Компилятор OLang

PriPlusTeam:
Арефьев Владислав
Овчинников Егор

Постановка задачи

Цель:

Разработать компилятор для объектно ориентированного языка.

Особенности языка:

- Базовые конструкции (литералы, условные операторы, циклы и прочее)
- Наследование
- Полиморфизм
- Структуры данных (массив, список)
- Шаблоны
- Класс может иметь члены в виде полей и методов

Инструменты для выполнения цели: **C++, LLVM-IR**

Пример кода на языке OLang

```
class Point is
  var x : Integer
  var y : Integer
  var z : Integer

  this() is
      this.init()
  end

  method init() is
      this.x := 1
      this.y := 2
      this.z := 3
  end

  method print() is
      printf("Point[x=%d, y=%d, z=%d]\n", this.x.get(), this.y.get(), this.z.get())
  end
end
```

Технологии

При разработке компилятора были реализованы:

- лексический анализатор;
- синтаксические анализатор;
- несколько итераций семантического анализа;
- кодогенерация с помощью *LLVM IR*.

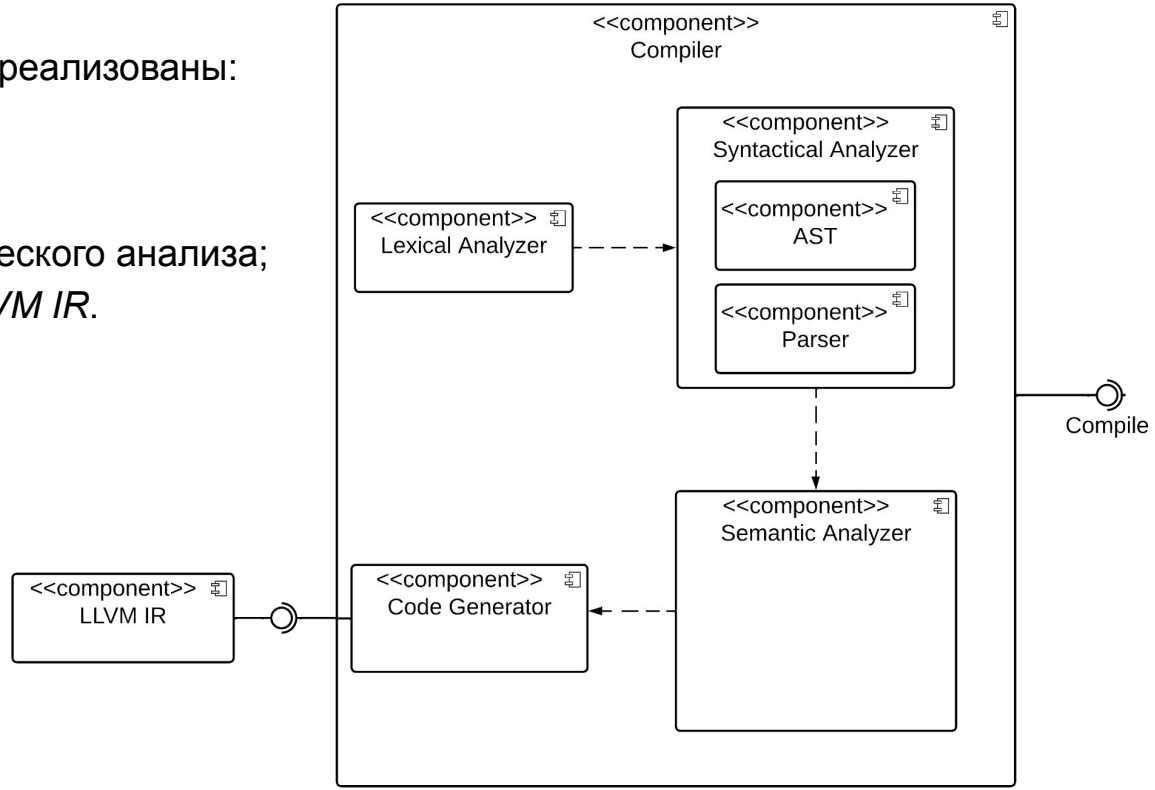
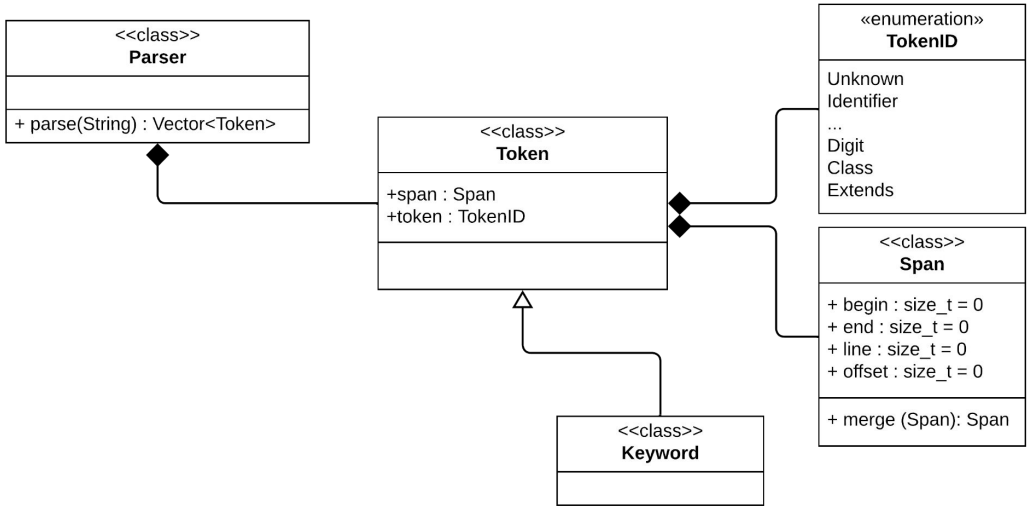


