DOI: 10.15514/ISPRAS-2026-38(6)-24



# Автоматическая генерация тестов для оптимизаторов на основе атрибутной грамматики

*С.В. Зеленов, ORCID: 0000-0003-0446-0541, <zelenov@ispras.ru>*

*Институт системного программирования РАН,*
*109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*

**Аннотация.** В статье представлен подход, позволяющий полностью автоматически генерировать тесты для оптимизирующих модулей в компиляторах. Подход основан на задании модели входных данных для компилятора в виде атрибутной BNF-грамматики, где атрибуты используются для автоматического обеспечения статической семантики. Подход реализован в виде программного комплекса на языке Haskell. Практическое применение подхода иллюстрируется на примере разработки генератора тестов для оптимизации «Удаление общих подвыражений».

**Ключевые слова:** Тестирование на основе моделей; тестирование компиляторов; оптимизатор; автоматическая генерация тестов; атрибутная грамматика

**For citation:** Зеленов С.В. Автоматическая генерация тестов для оптимизаторов на основе атрибутной грамматики. Труды ИСП РАН, 2026, том 38, вып. 6, часть 2, с. 41–56. DOI: 10.15514/ISPRAS-2026-38(6)-24.

# Automated Test Case Generation for Optimizers Based on Attribute Grammar

*S.V. Zelenov, ORCID: 0000-0003-0446-0541, <zelenov@ispras.ru>*

*Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
*25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*

**Abstract.** In the paper, we present an approach to automated generation tests for optimization modules in compilers. The approach is based on describing of input data for compiler in attribute BNF-grammar. Attributes are used to automatedly meet semantics context conditions. The approach is implemented as a program in Haskell. We illustrate practical application of the approach by an example of test generator development for the Common Subexpression Elimination optimization.

**Keywords:** model-based testing; compiler testing; optimizer; automated test generation; attribute grammar

**For citation:** Zelenov S.V. Automated Test Case Generation for Optimizers Based on Attribute Grammar. Trudy ISP RAN/Proc. ISP RAS, 2026, vol. 38, issue 6, part 2, pp. 41-56 (in Russian). DOI: 10.15514/ISPRAS-2026-38(6)-24.





## *1. Введение*

Компиляторы являются критическим программным обеспечением (ПО), поскольку от них зависит разработка подавляющего большинства программных продуктов.

Тестирование компилятора невозможно без привлечения средств автоматизации создания тестов, поскольку входные данные для компилятора представляет собой чрезвычайно сложный объект с многочисленными внутренними связями и зависимостями.

Работа компилятора состоит из четырех основных фаз [1]:

- синтаксический анализ;
- семантический анализ;
- оптимизация внутреннего представления;
- генерация исполняемого кода.

Как правило, тесты для компилятора создаются в виде программ на входном языке программирования. Проведение же модульного тестирования, во-первых, требует наличия доступа к внутренней инфраструктуре компилятора, а во-вторых, очень сильно осложняется необходимостью обеспечивать консистентное состояние этой внутренней инфраструктуры для каждого теста.

Подходы к построению тестов на основе фаззинга текста [2, 3] достаточно легко использовать, и они пригодны для тестирования компилятора любого языка. Однако, во-первых, для таких подходов требуется изначально иметь достаточно богатый корпус входных текстов, а во-вторых, фаззинг текста совершенно не гарантирует даже синтаксической корректности тестов, и поэтому получающиеся тесты де-факто пригодны лишь для тестирования лексического анализатора.

Инструмент Csmith [4] нацелен на генерацию исполнимых тестов на языке C99 и имеет довольно много средств для настройки того, какие конструкции и в каких пропорциях желательно получать в сгенерированных тестах. Однако, во-первых, этот инструмент пригоден для тестирования только компиляторов языка C, а во-вторых, не нацелен на тестирование никакой конкретной фазы компиляции. В частности, настроить его на генерацию тестов для конкретного оптимизатора можно лишь очень косвенным образом.

Использование формальных спецификаций и моделей [5-6] показало себя наиболее успешным подходом для организации нацеленного автоматического тестирования на систематической основе.

Подходы [7-10], нацеленные на генерацию тестов для фазы синтаксического анализа, основаны на использовании формального описания входного языка компилятора в виде контекстно-свободной грамматики в форме Бэкуса-Наура (Backus-Naur Form, BNF) и на идее нисходящего раскрытия нетерминалов. При этом, в работах [7, 9-10] основное внимание уделяется систематическому достижению различных критериев покрытия на основе BNF, а в работе [8] предлагается использовать стохастический подход, когда для различных альтернатив правил BNF дополнительно заданы также частоты их выбора генератором тестов. Общей проблемой тестирования фазы синтаксического анализа является необходимость наличия в компиляторе возможности запустить только синтаксический анализ, чего, как правило, не бывает.

Для тестирования фазы семантического анализа требуется описание семантики языка. В работе [11] для формулировки предикатов статической семантики используется формализм VDM. Процесс генерации состоит в переборе всех комбинаций значений атрибутов, при которых дизъюнкты в предикатах обращаются в True.

