


# Лекция 4. Объектно-ориентированное программирование

Коновалов А. В.

10 апреля 2024 г.

## Рекомендуемая литература

1. Стефан К. Дьюхэрст. Скользкие места C++. Как избежать проблем при проектировании и компиляции ваших программ. — М: ДМК Пресс, 2006. — 264 с.: ил.  
 См. советы 78, 79, 35, 53, 70 (именно в этом порядке).
2. Piumarta, I., Warth, A. (2008). Open, Extensible Object Models. In: Hirschfeld, R., Rose, K. (eds) Self-Sustaining Systems. S3 2008. Lecture Notes in Computer Science, vol 5146. Springer, Berlin, Heidelberg. [https://doi.org/10.1007/978-3-540-89275-5\\_1](https://doi.org/10.1007/978-3-540-89275-5_1)
3. Вирт Н., Гуткнехт Ю. Разработка операционной системы и компилятора. Проект Оберон: Пер. с англ. Борисов Е.В., Чернышов Л.Н. — М.: ДМК Пресс, 2012. — 560 с.: ил.
4. Axel-Tobias Schreiner. Object-Oriented Programming with ANSI-C — Hanser, 2011. — 223 p. ISBN 3-446-17426-5, скачать её можно с сайта автора: <https://www.cs.rit.edu/~ats/books/>

++ПЯИУН

# Объектно-ориентированное расширение НУИЯПа

В этом разделе будет рассмотрено расширение НУИЯПа в духе C++. В программах можно будет определять классы с виртуальными функциями.

```
t.Definition ::= ... | t.Class  
t.Statement ::= ... | t.CallConstructor  
t.Expr ::= ... | t.MCall
```

Определения пополняются определениями классов, операторы — оператором вызова конструктора, выражения — вызовом метода (виртуальной функции).

Рассматривать статические и неvirtуальные методы мы не будем, т.к. они, фактически, являются синтаксическим сахаром для обычных глобальных функций.

## Определение классов

```
t.Class ::=  
  (class s.Name (e.Base)  
    (fields (s.Name t.ConstExpr)*)?  
  
    (method s.Name (s.Name+) t.LocalVars? e.Code)*  
  )
```

```
e.Base ::= s.Name*
```

В определении класса указывается его имя, список имён базовых классов, объявление полей и перечень методов класса. В общем случае, синтаксис допускает множественное наследование.

Методы принимают не менее одного параметра — первым параметром (как и в Python) передаётся указатель на экземпляр, для которого метод был вызван.

## Вызов конструкторов и методов

```
t.CallConstructor ::= (init t.ObjectPtr s.Name)  
t.MCall ::= (mcall t.ObjectPtr s.Name t.Expr*)  
t.ObjectPtr ::= t.Expr
```

Здесь для наглядности введён синоним `t.ObjectPtr` для `t.Expr`.  
`t.ObjectPtr` — это просто выражение, значением которого является адрес переменной, содержащей экземпляр класса (в том числе, и адрес поля структуры или другого класса, адрес элемента массива и т.д.).

Конструктор принимает адрес объекта и имя класса и инициализирует его таблицу виртуальных функций.

Вызов метода принимает адрес объекта, имя метода и список аргументов.

## Семантические ограничения

Точно также, как и поля структур, поля классов являются целочисленными константами, определяющими смещения соответствующих значений в объекте, и тоже они располагаются в глобальном пространстве имён.

Имена методов также располагаются в глобальном пространстве имён (позже увидим, что они суть смещения в таблицах виртуальных функций).

Классы, не связанные отношением наследования, не могут содержать одноимённых методов. Класс-потомок может иметь методы с теми же именами, что и у предка, в этом случае метод будет переопределён. Переопределённый метод должен иметь то же количество параметров.

Имена, содержащие два прочерка подряд \_\_, считаются зарезервированными. В пользовательских идентификаторах во избежание конфликта имён нельзя записывать два прочерка подряд.

## Реализация объектно-ориентированного расширения

Для каждого класса создаётся таблица виртуальных функций — структура, содержащая адреса методов для каждого из классов.

Каждый класс отображается на структуру, первым полем которой является указатель на соответствующую виртуальную таблицу (иногда для краткости мы будем пользоваться этим жаргонизмом).

Конструктор просто присваивает это поле.

Имена методов — это смещения адресов методов в виртуальной таблице.

