Санкт-Петербургский Политехнический Университет Петра Великого
Институт компьютерных наук и технологий
Кафедра компьютерных систем и программных технологий

# Реферат

Разработка препроцессора макрокоманд для языка Kotlin

**Работу выполнил:**
Косолапов С.А.
Группа: 63501/3
**Преподаватель:**
проф., д.т.н. Мелехин В.Ф.

Санкт-Петербург
2016

# Содержание

# 1 Введение

Со времени появления первых языков программирования технологии написания программ постоянно совершенствуются. Язык программирования высокого уровня является средством описания задачи в терминах, близких как человеку, так и вычислительному устройству, на котором будет исполняться программа. За годы своего развития языки приобрели большое число конструкций, удобных для использования программистом в рассуждениях, а также упрощающих и ускоряющих написание и анализ программного кода.

Одним из препятствий, осложняющих написание понятного и лаконичного кода, является его дублирование. Языки программирования предоставляют следующие концепции, позволяющие частично решать данную проблему:

- Использование функций – наиболее распространённая концепция, реализованная в большинстве Тьюринг-полных языков
- Модульность на уровне файлов подразумевает возможность использовать один и тот же модуль (в виде исходного кода, либо прекомпилированный) повторно. Реализована в большинстве языков, независимо от используемых парадигм
- Использование функций как объектов первого класса - подход функционального программирования, позволяющий параметризовать участки кода. Он заключается в возможности определять функции как значения и передавать их в качестве параметра
- Наследование – концепция объектно-ориентированного программирования, позволяющая использовать код базового класса в классе-потомке

Приведённые концепции позволяют абстрагировать и локализовать повторяющиеся участки кода. Однако существуют случаи, когда с использованием средств языка программирования не представляется возможным достичь простоты и элегантности решения. В этом случае прибегают к использованию средств, выходящих за его рамки и производящих препроцессирование исходного кода, результатом которого является создание программ. Такой подход называется метапрограммированием, а одним из наиболее распространённых средств данного подхода являются макрорасширения.

# 2 Макрорасширения в языках программирования

## 2.1 Макрорасширения императивных языков

Первыми языками программирования ЭВМ были ассемблеры. Программа, написанная на таком языке, представляет собой последовательность символьных инструкций и их параметров, в результате преобразуемая компилятором в последовательность соответствующих машинных инструкций. То есть, по сути, изначально структурные различия между программой на ассемблере и аналогичной последовательностью инструкций в машинных кодах практически отсутствуют. Такое представление программ неудобно для восприятия человеком, так как влечёт за собой огромный разрыв между семантикой языка и семантикой описываемой на нём задачи. Неудивительно, что в скором времени стали появляться идеи по уменьшению такого семантического разрыва, и впоследствии появились первые средства для этого – макрокоманды (или макроинструкции) для ассемблеров.

*Макро*инструкции представляют собой параметризуемую последовательность *микро*инструкций ассемблера, то есть ассемблерных команд. За счёт параметризуемости появляется новый уровень абстракции, способный упростить написание программ, упрощая их понимание и анализ для людей. Идея макроинструкций для ассемблеров получила широкое распространение в конце 50-х годов 20 века, предлагались различные концепции[7]. Наиболее современный вид макро для ассемблера впервые был предложен Дугласом Макилроем (Douglas McIlroy) в 1959 году[11].

Впоследствии подход, подобный подходу для описания макрокоманд ассемблера, был реализован в языке C[1]. Макрорасширение встроено в препроцессор языка, который, в сумме, предоставляет некоторые приёмы метапрограммирования, такие как простейшая кодогенерация и условная компиляция.

Макро могут использоваться как объекты (object-like) и как функции (function-like). Причём большой проблемой является то, что макро из основного языка для программиста выглядят так же, как переменные или функции, что может повлечь случайное присвоение значения макро или случайный вызов функционального макро вместо функции с таким же именем. Для разрешения подобных проблем в языке C принято определять макро большими буквами.

В действительности, язык макро для языка C полностью отделён от основного языка, что влечёт за собой проблемы, связанные с *гигиеной*, о чём будет сказано позже, а также отсутствием возможностей интраспекции и, как следствие, большими ограничениями в применении его как серьёзного средства метапрограммирования.

Языки-последователи языка C практически без изменений унаследовали от него препроцессор, в том числе и макрорасширение. Как результат, в языках Objective C, C++ и некоторых других могут возникнуть такие же серьёзные проблемы при использовании макро, как это происходит с языком C. С другой стороны, отсутствие изменений в макрорасширениях можно объяснить снижением необходимости в использовании препроцессора: в более высокоуровневых языках появились полноценные константные типы, шаблоны в C++ и дженерики (generics) в Objective C, а также другие конструкции основного языка, которые значительно ограничивают область применения препроцессора.