В работах [12-13] статическая семантика описывается в виде атрибутной грамматики [14]. Генерация тестов основана построении кандидатов путем нисходящего раскрытия нетерминалов BNF с последующей проверкой ограничений на атрибуты, когда кандидаты,

не удовлетворяющие семантическим предикатам, отбрасываются. В работе [15] предложен похожий подход для генерации тестов для динамической семантики на основе формализма ASM (Abstract State Machine) [16]. Проблемой в этих подходах является *неэффективность* генерации, в том смысле, что построенные кандидаты крайне редко удовлетворяют всем предикатам, так что для получения одного теста приходится строить огромное количество кандидатов. А поскольку подавляющее количество кандидатов отбрасывается, страдает систематичность тестирования.

Промышленные компиляторы используют многочисленные оптимизации [17] для получения эффективных программ. Главной трудностью при создании тестов для оптимизатора является то, что для тестирования корректности данного модуля оптимизации требуется предоставить компилятору для компиляции некоторую тестовую программу, обработка которой *заставит* компилятор задействовать тестируемый оптимизатор.

Подходы к построению тестов на основе фаззинга структуры программы [18-20] также легко использовать. Однако, во-первых, каждый из этих подходов пригоден для тестирования компилятора лишь какого-то конкретного языка, во-вторых, для таких подходов требуется изначально иметь достаточно богатый корпус подходящих входных текстов с разметкой структуры, а в-третьих, фаззинг структуры не всегда гарантирует семантическую корректность тестов.

Идеальный метод генерации тестов для оптимизатора должен обладать следующими свойствами:

- он должен генерировать именно тесты, нацеленные на данный оптимизатор (то есть чтобы их обработка *заставила* компилятор задействовать тестируемый оптимизатор);
- он должен генерировать тесты систематично;
- он должен быть пригоден для генерации тестов на любом языке программирования;
- он должен обеспечивать семантическую корректность тестов;
- генерация тестов должна быть эффективной, то есть все необходимые условия должны обеспечиваться конструктивно, по построению, без массивной фильтрации кандидатов;
- генерируемые тесты должны быть исполнимыми.

В работе [21] предложен метод OTK (Optimizer Testing Kit) для разработки модели и генератора тестов, нацеленных на тестирование конкретного заданного модуля оптимизации. Метод включает в себя построение формальной модели теста на основе анализа алгоритма оптимизации, а затем – ручную разработку двух основных компонентов генератора тестов:

- итератора, который служит для перебора модельных структур;
- мэппера (mapper), который предназначен для отображения модельной структуры в семантически корректную входную программу для компилятора.

Метод OTK был успешно применен в нескольких проектах по тестированию оптимизаторов нескольких компиляторов нескольких языков, включая C, C++, а также JavaScript-подобного языка со строгой типизацией. В ходе проектов были разработаны модели и получены тесты для более двух десятков оптимизаторов, включая различные оптимизаторы выражений, оптимизаторы циклов и другие модули оптимизаций.

Опыт применения метода OTK показал, что ручная разработка компонентов генератора в очень многих случаях хорошо алгоритмизируется и могла бы быть осуществлена автоматически при условии дополнительного задания для модели некоторых вспомогательных параметров, которые описывали бы ограничения на семантику и на размер генерируемых тестов.





В настоящей статье предлагается подход для описания модели входных данных для оптимизатора в виде атрибутной BNF-грамматики. Атрибуты модели позволяют описывать семантику и ограничения на генерацию. В рамках этого подхода на основе такого задания модели полностью автоматически строится генератор тестов для целевого оптимизатора. При этом следует подчеркнуть, что построенный таким образом генератор в процессе генерации тестов сразу (то есть без последующей фильтрации) конструирует семантически корректные программы, удовлетворяющие описанию семантики, заданному в исходной атрибутной грамматике.

## *2. Описание предлагаемого подхода*

### 2.1 Архитектура генератора тестов

В нашем подходе мы строим генератор тестов в соответствии со следующей архитектурой, идея которой восходит к [21]. Работа генератора состоит из двух основных стадий (рис. 1):

- итерация различных абстрактных модельных тестовых структур;
- отображение каждой сгенерированной в ходе первой стадии модельной тестовой структуры в текст – соответствующую тестовую программу.

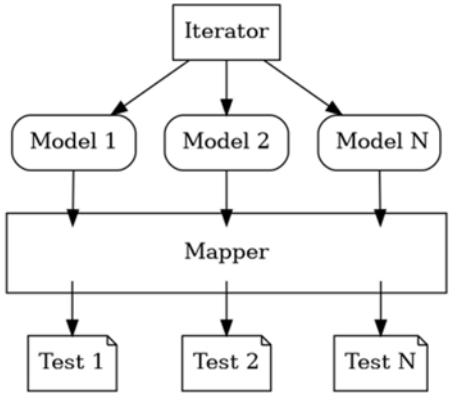



*Рис. 1. Процесс генерации тестов на основе моделей.*
*Fig. 1. Model-based test generation process.*

Модель тестовой структуры описывается в виде атрибутной BNF-грамматики. На основе этого описания автоматически строится соответствующий генератор тестов.

### 2.2 Нотация BNF для описания модели

Для начала приведем конкретный пример модели и ее BNF-грамматики.

Рассмотрим следующий язык арифметических выражений. Значения выражения можно либо напечатать, либо присвоить именованной константе. Статическая семантика языка заключается в том, что

1) все константы должны иметь разные имена;
2) имя константы должно объявляться до его использования.