Вызов метода, очевидно, косвенный. Из объекта извлекается указатель на таблицу виртуальных функций, к ней прибавляется смещение метода, реальный адрес читается по данному смещению.



## Компиляция класса без предка (1)

```
(class <имя класса> ()  
  (fields <поля>)  
  (method <имя метода> (this <параметры>) <тело>)...  
)
```

Для класса создаются две структуры. Имя первой структуры совпадает с именем класса и имеет вид

```
(struct <имя класса>  
  ("-" 1)  
  <поля>  
)
```

Первое поле структуры безымянное — в нём располагается таблица виртуальных функций.

## Компиляция класса без предка (2)

```
(class <имя класса> ()  
  (fields <поля>)  
  (method <имя метода> (this <параметры>) <тело>)...  
)
```

Вторая структура — таблица виртуальных функций с именем <имя класса>\_class\_\_, содержит по одному полю на каждый метод класса:

```
(struct <имя класса>_class__  
  (<имя метода> 1)...  
)
```

## Компиляция класса без предка (3)

```
(class <имя класса> ()  
  (fields <поля>)  
  (method <имя метода> (this <параметры>) <тело>)...  
)
```

Каждый метод компилируется в функцию с именем <имя класса>\_\_<имя метода>:

```
(function <имя класса>__<имя метода> (this <параметры>)  
  <тело>  
)  
...
```

## Компиляция класса без предка (4)

```
(class <имя класса> ()  
  (fields <поля>)  
  (method <имя метода> (this <параметры>) <тело>)...  
)
```

И, наконец, создаётся экземпляр виртуальной таблицы с именем `<имя класса>_vtbl__`, содержащий указатели на методы класса:

```
(var <имя класса>_vtbl__ <имя класса>_class__ "="  
  (<имя класса>__<имя метода> 1)...  
)
```

## Компиляция класса-наследника (1)

```
(class <имя класса> (<имя предка>)  
  (fields <поля потомка>)  
  (method <имя метода> (this <параметры>) <тело>)...  
)
```

Структура, создаваемая для класса, содержит первое  
безымянное поле размером с предка:

```
(struct <имя класса>  
  ("-" <имя предка>)  
  <поля потомка>  
)
```

## Компиляция класса-наследника (2)

```
(class <имя класса> (<имя предка>)  
  (fields <поля потомка>)  
  (method <имя метода> (this <параметры>) <тело>)...  
)
```

Методы, определённые в потомке, делятся на две группы: переопределённые методы базового класса и новые методы класса-потомка. Виртуальная таблица потомка начинается с виртуальной таблицы предка, после которой следуют новые методы потомка:

```
(struct <имя класса>_class__  
  ("-" <имя предка>_class__)  
  <имя нового метода>...  
)
```

## Компиляция класса-наследника (3)

Компиляция методов ничем не отличается. А вот таблица генерируется интересно.

```
(var <имя класса>_vtbl__ <имя класса>_class__ "="  
  ...  
  /* унаследованный метод */  
  <имя предка>__<имя метода>  
  ...  
  /* переопределённый метод */  
  <имя класса>__<имя метода>  
  ...  
  /* новый метод */  
  <имя класса>__<имя метода>  
  ...  
)
```

## Компиляция класса-наследника (4)

Первые `<имя предка>_class__` слов в таблице `<имя класса>_vtbl__` соответствуют методам базового класса — как унаследованным, так и переопределённым.

Для унаследованного метода записывается имя его реализации из базового класса `<имя предка>__<имя метода>`, для переопределённого — имя реализации в классе-потомке `<имя класса>__<имя метода>`. Порядок методов, очевидно, тот же, что и в описании структуры `<имя предка>_class__`.

Последующие слова соответствуют новым методам класса-потомка. Их порядок тот же, что и в описании структуры `<имя класса>_class__`.



## Компиляция конструктора

```
(init t.Address <имя класса>)
```

может быть легко переписан в эквивалентный код на НУИЯПе

```
(t.Address "=" <имя класса>_vtbl__)
```

т.е. фактически является синтаксическим сахаром.