Диаграмма классов: лексический анализатор

Особенности лексического парсера:

- Последовательный разбор текста программы на токены, в соответствии предоставленной грамматикой
- Разбор происходит с помощью машины состояний сразу для всего текста, результат передается на последующий этап.

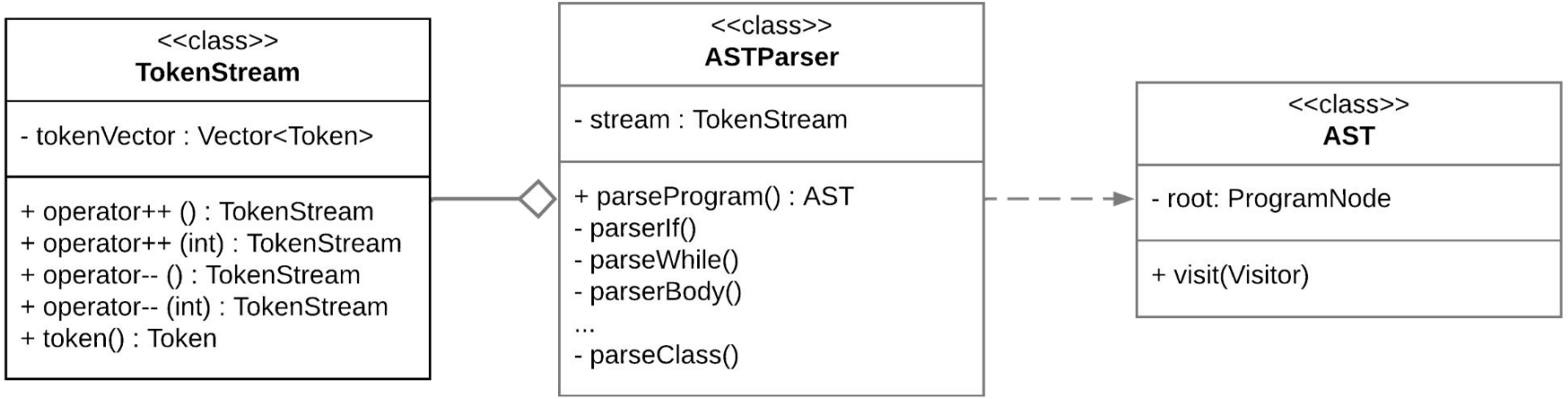


Синтаксический анализатор

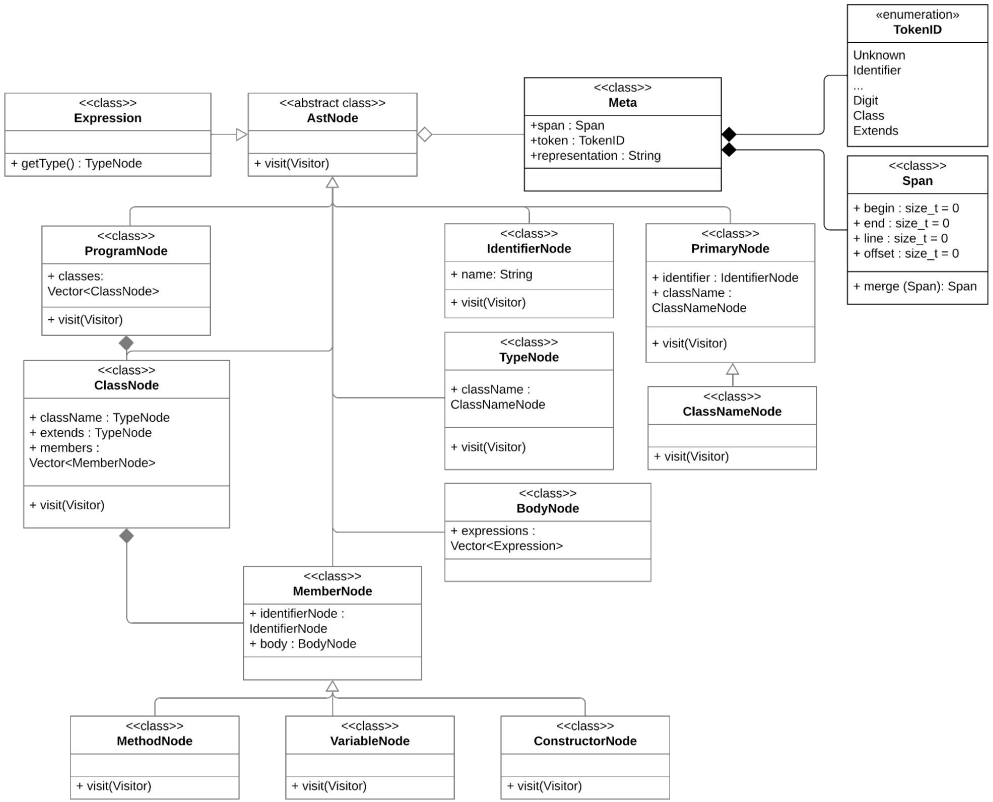
На этапе синтаксического анализа строится AST, узлы которого отвечают за определенные синтаксические конструкции. Все узлы наследуются от базового узла — **ast_node**, который обладает чисто виртуальными методами **visit, print, clone,** а также объектом хранящим информацию о положении узла в тексте.

Для построения используется нисходящий рекурсивный подход. В зависимости от последовательностей токенов, полученных на предыдущем этапе, создаются соответствующие узлы согласно грамматике и добавляются в дерево

Синтаксический анализатор



Синтаксический анализатор



Семантический анализ

Для выполнения семантического анализа использовались визитеры. Можно выделить два основных этапа семантического анализа:

- Scope checking – анализ областей видимости для классов, методов, членов класса и локальных переменных
- Type checking – проверка типов (возвращаемый тип сходится с заявленным в сигнатуре, при присваивании типы совпадают или их можно скастить, в условных выражениях используемый тип Boolean и т.д.)

Также во время основных этапов перестраивалось AST:

- Инстанцирование шаблонных классов, при нахождении инстанцирования поддерево шаблона клонируется и добавляется в программу, при этом тип переменной шаблона меняется на инстанцированный
- Генерация поддеревьев для цепочек вызовов в выражениях