BNF-грамматика для такого языка представлена на рис. 2.

Теперь дадим несколько комментариев по собственно нотации BNF.

Определяемый в грамматическом правиле нетерминал стоит слева от знака ::=. Нетерминал в угловых скобках (например, <INT>) является лексическим нетерминалом (то есть

принадлежит лексической части грамматики). При отображении в текст элементы обычного нетерминала будут разделяться пробелами (например, Print x ;), а элементы лексического нетерминала будут выводиться слитно (например, -127).

Фигурные скобки { ... } означают список, то есть повторение своего содержимого ноль или более раз. Квадратные скобки [ ... ] означают опциональный элемент, то есть их содержимое может отсутствовать. Круглые скобки ( ... ) служат для группировки подвыражения. Альтернативы в (под-)выражении разделяются вертикальной чертой |. Терминальные лексемы записываются в обратных одинарных кавычках. Квадратные скобки, перед которыми стоит знак «доллар»: $[ ... ], означают символьный класс, как в регулярных выражениях.

Символьный класс запрещен в правой части правила для обычного нетерминала. Ссылка на обычный нетерминал запрещена в правой части правила для лексического нетерминала. Остальные конструкции допустимы везде.

```
StartRule ::= { Insn } ;
Insn ::= Def | Print ;
Def ::= `Const` <ID> `=` Expr `;` ;
Print ::= `Print` Expr `;` ;
Expr ::= <INT> | <ID> | SubExpr ;
SubExpr ::= `(` Expr { (`+` | `-` | `*`) Expr } `)` ;
<ID> ::= $[A-Za-z_] { $[A-Za-z_0-9] } ;
<INT> ::= `0` | [`-`] $[1-9] { $[0-9] } ;
```

*Рис. 2. BNF-грамматика для языка арифметических выражений.*
*Fig. 2. BNF-grammar for the arithmetic expressions language.*

## 2.3 Порядок генерации модельной тестовой структуры

Генерация модельной тестовой структуры происходит по принципу обхода грамматики в глубину.

Генерация модельной структуры для некоторого BNF-выражения состоит в последовательной генерации модельных подструктур для каждого из элементов этого выражения. Для случая ссылки на нетерминал рекурсивно вызывается генерация модельной структуры для выражения, определяющего этот нетерминал. Если выражение состоит из нескольких альтернатив, конкретное выражение-альтернатива для генерации выбирается случайно.

Вернемся к рассмотрению примера языка арифметических выражений. Если сгенерировать тесты по грамматике этого языка (рис. 2), то получится, например, такой тест:

```
Print Pn3 ;
Const JAu = x45 ;
Const JAu = ( 117 - ( -127 - phO + -868 ) * Hns ) ;
```

Этот тест нарушает сразу оба правила статической семантики:

- константа JAu объявлена повторно;
- используются не объявленные константы Pn3, x45, phO и Hns.

## 2.4 Атрибуты в BNF для описания статической семантики

Для того, чтобы модельная тестовая структура генерировалась с учетом статической семантики, некоторые конструкции BNF-грамматики необходимо снабдить соответствующими атрибутами.

Атрибуты для конкретной конструкции грамматики записываются в специальной секции **непосредственно перед** этой конструкцией. Оформление такой секции слегка отличается в зависимости от *категории* аннотируемой конструкции, а именно:





- `@@{ ... }@` – атрибуты для всей грамматики.
- `@={ ... }@` – атрибуты для данного правила.
- `@|{ ... }@` – атрибуты для данного многоальтернативного выражения (то есть для всех его альтернатив).
- `@*{ ... }@` – атрибуты для данного одного выражения-альтернативы.
- `@.{ ... }@` – атрибуты для данного элемента выражения.

Собственно значения атрибутов перечисляются внутри соответствующей секции в форме списка объявлений вида `имя = `значение`;`.

Перейдем непосредственно к тому, каким именно образом закодировать статическую семантику с помощью атрибутов в BNF-грамматике.

### 2.4.1 Семантическое правило «Уникальность имен»

Проблему уникальности имен предлагается решать конструктивно. А именно, нужно заставить генератор выдавать гарантированно различные имена. Один из способов состоит в том, чтобы генерировать имена, в которые, в частности, содержат в себе номер генерируемого имени: `val_1`, `val_2`, `val_3` и т.д.

Чтобы это сделать, требуется в первую очередь в контексте всей грамматики с помощью атрибута `Int` описать целочисленную переменную, которая будет использоваться генератором как счетчик. Назовем ее `count`:

```
@@{ Int = `count`; }@
```

Теперь для того, чтобы сгенерировать значение некоторого элемента BNF-выражения с использованием счетчика, требуется в контексте этого BNF-элемента задать значение атрибута `str` в следующем виде: знак плюс, имя счетчика, знак двоеточие, а затем – шаблон в стиле C-функции `printf` от одного целочисленного аргумента.

В нашем примере нужно в контексте элемента `<ID>` в правой части правила для нетерминала `Def` задать значение атрибута `str`:

```
Def ::= `Const` @.{ str = `+count:val_%d`; }@ <ID> `=` Expr `;` ;
```

Если теперь сгенерировать тесты по такой обогащенной атрибутами грамматике этого языка, то получатся, например, такие тесты:

```
=== test 1 ===
Print ( -548 * 0 + vEZ ) ;
Const val_1 = 452 ;
Const val_2 = ( jhz * rjM + ( 0 + 154 + Tpa ) ) ;
=== test 2 ===
Const val_1 = -192 ;
Const val_2 = eYV ;
Print -992 ;
```

Заметим, для каждого очередного теста счетчик инициализируется заново.

