МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

TIGSTYLE: ДЕКЛАРАТИВНЫЙ СТИЛИСТИЧЕСКИЙ ЛИНТЕР ДЛЯ ЯЗЫКА ПРОГРАММИРОВАНИЯ TIGER

АВТОРЕФЕРАТ БАКАЛАВРСКОЙ РАБОТЫ

студента 4 курса 411 группы
направления 02.03.02 — Фундаментальная информатика и информационные технологии
факультета КНиИТ
Афанасьева Артёма Андреевича

Научный руководитель
доцент, к. ф.-м. н. _____ С. В. Миронов

Заведующий кафедрой
к. ф.-м. н., доцент _____ А. С. Иванов

Саратов 2020

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящее время языковые инструменты играют значительную роль в процессе разработки программного обеспечения (используются в IDE, сервисах хостинга проектов и т.д.). Особой популярностью пользуются линтеры — инструменты для проверки кода и поиска в нем различного рода ошибок и отклонений от принятых правил.

В данной работе рассматриваются стилистические линтеры. Их работа заключается в поиске и исправлении стилистических ошибок и отклонений в исходных текстах. Такие ошибки и отклонения могут присутствовать на разных уровнях программы. На текстовом уровне они представлены отсутствием должного форматирования, а на уровне синтаксиса — использованием нежелательных, либо сомнительных конструкций, которые могут быть заменены более удачными альтернативами.

К сожалению, большая часть стилистических линтеров предоставляет ограниченные возможности по определению собственного стиля оформления. Обычно они сводятся к включению/выключению предопределенных правил с помощью конфигурационного файла. Такой подход позволяет описывать лишь стили, которые можно представить в виде композиции имеющихся правил, так как определение собственных правил не предусмотрено.

Целью работы является разработка линтера tigstyle для языка Tiger, позволяющего задавать оформление декларативно: с помощью шаблонов форматирования и правил замены одних конструкций на другие. Такой подход позволяет использовать единый синтаксис для определения как встроенных, так и пользовательских правил.

Для достижения цели решались следующие задачи:

- Рассмотреть имеющиеся решения для форматирования и линтинга.
- Проанализировать существующие подходы к описанию стилистических требований.
- Разработать и реализовать алгоритм декларативного линтинга.
- Проанализировать полученные результаты.

Работа выполнена на 89 страницах машинописного текста; состоит из введения, 3 глав, заключения; содержит 20 рисунков и 11 приложений. Список литературных источников включает 45 наименований.

1 Теоретические сведения. Обзор предметной области

В первой главе содержатся теоретические сведения, необходимые для понимания работы линтера, а также обзор предметной области, завершающийся анализом преимуществ и недостатков существующих инструментов.

1.1 Оформление программ

Линтер имеет дело с двумя аспектами оформления: форматированием исходного кода и альтернативными синтаксическими конструкциями.

Первый аспект оформления — форматирование, под которым подразумевается взаимное расположение лексем исходного текста, определяемое типом, положением и количеством используемых пробельных символов. Ниже приведены 2 варианта форматирования фрагмента кода на языке C.

```
int main(){time_t t=time(NULL);struct tm*tm
=localtime(&t);switch(tm->tm_hour){case 6:
case 7:case 8:case 9:case 10:case 11:printf
("Good morning!\n");break;case 12:case 13:
case 14:case 15:case 16:case 17:printf("Go"
"od day!\n");break;default:printf("Good ni"
"ght!\n");break;}return 0;}
```

Рисунок 1 – Неформатированный код

Функция, приведенная на рисунке 1, синтаксически корректна и имеет простую логику, но ее восприятие затруднено плохим форматированием.

```
int main() {
    time_t t = time(NULL);
    struct tm* tm = localtime(&t);
    switch(tm->tm_hour) {
        case 6:case 7:case 8:
        case 9:case 10:case 11:
            printf("Good morning!\n");
            break;
        case 12:case 13:case 14:
        case 15:case 16:case 17:
            printf("Good day!\n");
            break;
        default:
            printf("Good night!\n");
            break;
    }
    return 0;
}
```

Рисунок 2 – Форматированный код

Функция, приведенная на рисунке 2, с точки зрения компилятора не отличается от предыдущей, однако человеку такой вариант форматирования проще понимать и модифицировать.

Второй аспект оформления — используемые в программе конструкции. Некоторые идеи в программировании можно представить различными способами. Например, бесконечный цикл в C можно оформить как с помощью конструкции `while`, так и с помощью конструкции `for` (см. рисунок 3). При этом, в случае использования `while`, в качестве условия окончания цикла может быть любое истинное выражение.

```
while (<true_constant>) {
  <body>
}
```

(a) С помощью конструкции `while`

```
for (;;) {
  <body>
}
```

(b) С помощью конструкции `for`

Рисунок 3 – Варианты оформления бесконечного цикла на языке C

В рассмотренном примере нет никаких причин отдавать предпочтение одной из конструкций перед другой. Выбор определяется лишь привычкой и вкусом программиста.