## 2.2 Макрорасширения для декларативных языков

С появлением в 1958 году языка Lisp, исторически второго языка высокого уровня после FORTRAN'a, начала своё развитие функциональная прадигма программирования. Именно на её базе родились концепции макрорасширений, которые будут рассмотрены далее. Первые макрорасширения для языка Lisp появились в 1963 году[14]. Со временем Lisp приобрёл свои диалекты, распространённые в наши дни - Scheme (середина 70-х годов), Common Lisp (1984 год) и наиболее молодой язык Clojure (2007 год). Далее, в качестве реализации языка с LISP-подобным синтаксисом, будет рассматриваться язык Scheme, потому что именно для него велась большая часть исследований в области макрорасширений функциональных языков.

Lisp-подобные языки, в частности и язык Scheme, имеют очень простой синтаксис, отражающий одну из основных концепций языка – *гомоиконность*, то есть единое представление кода и данных. Общий вид любой структуры данных языка, в том числе и являющихся синтаксическими конструкциями (S-выражения), представляет список:

```
1 (cmd-or-fun param1 ... paramN)
```

Причём, как и полагается функциональному языку, функции в нём являются объектами первого класса, то есть работа с ними осуществляется так же, как с переменными. В частности, функция может быть передана в другую функцию в качестве параметра. Кроме функций, язык позволяет использовать операторы ветвления и мультиветвления, а циклы заменяются рекурсией.

Ветвление не может быть заменено на функции, так как в случае использования условного оператора вычисление должно производиться только по одной ветви, а при использовании функций, независимо

от истинности заданного в условии предиката, будут выполнены обе ветви[2]. То же самое касается и условных операторов – логических «И» (AND) и «ИЛИ» (OR): в случае «ИЛИ», при условии истинности первого предиката, результат остальных неважен, как неважен для «И» результат вычисления предикатов, кроме первого, при условии его ложности.

Несмотря на невозможность определения подобных конструкций с помощью функций, в LISP-подобных языках, в том числе и в Scheme, есть средство абстракции более высокого уровня, позволяющее сделать это – макрорасширения. Таким образом, появляется возможность введения в язык новых синтаксических конструкций, которые ограничиваются лишь базовым представлением конструкций языка.

Простой синтаксис, позволяющий написать интерпретатор языка за весьма короткое время, является одной из основных причин, почему язык так популярен для исследований, в том числе, для макрорасширений.

# 3 Концепции макрорасширений

Как было сказано выше, именно в языке Scheme были впервые реализованы многие концепции, используемые впоследствии в реализациях макрорасширений для многих языков программирования. Ниже рассмотрены те из них, которые представляются наиболее интересными при реализации и полезными для применения в практике программирования.

## 3.1 Macro-by-Example

Концепция Macro-by-Example («макро по примеру», MBE) была описана в статье Е.Колбекера (Eugene Kohlbecker) в 1987 году[9], где утверждается, что данный механизм используется в различных версиях языка Scheme с 1982 года, однако не был ранее описан. Приведённая спецификация макрорасширения предлагает использовать более декларативный подход к определению макро, чем традиционный, с помощью конструкций итерации и сопоставления.

Язык определения макро, включающий MBE, включает в себя входную и выходную спецификацию кода. Этот язык имеет следующие особенности:

- Для конкретизации определения макро введён механизм сопоставления по шаблону (pattern-matching), включающий проверку ошибок, осуществляемую на входе
- Выходная спецификация соответствует форме выхода
- Повторения определяются естественным образом как для входной, так и для выходной спецификаций

Процессирование такого макроопределения производится в два этапа: проход (staging) и представление (representation). Анализ, производимый на первом этапе, показывает, что возможно использование набора тестов, селекторов и конструкторов во время макроподстановки. Второй этап включает в себя поиск подходящего представления для функций.

Общий вид можно представить следующим образом:

```
(declare-syntax <name-of-macro> [(<input 1>) <output1>] [(<input2>) <output2>] ...)
```

В результате, к примеру, чтобы определить операцию логического умножения, необходимо написать следующее макроопределение:

```
(declare-syntax and
  [(and) true]
  [(and e) e]
  [(and e1 e2 ...) (if e1 (and e2 ...) false)])
```

Такой код позволяет простыми средствами определить безопасные конструкции, позволяющие, в том числе, расширить синтаксис базового языка.

## 3.2 Гигиена

Как было указано выше, макро в языке C имеют ряд проблем, прежде всего, связанных со способом процессирования таких макро[8]. Макроподстановка осуществляется «в лоб», игнорируя синтаксическую идентичность между вызовами макро и функций или обращению к переменным. Таким образом, неправильный способ именования макро может привести к неочевидным конфликтам имён. Такие конфликты иногда очень трудно разрешить. Можно допустить и другую ситуацию: предположим, что макроопределение объявлено в локальной области видимости. Так как областей видимости для макро нет, то мы можем запросто вызвать макро в совершенно другой области видимости, что приведёт к ошибке.

Ситуация осложняется ещё сильнее в случае, когда в макроопределении используются переменные. Ничто не запрещает в макроопределениях языка C объявлять переменные. Но также нет никакой проверки наличия переменной с таким же именем в области вызова макро! Таким образом, переменная, объявленная до вызова макро, может быть изменена. Ниже приведены фрагменты кода на языке C, иллюстрирующие данную проблему.