Итак, мы добились, чтобы объявляемые имена констант в рамках одного теста были уникальными. Перейдем теперь ко второму правилу семантики: в выражениях должны использоваться только имена констант, которые были объявлены ранее.

### 2.4.2 Семантическое правило «Использовать только ранее объявленные имена»

Проблему использования только ранее объявленных имен также предлагается решать конструктивно. А именно, нужно заставить генератор хранить множество всех объявленных к данному моменту имен (то есть непосредственно после генерации узла модельной

структуры, в котором объявляется новое имя, добавлять это имя в множество объявленных имен), и в качестве используемого имени выбирать элемент из этого множества.

При этом возникает следующее естественное требование к работе генератора: пока множество объявленных имен пусто (то есть ни одного имени еще не объявлено), запрещены раскрытия тех BNF-альтернатив, в которых обязательно требуется использовать ранее объявленное имя. В общем случае это требование можно удовлетворить путем перебора BNF-альтернатив с возвратом из тех из них, в которых это требование нарушается, до нахождения первой успешной BNF-альтернативы.

Вернемся к рассмотрению нашего примера языка арифметических выражений.

Чтобы решить проблему использования только ранее объявленных констант, требуется в первую очередь в контексте всей грамматики с помощью атрибута `StrSet` описать переменную типа «множество строк», которая будет использоваться генератором хранения имен объявленных констант. Назовем эту переменную `Consts`:

```
@@{ ...
    StrSet = `Consts`;
  }@
```

Чтобы добавить строковое представление некоторого элемента BNF-выражения в множество, требуется в контексте этого BNF-элемента задать атрибут `addThisAsStr` и присвоить ему в качестве значения имя множества.

В нашем примере нужно в контексте элемента `<ID>` в правой части правила для нетерминала `Def` задать значение атрибута `addThisAsStr`:

```
Def ::= `Const`
    @.{ str = `+count:val_%d`;
        addThisAsStr = `Consts`;
    }@ <ID> `=` Expr `;` ;
```

Теперь для того, чтобы при генерации в качестве значения некоторого элемента BNF-выражения использовать строку из множества, требуется в контексте этого BNF-элемента задать атрибут `choose` и присвоить ему в качестве значения имя множества.

В нашем примере нужно в контексте элемента `<ID>` в правой части правила для нетерминала `Expr` задать значение атрибута `choose`:

```
Expr ::= <INT> | @.{ choose = `Consts`; }@ <ID> | SubExpr ;
```

Если теперь сгенерировать тесты по такой обогащенной атрибутами грамматике этого языка, то получится, например, такой тест:

```
Const val_1 = ( ( val_1 + -429 * 789 ) + val_1 ) ;
Const val_2 = ( ( val_2 - val_2 * 482 ) + ( -802 - 368 ) ) ;
```

В силу того, что генерация элементов BNF-выражения происходит последовательно, для модельного узла `Def` по умолчанию сперва генерируется имя новой константы, а только потом – арифметическое выражение для правой части. Таким образом, в тот момент, когда генерируется правая часть, в множестве `Consts` уже находится имя объявляемой в этом узле константы.

Чтобы исправить ситуацию, нужно изменить порядок генерации элементов в BNF-правиле `Def`.

### 2.4.3 Управление порядком генерации элементов для BNF-выражения

Чтобы задать порядок генерации некоторого элемента BNF-выражения относительно других элементов этого BNF-выражения, требуется в контексте этого BNF-элемента задать атрибут `genOrder` и присвоить ему в качестве значения целое число. По умолчанию у всех BNF-элементов этот атрибут считается равным нулю.





Генерация элементов BNF-выражения будет происходить по возрастанию значений их атрибута `genOrder`, а для BNF-элементов с равными значениями атрибута `genOrder` – в порядке следования в BNF-грамматике.

Таким образом, в нашем примере нужно в контексте элемента `<ID>` в правой части правила для нетерминала `Def` задать значение атрибута `genOrder = `1``:

```
Def ::= `Const`
    @.{ str = `+count:val_%d`;
        addThisAsStr = `Consts`;
        genOrder = `1`;
    }@ <ID> `=` Expr `;` ;
```

Если теперь сгенерировать тесты по такой обогащенной атрибутами грамматике этого языка, то получатся, например, такие тесты:

```
=== test0001.tst ===

Print ( 795 + ( -676 * 532 - 898 ) + ( -187 * 312 - 491 ) ) ;
Const val_1 = -120 ;
Print ( val_1 * 482 - val_1 ) ;
Const val_2 = ( 255 * ( val_1 * -176 - 203 ) + val_1 ) ;

=== test0002.tst ===

Const val_1 = ( ( 886 + 304 + ( 0 + 427 * -940 ) ) - -449 + -386 ) ;
Const val_2 = val_1 ;
Const val_3 = -976 ;
Print ( ( val_2 + val_1 - 0 ) - val_3 - ( val_2 - 0 + -566 ) ) ;
```