Однако существуют случаи, когда из нескольких альтернативных представлений одним следует отдавать предпочтение, а других — избегать. Например, если используется сложная для понимания конструкция, которая может быть заменена более простой. Использование таких нежелательных конструкции можно считать ошибкой в оформлении программы.

Ошибки такого рода часто встречаются в коде начинающих программистов. К ним, например, относится неоправданное использование конструкции `else if` вместо `else` (см. рисунок 4).

```
if (<cond> == false) {
  <body-1>
} else if (<cond> == true) {
  <body-2>
}
```

(a) Нежелательный вариант

```
if (<cond>) {
  <body-2>
} else {
  <body-1>
}
```

(b) Предпочтительный вариант

Рисунок 4 – Использование `else if` вместо `else`

1.2 Обзор предметной области

Линтерами называют языковые инструменты, предназначенные для нахождения любых отклонений в программах, о которых не сообщает компилятор. Они уходят своими корнями к утилите lint, разработанной Стивеном Джонсоном в 1978 году в Bell Labs для поиска фрагментов кода в программах на C, вызывающих потенциальные проблемы с переносимостью.

С момента появления lint сфера применения линтеров значительно расширилась. Современные линтеры поддерживают проверки на наличие синтаксических ошибок, использование неопределенных переменных, нарушение стилевых правил и многие другие потенциальные проблемы. Также, в отличие от lint, они способны трансформировать исходный текст, исправляя нежелательные фрагменты кода.

В данной работе рассматриваются стилистические линтеры. Их работа заключается в поиске и исправлении ошибок в оформлении программ. Эти инструменты работают с кодом как на текстовом уровне, так и на структурном.

На текстовом уровне выполняются проверки, связанные с отдельными символами (или группами символов). Например, здесь могут проверяться длины строк, наличие табуляций, перевода строки в конце файла и т. п. На структурном уровне проверяется структура программ. Например для языка C, здесь можно осуществить проверку наличия оператора `break` внутри каждого блока `case`, или блока `default` в конце конструкции `switch`.

На структурном уровне также же возможны проверки, связанные с наличием нежелательных конструкций. Так `hlint`, линтер для языка Haskell, имеет ряд правил, связанных с упрощением конструкций, за счет использования более специализированных функций. Например, фрагмент кода `concat (map escapeC s)` `hlint` предложит заменить на `concatMap escapeC s`. А линтер для языка C++ `clang-tidy` позволяет заменять циклы `for`, в которых используется переменная-итератор на эквивалентные диапазонные циклы из C++11.

В рамках работы был проведен обзор популярных линтеров для различных языков программирования. Рассматривались способы конфигурации, предоставляемые линтерами. Под конфигурацией имеется ввиду включение/выключение правил, добавление новых (собственных) правил.

Подборка линтеров основана на списке из репозитория Awesome Linters. Представленные там линтеры были отсортированы по количеству звезд их

github-репозитория. После этого были выбраны 5 самых популярных — по одному для каждого языка: `standard` для JavaScript ($23,8$ тыс. звезд), `swiftlint` для Swift ($13,5$ тыс. звезд), `rubocop` для Ruby ($10,8$ тыс. звезд), `checkstyle` для Java ($5,3$ тыс. звезд) и `pycodestyle` для Python ($4,1$ тыс. звезд).

Было замечено, что в подборке чаще всего (в 4 из 5 линтеров) встречается настройка с помощью конфигурационных файлов. Этот способ достаточно прост и предоставляет некоторую гибкость за счет параметризации правил, однако не позволяет пользователю добавлять собственные правила. Также, во всех линтерах помимо `swiftlint` добавление собственных правил возможно только путем расширения инструмента. Для пользователей, не имеющих опыта работы с представлением программ в виде синтаксических деревьев это может оказаться значительным препятствием. `swiftlint` позволяет описывать собственные правила с помощью регулярных выражений, но область применения такого подхода ограничена простыми конструкциями.

На основе анализа было сделано заключение о том что, рассмотренные линтеры не предоставляют достаточно гибкого и в тоже время простого способа определения стилистических правил. Данная работа нацелена на решение этой проблемы.

1.3 Язык Tiger

В качестве целевого языка для tigstyle был выбран модельный язык Tiger из серии книг Эндрю Аппеля «Modern Compiler Implementation».

По описанию Аппеля, Tiger — это простой, но не тривиальный императивный Algol-подобный язык программирования, с вложенными областями видимости и записями. Tiger поддерживает целочисленные и строковые переменные, массивы, записи; вложенные описания типов, переменных и функций. По синтаксису Tiger напоминает функциональные языки.

Ниже приведена простая Hello World программа на Tiger, использующая функцию для вывода строки:

```
1 let function hello() = print("Hello, World!\n")
2  in hello() end
```

Более подробное описание языка доступно в книгах Аппеля, а также на сайте исследовательского института LRDE.

2 Реализация

Вторая глава начинается с пояснения задачи и постановки требований к инструменту. Далее идет описание используемых в работе технологий, а также применяемых структур данных. За ним следует обзор архитектуры инструмента и писание отдельных частей программы и этапов ее работы.