## Компиляция вызовов методов

На первый взгляд, компиляцию вызова метода тоже можно рассматривать как синтаксический сахар: выражение

```
(mcall t.Object s.Method e.Args)
```

переписывается в

```
(call (L ((L t.Object) "+" s.Method)) t.Object e.Args)
```

## Компиляция вызовов методов

На первый взгляд, компиляцию вызова метода тоже можно рассматривать как синтаксический сахар: выражение

```
(mcall t.Object s.Method e.Args)
```

переписывается в

```
(call (L ((L t.Object) "+" s.Method)) t.Object e.Args)
```

Однако, если вызов метода реализовать именно так, возникнет проблема: выражение `t.Object` будет вычисляться дважды.

Поэтому вызов метода придётся реализовывать как примитив языка.

Проверка типа во время выполнения

## Проверка типа во время выполнения (1)

На практике распространена ситуация, когда мы, имея указатель на базовый класс, хотели бы узнать — не указывает ли он на самом деле на конкретного потомка класса?

Для проверки точного соответствия можно просто сравнить на равенство указатели на виртуальные таблицы.

Но, в соответствии с принципом подстановки Барбары Лисков, этого недостаточно. Нужно уметь отвечать на вопрос:  
*«указывает ли на самом деле указатель на базовый класс на конкретного потомка или любого потомка этого потомка?»*

## Проверка типа во время выполнения (2)

Существует несколько вариантов решения этой проблемы.

Можно в первом слове виртуальной таблицы хранить ссылку на виртуальную таблицу предка (соответственно, 0 для класса без предка). Тогда, двигаясь по этой цепочке, мы или найдём указатель на искомую таблицу, или дойдём до конца цепочки.

Преимущество: глубина наследования не ограничена.

Недостаток: линейное время проверки.

## Проверка типа во время выполнения (3)

Никлаус Вирт в [3] использовал элегантный приём: он ограничил глубину наследования некоторой константой (обозначим её  $MAXD$ ) и список заменил на массив. В начале каждой виртуальной таблицы резервируется массив на  $MAXD$  слов.

Будем говорить, что глубина в иерархии классов без предков равна 0, их потомков — 1, потомков их потомков — 2 и т.д.

Тогда для класса глубины  $D$  в  $D$ -м элементе массива будет находиться ссылка на свою собственную виртуальную таблицу, в  $(D - 1)$ -м — ссылка на таблицу его предка и т.д. Элементы массива, бóльшие  $D$ , заполняются нулями.

## Проверка типа во время выполнения (4)

Таким образом, для некоторого класса  $C$  глубины  $d_C$  для него самого и для всех его потомков в ячейке с индексом  $d_C$  будет ссылка на его виртуальную таблицу.

Пусть, мы хотим проверить, является ли объект, доступный по указателю, экземпляром класса  $C$ . Для этого достаточно просто сравнить содержимое ячейки массива  $d_C$  с указателем на виртуальную таблицу этого класса.

Проверка будет осуществлена за константное время.

Немного усложнив схему, можно сохранить постоянное время (сделав его, правда, чуть больше) и при этом допустить неограниченную глубину иерархии наследования. Подумайте над этим самостоятельно.



## Множественное наследование

# Множественное наследование

Рассмотренный синтаксис определения классов допускает указание нескольких базовых классов — у класса может быть несколько предков, состояние (поля) и поведение (методы) которых он может наследовать. Разумеется, потомок может расширять состояние — добавлять поля и изменять поведение — переопределять методы у любого из своих предков.

Ограничение, что классы, не связанные иерархией наследования, не могут иметь одноимённых методов, сохраняется. Как следствие, потомок может унаследовать от двух разных предков одноимённые методы только в случае, если сами предки являются потомками одного и того же класса.

## Размещение в памяти (1)

В случае одиночного наследования экземпляр наследника включает экземпляр базового класса. Экземпляр базового класса располагается в наследнике по смещению 0, благодаря чему указатель на наследник можно рассматривать и как указатель на базовый класс.

В случае множественного наследования экземпляр потомка будет содержать несколько экземпляров предков и только один из них сможет располагаться по смещению 0. Будем считать, что экземпляр наследника начинается с экземпляров базовых классов в том порядке, в каком они перечислены в `e.Base`.

## Размещение в памяти (2)

По смещению 0 будет располагаться только один из базовых классов (первый), для обращения к остальным нужно выполнить *восходящее преобразование* — преобразовать указатель на потомок в указатель на один из непосредственных предков.