Пример семантического анализа

```
class scope : public std::enable_shared_from_this<scope> {
 private:
  std::shared_ptr<scope> parent_;
  std::shared_ptr<scope_symbol> generic_symbol_{ p: new scope_symbol};
  std::shared_ptr<scope_symbol> symbol_{ p: new scope_symbol};
  std::string name_;
  scope_type type_{scope_type::Class};

 public:
  scope() = default;

  explicit scope(std::shared_ptr<scope> parent, std::string name = "",
                 scope_type type = scope_type::Class)
      : parent_{std::move(parent)}, name_{std::move(name)}, type_{type} {}

  explicit scope(std::shared_ptr<scope_symbol> symbol, std::string name = "",
                 scope_type type = scope_type::Class)
      : parent_{},
        symbol_{std::move(symbol)},
        name_{std::move(name)},
        type_{type} {}

  scope(std::shared_ptr<scope> parent, std::shared_ptr<scope_symbol> symbol,
        std::string name = "", scope_type type = scope_type::Class)
      : parent_{std::move(parent)},
        symbol_{std::move(symbol)},
        name_{std::move(name)},
        type_{type} {}

  std::shared_ptr<scope> push(const std::string& name = "",
                              scope_type type = scope_type::Class);

  std::shared_ptr<scope> push(const std::shared_ptr<scope_symbol>& symbols,
                              const std::string& name = "",
                              scope_type type = scope_type::Class);
```

```
std::shared_ptr<expression_ext> expression_node::constr_call_checking(
    std::shared_ptr<ast_node> constr_call, std::shared_ptr<class_node> clazz,
    error_handling::error_handling& error_handler) {
  if (!expression_values.empty() && expression_values[0].first) {
    error_handler.register_error( err: error_handling::make_error_t(
        node: *constr_call,  msg: "error: invalid constructor call"));
    return EMPTY_VAR;
  }

  std::shared_ptr<constructor_node> ctor;
  std::vector<std::shared_ptr<expression_ext>> args = {};
  if (expression_values.empty()) {
    ctor = clazz->find_ctr( &: error_handler);
  } else {
    ctor = clazz->find_ctr( args: expression_values[0].second,  &: error_handler);
    args = get_args_objects( arguments: expression_values[0].second->get_arguments(),
                             &: error_handler);
  }

  if (!ctor) {
    error_handler.register_error( err: error_handling::make_error_t(
        node: *constr_call,  msg: "error: cannot find constructor"));
    return EMPTY_VAR;
  }

  auto ctor_call :shared_ptr<...>  =
      std::make_shared<constructor_call>(clazz,  constr: ctor, args,  type: clazz->get_type());
  return ctor_call;
}
```

Пример семантического анализа

```
class type_visitor : public semantic_visitor {
 private:
  using type = details::type_node;

  std::unordered_map<std::string, std::unordered_map<std::string, bool>>
      type_casting_ = {
          { x: type::intT,  y: {{ x: type::realT,  y: true}, { x: type::RealT,  y: true}}},
          { x: type::realT,  y: {{ x: type::intT,  y: true}, { x: type::IntegerT,  y: true}}},
          { x: type::IntegerT,
            y: {{ x: type::intT,  y: true}, { x: type::RealT,  y: true}, { x: type::AnyT,  y: true}}},
          { x: type::RealT,
            y: {{ x: type::realT,  y: true}, { x: type::IntegerT,  y: true}, { x: type::AnyT,  y: true}}},
      };
  std::unordered_set<std::string> types_ = {type::IntegerT, type::RealT,
                                            type::BooleanT, type::AnyT};
  std::unordered_map<std::string, std::string> constructor_calls_;
  error_handling::error_handling error_;

  void register_error(const details::ast_node& node, const std::string& msg) {
    error_.register_error( err: error_handling::make_error_t(node, msg));
  }
```

Кодогенерация

Кодогенерация также происходит с помощью визитеров. Для генерации промежуточного представления используется LLVM. Благодаря ему происходит точечная генерируются инструкций в модуле.

Основные этапы генерации:

- Регистрация глобальных функций
- Генерация типов
- Регистрация членов классов
- Добавление стандартной библиотеки
- Генерация виртуальных таблиц
- Генерация определений функций

Особенности генерации:

- Создание объектов происходит с помощью malloc
- Методы стандартной библиотеки написаны на C
- Точка входа любой класс с методом main
- Что-то про наследование…

Кодогенерация: типы

```
%__VTableBase = type { void (%Base*)* }
%Base = type { %__VTableBase* }
%__VTableInheritor = type { void (%Inheritor*)* }
%Inheritor = type { %__VTableInheritor* }
%__VTableTester = type { void (%Tester*, %Base*)* }
%Tester = type { %__VTableTester* }


@__VTableBase = global %__VTableBase { void (%Base*)* @print }
@__VTableInheritor = global %__VTableInheritor { void (%Inheritor*)* @print.44 }
```

Кодогенерация: функции

```
define %Integer* @"<init>.1"(%Integer* %0, %Integer* %i) {
entry:
  %1 = getelementptr inbounds %Integer, %Integer* %0, i32 0, i32 1
  %2 = getelementptr inbounds %Integer, %Integer* %i, i32 0, i32 1
  %3 = load i32, i32* %2, align 4
  store i32 %3, i32* %1, align 4
  ret %Integer* %0
}
```