К примеру, если мы хотим посчитать дискримнинант для квадратного уравнения, можем определить следующий макро[5].

```
#define discriminant(a, b, c) b*b - 4*a*c
```

В некоторых случаях это будет работать. Однако несложно привести пример кода, для которого макроподстановка будет осуществлена совсем не так, как этого может ожидать программист. К примеру, код ниже иллюстрирует, что может произойти, когда

```
1 // подстановка произойдёт правильно
2
3 10*discriminant(42, x, y)
4
5 // подстановка произойдёт неправильно
6 2*discriminant(x - y, x + y, x - y)*5
```

Всему виной отсутствие восприятия передаваемых параметров как целостных объектов парсера: они подставляются «как есть». В результате, во втором случае, вместо ожидаемого

**2*(x + y)*(x + y) - 4*(x - y)*(x - y)**

получим

**2*x + y*x + y - 4*x - y*x - y*5**.

Как и в случае разграничения именования объектов языка и макро, в данном случае могут помочь простые решения, такие как, например, обрамление скобками всех параметров макроопределения. Иллюстрирующий такой подход код приведён ниже.

```
1 #define discriminant(a, b, c) ((b)*(b) - 4*(a)*(c))
2
3 // подстановка произойдёт правильно
4 2*discriminant(x - y, x + y, x - y)*5
5 // получим 2*((x + y)*(x + y) - 4*(x - y)*(x - y))*5
```

Есть и довольно странные вещи, связанные с использованием макро. К примеру, такой код будет работать:

```
1 #define foo "salad
2 printf(foo bar")
```

Таким образом, макро в языке C позволяют писать абсолютно неправильные синтаксические конструкции.

Аналогичные проблемы существуют не только для макро языка C и позволяют понять, почему для макрорасширений важно свойство гигиены. Столкнулись с ними впервые ещё во времена, когда макрорасширения для языка Lisp только появлялись[13].

Свойство гигиены определяет, что переменные, определённые внутри макро, никогда не будут иметь конфликты с переменными, объявленными в области, где будет происходить раскрытие макро. Иными словами, пространство имён, определённое внутри макро, не должно пересекаться с пространством имён области, где будет произведена макроподстановка. Таким образом, наилучшим случаем будет являться гарантия, что любая переменная, объявленная внутри макроопределения, будет иметь уникальное имя при каждом его раскрытии.

Впервые гигиена была реализована и формально описана в 1986 году для языка Scheme[10]. Реализовать гигиену можно по-разному. Например, в реализации гигиены для языка Nemerle используется цветовое кодирование[13]. Каждый идентификатор внутри макро описывается как Ref(v, c, g), где v – имя идентификатора, c – цвет, a g – глобальное окружение. Цвет уникален для каждого раскрытия макро.

Существуют также другие подходы к реализации макрорасширений, обладающих свойством гигиены[2][6][4].

## 3.3 Квазицитирование

Гигиенические макрорасширения позволяют не задумываться об именах переменных, гарантируя, что сгенерированное имя точно будет уникальным и конфликтов не возникнет. Однако есть и обратная сторона медали. В случае, если внутри макроопределения необходим доступ к некоторой части кода, у программиста нет никаких способов сделать это, потому что любое имя переменной, объявленное внутри макро, будет изменено.

Квазицитирование реализовано в некоторых языках, например, Lisp[3], Nemerle[13], Scala[12].

## 3.4 Passing styles

# 4 Макрорасширения в многопрадигменных ЯП

## 4.1 Реализация макросов в языке Rust

## 4.2 Реализации макросов в языке Scala

# 5 Необходимость макросов в языке Kotlin

# 6 Возможные подходы к реализации макросов в языке Kotlin

# 7 Заключение

# Список литературы


[1] Macros – the c preprocessor. `https://gcc.gnu.org/onlinedocs/cpp/Macros.html`. Доступ осуществлён: 12.08.2016.

[2] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[3] Alan Bawden. Quasiquotation in lisp. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 4–12, 1999.

[4] W. Clinger. Hygienic macros through explicit renaming, 1991.

[5] William Clinger and Jonathan Rees. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 155–162, New York, NY, USA, 1991. ACM.

[6] David Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Boston, MA, USA, 2010. AAI3398847.

[7] I.D. Greenwald; M. Kane. The Share 709 System: Programming and Modification. *Journal of the ACM*, 6(2):128–133, 1959.

[8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1978.

[9] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 77–84, New York, NY, USA, 1987. ACM.

[10] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 151–161, New York, NY, USA, 1986. ACM.

[11] M. Douglas McIlroy. Macro instruction extensions of compiler languages. *Commun. ACM*, 3(4):214–220, April 1960.

[12] Denys Shabalin, Eugene Burmako, and Martin Odersky. Quasiquotes for Scala. Technical report, 2013.

[13] Kamil Skalski. Syntax-extending and type-reflecting macros in and object-oriented language. Master's thesis, Institute of Computer Science, University of Wroclaw, Wroclaw, 2005.

[14] Guy L. Steele, Jr. and Richard P. Gabriel. The evolution of lisp. *SIGPLAN Not.*, 28(3):231–270, March 1993.