Расширим синтаксис выражений операцией восходящего преобразования:

```
t.Expr ::= ...  
      | (upcast t.Expr from s.Name to s.BaseName)
```

Фактически, преобразование указателя будет сводиться к прибавлению смещения на базовый класс.

## Размещение в памяти (3)

Обращаться к полям и методам, унаследованным от первого базового класса, можно непосредственно, а полям и методам остальных предков — только после восходящего преобразования.

Языки высокого уровня подобное преобразование вставляют неявно, учитывая типы на фазе семантического анализа. НУИЯП++ рассматривается как промежуточное представление программы, полученное на выходе стадии анализа и передаваемое на вход стадии синтеза, т.е. эти преобразования в программу должны быть добавлены явно.

## Таблицы виртуальных функций (1)

Экземпляр потомка содержит несколько подобъектов для экземпляров базовых классов, соответственно, он будет содержать несколько указателей на таблицы виртуальных функций. Первая из них будет доступна непосредственно по смещению 0, для доступа к остальным нужно выполнить восходящее преобразование.

Кроме того, объекты-предки сами могут иметь несколько предков, а значит, и сами они могут иметь несколько виртуальных таблиц.

## Таблицы виртуальных функций (2)

Случай, когда потомок переопределяет метод первого базового класса, ничем не отличается от случая одиночного наследования.

Интересное начинается, когда переопределяется метод не первого базового класса. В момент вызова метода указатель на объект ссылается на начало экземпляра предка, которое не совпадает с началом самого объекта. А код метода пишется в предположении, что первый параметр («this» или «self») ссылается на начало потомка.

## Таблицы виртуальных функций (3)

При вызове виртуальной функции потребуется корректировать указатель на объект, вычитая из него смещение базового класса в производном.

Смещение разумно хранить в виртуальной таблице вместе с указателем на виртуальную функцию.

```
(struct VTableEntry  
    (VTableEntry_function ptr)  
    (VTableEntry_offset int)  
)
```



## Реализация множественного наследования (1)

```
(class <потомок> (<предок1> ... <предокN>) (<поля>) <методы>)
```

Структура для класса содержит поля для каждого из предков.  
Обратим внимание на то, что эти поля теперь именованные:

```
(struct <потомок>  
  (<потомок>__<предок1> <предок1>)  
  ...  
  (<потомок>__<предокN> <предокN>)  
  <поля>  
)
```

## Реализация множественного наследования (2)

```
(class <потомок> (<предок1> ... <предокN>) (<поля>) <методы>)
```

Новыми методами расширяется только виртуальная таблица первого предка:

```
(struct <потомок>_class__  
    ("-" <предок1>_class__)  
    (<имя нового метода> VTableEntry)...  
)
```

## Реализация множественного наследования (3)

Для класса генерируется несколько виртуальных таблиц по количеству указателей на виртуальные таблицы в экземпляре (их может быть больше, чем непосредственных предков, если сами предки имеют несколько предков). Поскольку виртуальных таблиц для одного класса несколько, в имя таблицы добавляется и имя предка.

Для первого предка смещения по определению равны нулю:

```
(var <потомок>__<предок1>_vtbl__ <потомок>_class__ "="  
  ...  
  <предок1>__<имя метода> 0  
  ...  
  <потомок>__<имя метода> 0  
)
```

## Реализация множественного наследования (4)

Для других предков смещения для переопределённых функций равны смещению экземпляра предка в экземпляре потомка:

```
(var <потомок>__<предокi>_vtbl__ <предокi>_class__ "="  
  ...  
  /* унаследованный метод */  
  <предокi>__<имя метода> 0  
  ...  
  /* переопределённый метод */  
  <потомок>__<имя метода> <потомок>__<предокi>  
)
```

## Реализация множественного наследования (5)

В случае косвенных предков смещения надо складывать:

```
(var <потомок>__<прапредокj>_vtbl__ <прапредокj>_class__ "="  
  ...  
  /* унаследованный метод */  
  <прапредокj>__<имя метода> 0  
  ...  
  /* переопределённый метод */  
  <потомок>__<имя метода>  
    (<потомок>__<предокi> "+" <предокi>__<прапредокj>)  
)
```