Окончательная атрибутная BNF-грамматика для языка семантически корректных арифметических выражений представлена на рис. 3.

```
@@{ Int = `count`;
    StrSet = `Consts`;
  }@

StartRule ::= { Insn } ;
Insn ::= Def | Print ;
Def ::= `Const`
    @.{ str = `+count:val_%d`;
        addThisAsStr = `Consts`;
        genOrder = `1`;
    }@ <ID> `=` Expr `;` ;
Print ::= `Print` Expr `;` ;
Expr ::= <INT> | @.{ choose = `Consts`; }@ <ID> | SubExpr ;
SubExpr ::= `(` Expr { (`+` | `-` | `*`) Expr } `)` ;
<ID> ::= $[A-Za-z_] { $[A-Za-z_0-9] } ;
<INT> ::= `0` | [`-`] $[1-9] { $[0-9] } ;
```

*Рис. 3. Атрибутная BNF-грамматика семантически корректных арифметических выражений.*
*Fig. 3. Attribute BNF-grammar of semantically correct arithmetic expressions.*

### 2.4.4 Семантическое правило «Вложенные области видимости для имен»

Немного усложним язык арифметических выражений из нашего примера. А именно, добавим возможность группировать инструкции в блоки. Для этого добавим третью альтернативу в правило `Insn`:

```
Insn ::= Def
       | Print
       | `{` { Insn } `}` ;
```

При этом потребуем, что при выходе из некоторого блока имена всех объявленных в этом блоке констант должны «забываться», но при этом все имена из объемлющих блоков, объявленные до данного блока, в данном блоке «видны».

Для того, чтобы удовлетворить такое требование, нужно заставить генератор перед началом генерации модельного узла, соответствующего за вложенную область видимости, *запомнить* значение множества имен, а после завершения генерации этого модельного узла – восстановить запомненное значение. Таким образом, все изменения в множестве имен будут локализованы внутри процесса генерации этого модельного узла и будут забыты по окончании этого процесса.

Чтобы описанным выше образом локализовать изменения в некотором множестве внутри процесса генерации некоторой BNF-альтернативы, требуется в контексте этой BNF-альтернативы задать атрибут `local` и присвоить ему в качестве значения имя множества.

В нашем примере нужно в контексте третьей альтернативы правила `Insn` задать значение атрибута `local`:

```
Insn ::= Def
       | Print
       | @*{ local = `Consts`;}@ `{` { Insn } `}` ;
```

Если теперь сгенерировать тесты по такой обогащенной атрибутами грамматике этого языка, то получится, например, такой тест:

```
Print ( ( -708 * -168 - 0 ) + ( -647 + -243 + 0 ) - -724 ) ;
{
  Const val_1 = 0 ;
  {
    {
      Print val_1 ;
      Const val_2 = val_1 + 833 ;
      Print val_1 + 538 * val_2 ;
    }
    Print ( 0 * -146 * val_1 ) ;
  }
}
{
  Print -985 ;
  Const val_3 = ( ( 269 - -282 ) + ( -275 - 425 ) * 768 ) ;
}
```

Здесь при объявлении `Const val_2` в правой части использована константа `val_1` из объемлющего блока, а при объявлении `Const val_3` **не** использовано ни одной константы, поскольку все они были объявлены в «соседнем» блоке в другой области видимости и уже «забыты».

## *3. Применение подхода для генерации тестов для оптимизаторов*

В соответствии с методом ОТК [21], в целях генерации тестов для заданного оптимизатора в первую очередь производится анализ алгоритма данной оптимизации. Результатом анализа алгоритма оптимизации является перечень интересных для оптимизатора сущностей и связей между ними. На основании этого перечня разрабатывается формальная модель в объектно-ориентированном стиле. Далее для построения автоматического генератора тестов требуется вручную разработать итератор и мэппер (см. рис. 1).

Подход ОТК позволяет получать итератор и мэппер полностью автоматически на основании описания модели в виде атрибутной BNF-грамматики.

Здесь мы продемонстрируем связку метода ОТК с предложенным в настоящей работе подходом на примере разработки генератора тестов для оптимизации удаления общих подвыражений (Common-subexpression elimination – CSE) в компиляторе языка `C`.

Для начала кратко сформулируем суть алгоритма этой оптимизации.





### 3.1 Алгоритм оптимизации «Удаление общих подвыражений»

Некоторое вхождение $A$ выражения $E$ в программу является *общим подвыражением*, если существует другое вхождение $B$ выражения $E$, причем всегда в ходе выполнения программы вычисление $E$ в точке $B$ предшествует вычислению $E$ в точке $A$, и все операнды выражения $E$ остаются неизменными между этими двумя вычислениями.

*Удаление общих подвыражений* – это такая трансформация программы, которая удаляет повторные вычисления общих подвыражений и заменяет их на использование сохраненных значений.

Локальное удаление общих подвыражений работает в рамках одного линейного участка (basic block).

Глобальное удаление общих подвыражений работает в рамках графа потока управления, представляющего некоторую процедуру.

### 3.2 Перечень интересных для оптимизатора сущностей и связей

Для алгоритма CSE интересными являются следующие сущности:

- процедура,
- линейный участок,
- выражение,
- общее подвыражение.