Кодогенерация: объявления методов

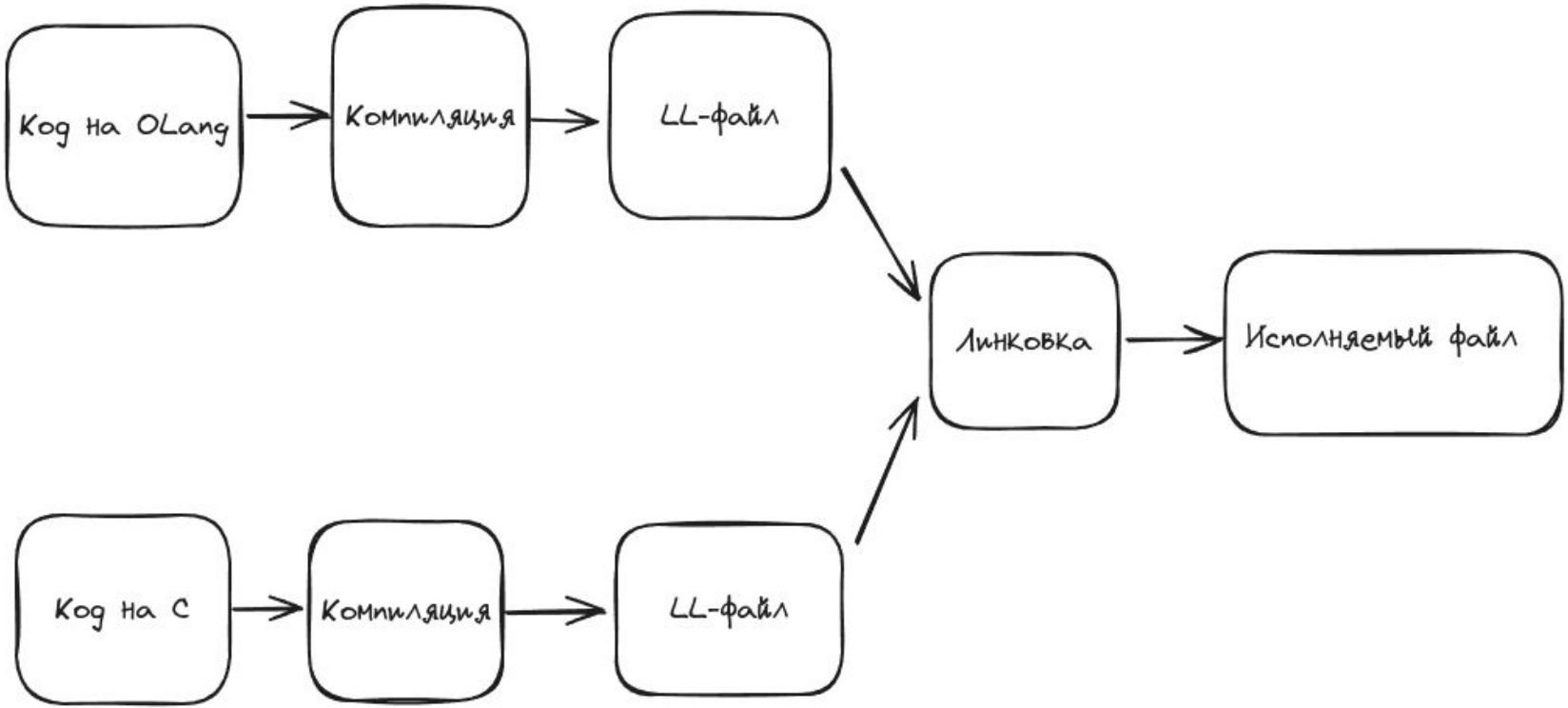
```
declare %Integer* @UnaryMinusI(%Integer*)

declare %Real* @toRealI(%Integer*)

declare %Integer* @PlusII(%Integer*, %Integer*)

declare %Real* @PlusIR(%Integer*, %Real*)
```

Кодогенерация: использование C



Результаты

Что было реализовано:

- Генерация последовательных объявлений классов, с членами и методами, основными выражениями в виде return, if, while, assingment
- Реализовано наследование и полиморфизм
- Генерация шаблонов
- Создана стандартная библиотека
- Вызов базового конструктора
- Добавлен printf

На данный момент отсутствует:

- Структуры данных (списки и массивы). Есть массив Integer-ей
- Кастинг типов
- Отсутствует null, в связи с этим невозможно создать списки
- Присутствуют незначительные баги