## Компиляция конструкций

- ▶ Конструктор (`init t.ObjectPtr s.Name`) теперь должен инициализировать все виртуальные таблицы класса.
- ▶ Операция вызова метода (`mcall t.ObjectPtr s.Method t.Expr`) должна из виртуальной таблицы извлекать смещение и вычитать его из указателя на объект при вызове метода
- ▶ Операция восходящего приведения типа  
(`upcast t.ObjectPtr from <потомок> to <предокі>`)  
просто прибавляет смещение  
(`t.ObjectPtr "+" <потомок>__<предокі>`)

## Виртуальное наследование (1)

При рассмотренном множественном наследовании экземпляры предков размножаются:

```
(class A () ...)  
(class B1 (A) ...)  
(class B2 (A) ...)  
(class C (B1 B2) ...)
```

в экземпляре класса C будет находиться два экземпляра класса A (внутри B1 и B2 соответственно).

Виртуальное наследование — механизм C++, позволяющий избежать размножения экземпляров предков.

## Виртуальное наследование (2)

Рассмотрим вариант НУИЯП++В, в котором, для простоты, всё наследование виртуальное.



Если Вы видите этот слайд, значит, я не успел написать этот раздел. Читайте рекомендованные параграфы в [1]. 😊

## Прототипное ООП

## Прототипное ООП — НУИЯП-П

Прототипное программирование — разновидность ООП, в которой отсутствует понятие класса. В прототипном ООП для осуществления повторного использования вместо наследования класса используется операция *клонирования* объекта.

Объект-клон наследует все поля и методы своего *прототипа*.

Мы рассмотрим подход, когда объект-клон сохраняет связь со своим прототипом и делегирует к нему обращения к отсутствующим у себя полям и методам.

Мы будем рассматривать объекты как ассоциативные массивы, отображающие *символы* на слова (т.е. числа или адреса). Словарную пару из символа и слова мы будем называть *слотом*.

Если в слоте находится указатель на функцию, то этот слот мы можем считать методом данного объекта.

## Синтаксис НУИЯПа-П (1)

```
t.Definition ::= ... | (symbol s.Name)
t.Statement ::= ...
    | (clone t.Address t.ObjectPtr)
    | (p-set t.ObjectPtr s.Name "=" t.Expr)
t.Expr ::= ...
    | (p-get t.ObjectPtr s.Name)
    | (p-mcall t.ObjectPtr s.Name t.Expr*)
```

Кроме того, определены константы PObject и MAX\_SLOTS.

## Синтаксис НУИЯПа-П (2)

Конструкция `(symbol s.Name)` определяет символ, который можно использовать в качестве имени слота.

Оператор `clone` инициализирует переменную по адресу `t.Address`, используя в качестве прототипа `t.ObjectPtr` (последний может быть нулём).

Оператор `p-set` и операции `p-get`, и `p-call` служат, соответственно, для записи и чтения слотов и вызова метода.

Константа `PObject` содержит размер объекта и используется для описания переменных, полей структур и т.д. Константа `MAX_SLOTS` содержит максимальное количество слотов, которые могут быть в объекте — таким образом мы избегаем динамического выделения памяти.

## Семантика НУИЯПа-П (1) — символы

Семантику НУИЯПа-П можно определить путём трансформации программы в эквивалентную программу на базовом НУИЯПе.

Символы — это некоторые уникальные значения, не равные друг другу. Проще всего это обеспечить, создавая для каждого символа переменную (ненулевого размера) — символом будет адрес этой переменной.

К примеру, `(symbol s.Name)` можно транслировать в `(var s.Name 1)`. Значение этой переменной не важно, т.к. использоваться не будет.

Для отладочных целей можно в переменной хранить имя этого символа:

```
(symbol <имя символа>)
```

↓

```
(var <имя символа> "=" '<имя символа>' 0)
```

## Семантика НУИЯПа-П (2) — слоты и объекты

Объекты должны быть ассоциативными массивами, отображающими символы на значения.

Промышленная реализация тут могла бы использовать какую-либо эффективную структуру данных, например, хэш-таблицу, однако, мы ради простоты будем использовать простые массивы пар с линейным поиском в них:

```
(const Symbol "=" 1)
(const MAX_SLOTS "=" 42)
```

```
(struct Slot
  (Slot_symbol Symbol)
  (Slot_value 1)
)
```

```
(struct PObject ("- (Slot "*" MAX_SLOTS)))
```

## Семантика НУИЯПа-П (3) — клонирование (1)

У каждого объекта есть слот с зарезервированным именем `proto__`, содержащий указатель на прототип. Оператор клонирования устанавливает этот слот и зануляет все остальные слоты переменной:

```
(clone t.Address t.ObjectPtr)
```

компилируется в

```
(call clone__ t.Address t.ObjectPtr)
```

где вызывается функция `clone__` библиотеки поддержки времени выполнения.