Интересные связи между перечисленными сущностями суть следующие:

- Процедура содержит несколько линейных участков.
- Линейный участок содержит несколько выражений.
- Выражение содержит несколько подвыражений (каждое из которых в свою очередь тоже является выражением).
- Некоторые подвыражения могут являться общими подвыражениями.

Будем моделировать структуру линейных участков в процедуре в виде последовательности частей, каждая из которых начинается с линейного участка и завершается инструкцией `if-then-else`. Для инструкции `if-then-else` требуется условие ветвления, которое мы будем представлять как сравнение двух переменных.

Будем моделировать линейный участок в виде последовательности присваиваний с какими-то подвыражениями в правой части. При этом все переменные будем объявлять в виде аргументов процедуры, а в теле процедуры новых переменных объявлять не будем, а будем лишь оперировать с переменными, объявленными как аргументы.

### 3.3 Описание модели в виде BNF-грамматики

BNF-грамматика для приведенного выше перечня сущностей и связей представлена на рис. 4.

По сути, это некоторая редукция BNF-грамматики языка `C`, которая содержит все сущности и связи из нашего перечня. Теоретическое обоснование возможности построения такой редукции для произвольного перечня сущностей и связей можно найти в работе [22].

Теперь требуется добавить в эту грамматику атрибуты для двух целей:

1) обеспечить семантическую корректность;
2) обеспечить, чтобы правило `CSE` действительно представляло бы собой **общее** подвыражение.

## 3.4 Семантические коррекции модели

В нашем случае для обеспечения семантической корректности нужно удовлетворить два требования:

1) все имена аргументов процедуры (правило `Arg`) должны быть уникальны;
2) все имена используемых переменных (правило `Var`) должны быть именами аргументов процедуры.

Методы обеспечения обоих этих требований были подробно описаны в предыдущем разделе.

Для того, чтобы имена аргументов были уникальны, объявим для грамматики счетчик `Na` и зададим в контексте элемента `<ID>` в правой части правила для нетерминала `Arg` атрибут `str = `+Na:a_%d`;`.

Для того, чтобы используемые переменные были аргументами, объявим для грамматики множество `Args` и зададим в контексте элемента `<ID>` в правой части правила для нетерминала `Arg` атрибут `addThisAsStr = `Args`;`, а в контексте элемента `<ID>` в правой части правила для нетерминала `Var` атрибут `choose = `Args`;`.

Соответствующие добавки в грамматику представлены на рис. 5.

```
GCSE ::= Fun ;
Fun ::= `void f(` ArgList `)` `{` Body `}` ;
ArgList ::= Arg { `,` Arg } ;
Arg ::= `int` <ID> ;
Body ::= { BB IfThenElse } ;
BB ::= `{`  { Asgn }  `}` ;
Var ::= <ID> ;
Asgn ::= Var `=` Expr `;` ;
Expr ::= Var | <INT>
       | `(` Expr Op Expr `)`
       | CSE ;
CSE ::= `(` Expr Op Expr `)` ;
Op ::= `+` | `-` | `*` ;
IfThenElse ::= `if(` Cond `)` BB [ `else` BB ] ;
Cond ::= Var ( `==` | `>` | `<` ) Var ;
<ID> ::= $[A-Za-z_] { $[A-Za-z_0-9] } ;
<INT> ::= `0` | [`-`] $[1-9] { $[0-9] } ;
```

*Рис. 4. BNF-грамматика для оптимизатора CSE.*
*Fig. 4. BNF-grammar for the CSE optimizer.*

```
@@{ Int = `Na`;
    StrSet = `Args`;
  }@
...
Arg ::= `int`
     @.{ str = `+Na:a_%d`;
         addThisAsStr = `Args`;
     }@ <ID> ;
...
Var ::= @.{ choose = `Args`; }@ <ID> ;
...
```

*Рис. 5. Семантические коррекции BNF-грамматики для оптимизатора CSE.*
*Fig. 5. Semantic corrections of the BNF-grammar for the CSE optimizer.*

## 3.5 Настройка генерации общих подвыражений

Теперь, когда мы добились того, что генерируемые C-программы являются семантически корректными, нужно перейти к основной задаче: обеспечить, что генерируемые программы, будучи поданы на вход компилятору, заставят целевой модуль оптимизации CSE произвести





какие-то осмысленные действия для этой оптимизации. Для этого нужно так настроить генерацию тестов, чтобы генерируемые структуры содержали бы *общие* подвыражения.

Воспользуемся опытом обеспечения семантического правила «Использовать только ранее объявленные имена» и заставим генератор делать так:

1) предварительно отдельно генерировать пул выражений и сохранять их в виде строк в некотором специальном множестве;
2) в ходе генерации собственно теста в качестве раскрытия последней альтернативы правила `Expr` (напомним, что эта альтернатива состоит только в ссылке на нетерминал `CSE`) брать строку из заготовленного множества-пула выражений.

Второй пункт из этих двух мы реализовывать уже умеем. Объявим для грамматики множество `CSEs` и зададим в контексте элемента `CSE` в правой части правила для нетерминала `Expr` атрибут `choose = `CSEs`;`.

Остается лишь вопрос о том, как реализовать первый из этих пунктов.

### 3.5.1 Предварительная генерация

Основная проблема здесь заключается в том, что генератор работает по принципу раскрытия нетерминалов, но в нашей грамматике отсутствует нетерминал, в котором отдельно «перечисляются» будущие общие подвыражения для теста.

