ребра в графе – это переход из одного блока в другой. Чаще всего ребра два – для true и false ветки. Но может быть и больше, если это switch.
- Если из узла выходит одно ребро, то это линейный код – базовые блоки в этом случае можно объединить.

Static Single Assignment Form

- SSA – это промежуточное представление кода, в котором переменной присваивается значение только один раз
- Если в исходном коде одной и той же переменной значение присваивается во второй раз, то эта перменная переименовывается, чаще всего к ней добавляется индекс

Phi-функция

- Если же присваивание происходит внутри двух базовых блоков, которые затем переходят в один, то в этом базовом блоке проставляется Phi-функция:

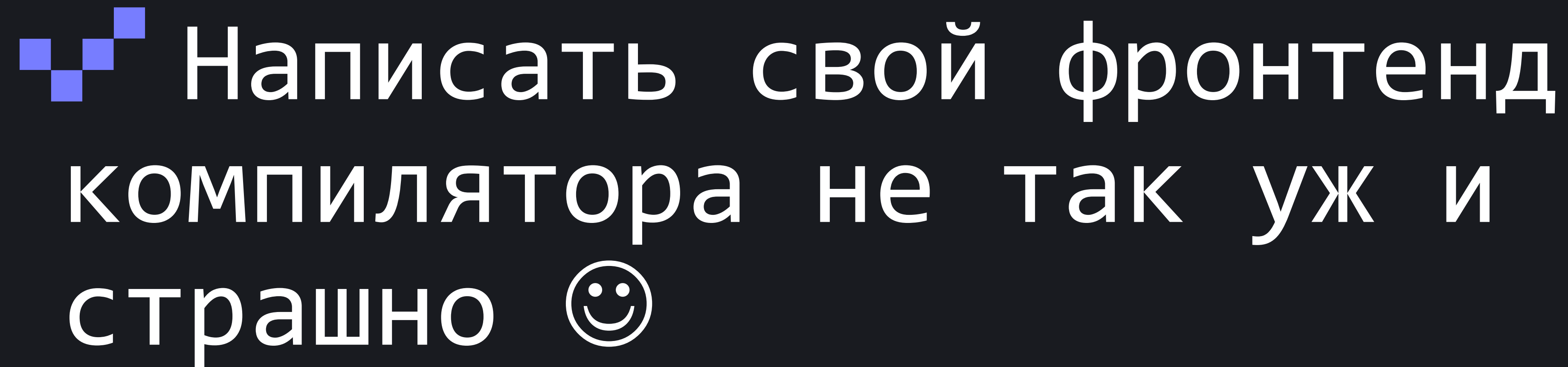
```
Y1 = 0
if (condition)
  Y2 = 1
else
  Y3 = 2
Y4 = Phi(Y2, Y3)
```

- Phi-функция позволяет понять, из какого базового блока пришла переменная

Ок, и?

- Такая форма позволяет легко реализовать Dead Code Elimination:
  - Удалить базовые блоки под константными условиями
  - Удалить код после инструкций прерывания (return, throw, ...)
  - Удалить переменные, которые не были использованы при вычислении (dead store)

Вывод



Q&A