## Семантика НУИЯПа-П (4) — клонирование (2)

Реализация функции clone\_\_:

```
(symbol proto__)
```

```
(function clone__ (new proto)
  (var (i int))
  (((L new) "+" Slot_symbol) "=" proto__)
  (((L new) "+" Slot_value) "=" (L proto))
  (i "=" 1)
  (while ((L i) "<" MAX_SLOTS)
    (((((L new) "+" ((L i) "*" Slot)) "+" Slot_symbol) "=" 0)
      (i "=" ((L i) "+" 1))
    )
  )
)
```

## Семантика НУИЯПа-П (5) — чтение и запись слотов (1)

p-get и p-set также компилируются в вызов функции библиотеки времени выполнения:

```
(p-get t.ObjectPtr s.Name)
```

↓

```
(call pget__ t.ObjectPtr s.Name)
```

```
(p-set t.ObjectPtr s.Name "=" t.Expr)
```

↓

```
(call pset__ t.ObjectPtr s.Name t.Expr)
```

## Семантика НУИЯПа-П (6) — чтение и запись слотов (2)

Реализация `pset__` относительно простая:

- ▶ если слот с заданным именем имеется, обновляем его значение,
- ▶ если слота нет, но есть свободный слот — записываем туда,
- ▶ иначе аварийно завершаем программу.

Реализация чтения слота интереснее — в ней есть рекурсия:

- ▶ если слот нашёлся в текущем объекте, возвращаем связанное с ним значение,
- ▶ если слот не нашёлся, рекурсивно ищем в прототипе,
- ▶ если прототипа нет — аварийно завершаем программу.

В примерах кода мы аварийно завершаем программу вызовом `(call halt 127)`. Если `(symbol <имя>)` компилируется в определение переменной, хранящей массив символов своего имени, то здесь можно распечатывать сообщение об ошибке, упоминая данный слот.

## Семантика НУИЯПа-П (7) — запись в слот

```
(function pset__ (obj symbol value)
  (var (p ptr) (end ptr))
  (p "=" obj)
  (end "=" (obj "+" (Slot "*" MAX_SLOTS)))
  (while (((L p) "<" (L end)) and
          (((L ((L p) "+" Slot_symbol)) "<>" (L symbol)) and
           ((L ((L p) "+" Slot_symbol)) "<>" 0)))
    (p "=" ((L p) "+" Slot))
  )
  (if ((L p) "<" (L end))
    (if ((L ((L p) "+" Slot_symbol)) "=" 0)
      (((L p) "+" Slot_symbol) "=" (L symbol))
    )
    (((L p) "+" Slot_value) "=" (L value))
  else
    (call halt 127)
  )
)
```

## Семантика НУИЯПа-П (8) — чтение из слота

```
(function pget__ (obj symbol)
  (var (p ptr) (end ptr) (proto ptr))
  (p "=" obj)
  (end "=" (obj "+" (Slot "*" MAX_SLOTS)))
  (while (((L p) "<" (L end)) and
           (((L ((L p) "+" Slot_symbol)) "<>" (L symbol)) and
            ((L ((L p) "+" Slot_symbol)) "<>" 0)))
    (p "=" ((L p) "+" Slot))
  )
  (if (((L p) "<" (L end)) and ((L ((L p) "+" Slot_symbol)) "<>" 0))
    (return (L ((L p) "+" Slot_value)))
  else
    (proto "=" (call pget__ (L obj) proto__))
    (if ((L proto) "<>" 0)
      (return (call pget__ (L proto) (L symbol)))
    else
      (call halt 127)
    )
  )
)
```

## Семантика НУИЯПа-П (9) — Вызов методов

Вызов метода — это по сути чтение слота с данным именем + вызов соответствующей функции. Аналогично его можно рассматривать как синтаксический сахар

```
(p-mcall t.Object s.Name e.Args
```

↓

```
(call (pget__ t.Object s.Name) t.Object e.Args)
```

Но опять же, это должен быть примитив, чтобы выражение `t.Object` не вычислялось дважды.