Для решения этой проблемы мы *добавим* в грамматику такой фиктивный нетерминал `_CSE_List_`, на который нигде в грамматике *не будет* ссылок, но зато внутри этого нетерминала будут генерироваться общие подвыражения (в соответствии с правилом `CSE`), и для сохранения их строковых представлений в множестве `CSEs`, мы зададим в контексте элемента `CSE` в правой части правила для нетерминала `_CSE_List_` атрибут `addThisAsStr = `CSEs`;`:

```
_CSE_List_ ::= { @.{ addThisAsStr = `CSEs`; }@ CSE } ;
```

Для того, чтобы заставить генератор использовать этот нетерминал для предварительной генерации, требуется в контексте BNF-выражения для нетерминала `Body` задать атрибут `preGen = `_CSE_List_`;`.

Соответствующие добавки в грамматику представлены на рис. 6.

```
@@{ Int = `Na`;
    StrSet = `Args, CSEs`;
  }@
...
Body ::= @*{ preGen = `_CSE_List_`; }@
         { BB IfThenElse } ;
_CSE_List_ ::= { @.{ addThisAsStr = `CSEs`; }@ CSE } ;
...
Expr ::= Var | <INT>
       | `(` Expr Op Expr `)`
       | @.{ choose = `CSEs`; }@ CSE ;
...
```

*Рис. 6. Настройка в BNF-грамматике генерации общих подвыражений.*
*Fig. 6. Configuring of the BNF-grammar to generate common subexpressions.*

Пример сгенерированного теста представлен на рис. 7.

Можно заметить, что в этом примере пул общих подвыражений содержит выражения:

- `( a_4 - ( ( ( a_4 - 817 ) * ( -109 - -751 ) ) + a_1 ) );`
- `( -907 * a_3 ).`

В целом по трудозатратам предложенный подход с использованием атрибутных грамматик показал превосходные результаты по сравнению с ручной разработкой генератора.

Разработка грамматики со всеми атрибутами для оптимизатора CSE заняла примерно один час.

## *4. Реализация подхода*

Предложенный в настоящей статье подход по автоматической генерация тестов для оптимизаторов на основе атрибутных грамматик реализован в виде программного комплекса на языке Haskell с использованием библиотеки Haskell QuickCheck.

Порядок работы с программным комплексом состоит из двух стадий:

1) на первой стадии программному комплексу на вход подается описание модели в виде атрибутной BNF-грамматики, и программный комплекс транслирует это описание в программу тоже на языке Haskell;
2) после компиляции этой программы ее можно запустить на исполнение, и в результате будут сгенерированы тесты.

```
void f( int a_1 , int a_2 , int a_3 , int a_4 )
{
  {
    a_2 = ( ( a_4 - ( ( ( a_4 - 817 ) * ( -109 - -751 ) ) + a_1 ) ) * a_4 );
    a_1 = a_2 ;
  }
  if( a_4 > a_2 ) {
    a_2 = ( a_4 - ( ( ( a_4 - 817 ) * ( -109 - -751 ) ) + a_1 ) ) ;
    a_1 = a_1 ;
  }
  else {
    a_2 = ( ( -907 * a_3 ) - a_4 ) ;
    a_4 = ( a_4 - ( ( ( a_4 - 817 ) * ( -109 - -751 ) ) + a_1 ) ) ;
  }
  {
    a_2 = 702 ;
    a_2 = ( a_4 - ( ( ( a_4 - 817 ) * ( -109 - -751 ) ) + a_1 ) ) ;
  }
  if( a_4 < a_4 ) {
    a_3 = 0 ;
    a_2 = ( ( -907 * a_3 ) * -726 ) ;
  }
  else {
    a_3 = 119 ;
    a_1 = a_1 ;
  }
}
```

*Рис. 7. Пример сгенерированного теста для CSE.*
*Fig. 7. Example of generated test for CSE.*

## *5. Заключение*

В статье описывается модельный подход к разработке тестов, нацеленных на тестирование оптимизирующих компиляторов.

Модель входных данных для целевого оптимизатора описывается с помощью атрибутной BNF-грамматики. Атрибуты позволяют лаконично задавать ограничения статической семантики. Приведены способы задания таких ограничений для нескольких широко распространенных категорий семантических требований. Показано, что такое задание семантических ограничений дает возможность генерировать семантически корректные тесты конструктивно, то есть получать корректный тест сразу непосредственно во время генерации.

Продемонстрированный пример, безусловно, не охватывает всех имеющихся возможностей подхода для задания семантических ограничений и призван лишь показать суть подхода.





Подход можно использовать для разработки тестов не только для оптимизирующих компиляторов, но и для генерации тестов сложной структуры для других видов ПО.

Подход активно развивается. В программный комплекс добавляются различные дополнительные возможности, в частности, постепенно расширяется перечень поддерживаемых категорий семантических ограничений.

Подход позволяет получать тесты без необходимости ручной разработки каких-либо компонентов генератора тестов. Генерация происходит полностью автоматически на основе лишь модели в виде атрибутной BNF-грамматики. Модель в виде атрибутной BNF-грамматики является по сути конфигурацией генератора тестов. С ней очень удобно работать: она крайне компактна, ее можно быстро разработать и легко модифицировать.

## Список литературы / References


[1]. Aho A., Sethi R., Ullman J.D. Compilers. Principles, Techniques, and Tools. Addison-Wesley Publishing Company, Inc. 1985.

[2]. Dewey K., Roesch J., Hardekopf B. Fuzzing the Rust Typechecker Using CLP (T). In Proceedings of the 2015 30th International Conference on Automated Software Engineering, 2015, pp. 482-493.

[3]. Holler Ch., Herzig K., Zeller A. Fuzzing with Code Fragments. In Proceedings of the 21st USENIX Conference on Security Symposium, 2012, pp. 445-458.

[4]. Yang X., Chen Y., Eide E., Regehr J. Finding and Understanding Bugs in C Compilers. In Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011, pp. 283–294.

[5]. Petrenko A.K. Specification Based Testing: Towards Practice. LNCS, 2001, 2244, pp. 287-300.

[6]. Bourdonov I.B., Kossatchev A.S., Kuliamin V.V., Petrenko A.K. UniTesK Test Suite Architecture. Proc. FME'2002 conference, LNCS, 2391. Copenhagen, Denmark, 2002, pp. 77-88.

[7]. Purdom P. A Sentence Generator for Testing Parsers. BIT, 1972, no. 2, pp. 336-375.

[8]. Maurer P.M. The design and implementation of a grammar-based data generator. Software Practice and Experience, 1992, vol. 22, no. 3, pp. 223-244.

[9]. Lämmel R. Grammar testing. Proc. of Fundamental Approaches Software Engineering, 2001, 2029, pp. 201-216.

[10]. Зеленов С.В., Зеленова С.А. Генерация позитивных и негативных тестов для парсеров. Программирование, Москва, 2005, т. 31, № 6, с. 25-40. / Zelenov S.V., Zelenova S.A. Generation of positive and negative tests for parsers. Program. Comput. Software, 2005, vol. 31, no. 6, pp. 310-320.

[11]. Петренко А.К. и др. Тестирование компиляторов на основе формальной модели языка. Препринт института прикладной математики им. М.В. Келдыша № 45, 1992.

[12]. Duncan A.G., Hutchison J.S. Using Attributed Grammars to Test Designs and Implementation. Proc. 5th international conference on Software engineering. Piscataway, NJ, USA. IEEE Press, 1981, pp. 170-178.

[13]. Harm J., Lämmel R. Two-dimensional Approximation Coverage. Informatica Journal, 2000, vol. 24, no. 3.

[14]. Paakki J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, 1995, vol. 27, no. 2, pp. 196-255.

[15]. Kalinov A., Kossatchev A., Posypkin M., Shishkov V. Using ASM Specification for automatic test suite generation for mpC parallel programming language compiler. Proc. Fourth International Workshop on Action Semantic, AS'2002, BRICS note series NS-02-8, 2002, pp. 99-109.

[16]. Gurevich Y. Abstract State Machines: An Overview of the Project. LNCS 2942, 2004, pp. 6-13.

[17]. Allen R., Kennedy K. Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, 2002.

[18]. Limpanukorn B., Wang J., Kang H.J., Zhou E.Z., Kim M. Fuzzing MLIR Compilers with Custom Mutation Synthesis. arXiv, 2024. DOI: 10.48550/arXiv.2404.16947 [cs.SE].

[19]. Wu J., Zheng J., Yang Zh., Yu Zh. Compiler Optimization Testing Based on Optimization-Guided Equivalence Transformations. FSE '25, June 23-28, 2025, Trondheim, Norway. arXiv, 2025. DOI: 10.48550/arXiv.2504.04321 [cs.SE].

[20]. Zhou Z., Limpanukorn B., Kang H.J., Wang J., Wu Y., Kiss A., Hodovan R., Kim M. Targeted Testing of Compiler Optimizations via Grammar-Level Composition Styles. arXiv, 2025. DOI: 10.48550/arXiv.2512.04344 [cs.SE].

[21]. Zelenov S., Zelenova S. Model-Based Testing of Optimizing Compilers. Proc. of the 19th IFIP TC6/WG6.1 International Conference on Testing of Software and Communicating Systems – 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007). LNCS 4581, Springer-Verlag, Berlin, 2007, pp. 365-377.

[22]. Зеленов С.В., Зеленова С.А., Косачев А.С., Петренко А.К. Генерация тестов для компиляторов и других текстовых процессоров. Программирование, Москва, 2003, т. 29, № 2, с. 59-69. / Zelenov S.V., Zelenova S.A., Kosachev A.S., Petrenko A.K. Test generation for compilers and other text processors. Program. Comput. Softw, vol. 29, no. 2, 2003, pp. 104-111.

## *Информация об авторах / Information about authors*

Сергей Вадимович ЗЕЛЕНОВ – кандидат физико-математических наук, ведущий научный сотрудник отдела технологий программирования Института системного программирования. Сфера научных интересов: тестирование на оснофе моделей и спецификаций, автоматическая генерация сложно-структурированных тестовых данных.

Sergey Vadimovich ZELENOV – Cand. Sci. (Phys.-Math.), Leading Researcher of the Software Engineering Department of the Institute for System Programming of the RAS. Research interests: model-based testing, specification-based testing, authomatical generation of test data with complex structure.