Работа инструмента может осуществляться в двух режимах: отчета и интерактивном. В режиме отчета линтер проверяет всю программу и возвращает список найденных ошибок. В интерактивном режиме линтер последовательно проверяет узлы программы. При нахождении ошибки пользователю предлагается ее исправить. Если пользователь соглашается, исправленный код сохраняется в выходной файл, а линтер переходит к следующему узлу.

Линтер проверяет ошибки двух видов: связанные с форматированием либо с возможностью применения правила замены. Правила форматирования и замены определяются в шаблонном файле на специальном языке.

В основе правил лежат шаблоны — выражения на языке Tiger, в которые вместо подвыражений могут входить специальные индентификаторы — переменные захвата. Списочные переменные начинаются с символа `@` и используются вместо списочных конструкций. Скалярные переменные начинаются с символа `$` и используются вместо несписочных конструкций.

Правило форматирования представляет шаблон, окруженный парными скобками `[[`, `]]` и не содержащий вложенных подвыражений (только переменные захвата). Пример правила форматирования приведен на рисунке 5.

```
/* exp:let */
[[let @ in @ end]]
[[let @
  in @
  end]]
```

Рисунок 5 – Правило форматирования для конструкции let-in-end

Правило замены состоит двух частей (левой и правой), которые связаны символом >. Части правила — шаблоны, окруженные парными скобками `[`, `]` . Пример правила замены приведен на рисунке 6.

```
/* redundant parenthesis */
[if ($1) then $2 else $3] > [if $1 then $2 else $3]
```

Рисунок 6 – Правило замены, избавляющееся от ненужных скобок

Также существуют ошибки форматирования, которые невозможно описать используя шаблоны. Они связаны с конструкциями списочного вида:

— последовательностями определений (переменных, типов, функций);
— параметров и аргументов функций;
— аргументов конструкторов для записей, кортежей, списков;
— цепочек бинарных операторов.

Такие конструкции обрабатываются по предопределенному алгоритму. tigstyle не позволяет задать их стиль декларативно.

Работа линтера связана с обработкой языковых конструкций, извлеченных из исходного текста на этапе разбора. В связи с этим структуры данных, используемые для хранения этих конструкций, являются одним из самых важных элементов линтера. Речь идет о синтаксических деревьях, представляющих узлы программы, и токенах.

Для хранения токенов был выбран двусвязный список. Это структура данных позволяет эффективно осуществлять обход, вставлять и удалять отдельные элементы и их цепочки (для осуществления замен в исходном тексте).

Для представления узлов синтаксического дерева используется единственный класс (вместо альтернативы, когда используется отдельный класс для каждого типа узла). Каждый узел содержит список с его дочерними узлами. Для большинства конструкций языка длина списка фиксирована и заранее известна. Для списочных конструкций длина является произвольной.

Дочерними узлами могут быть как конструкции (поддеревья) так и отдельные токены (листовые узлы). В том числе, в качестве дочерних узлов могут выступать не значимые с точки зрения семантики токены вроде разделителей (`;`, `:`, ...), скобок (`{}`, `()`, ...) и даже комментариев. Для хранения комментариев у каждого внутреннего узла дерева имеется 3 списка: предшествующих, последующих и внутренних комментариев. Прикрепление комментариев к узлам происходит на отдельной стадии постобработки. В грамматику языка они не входят.

Такие деревья, содержащие токены и комментарии, называются Lossless Syntax Tree, что можно перевести на русский как «синтаксическое дерево без потерь». Под потерями имеется в виду утрата какой-либо значимой с точки зрения человека (а не компилятора) информации о коде: форматировании, комментариях и т.п.

Работа инструмента происходит в несколько этапов. Сначала осуществляется разбор исходных текстов входной программы и файла с правилами. Затем следует этап постобработки: поиск семантических ошибок в правилах и присоединение комментариев к синтаксическому дереву программы. Далее выполняется линтинг: поиск ошибок в оформлении комментариев, форматирование конструкций; поиск кандидатов для применения замен. Если используется режим отчета, найденные ошибки выводятся к консоль. Если используется интерактивный режим, поиск ошибок чередуется с запросами к пользователю и исправлением найденных ошибок.

Код инструмента разделен на 4 модуля: `tok`, `ast`, `frontend`, `linting`.

Модуль `tok` определяет классы `Token` и `Position`. Класс `Token` описывает токены, возвращаемые лексическим анализатором. Каждый токен хранит свою начальную и конечную позиции в исходном тексте. Эти позиции представлены экземплярами класса `Position`. Помимо описанных классов модуль предоставляет функции для работы с токенами. Большая их часть связана либо с манипуляцией двусвзяным списком, либо с изменением позиций токенов.

Модуль `ast` содержит определение класса `Node`, представляющего собой узел дерева разбора программы, а также большой набор вспомогательных функций для обхода, перемещения, клонирования, печати и выполнения других действий над узлами дерева.

Модуль `frontend` отвечает за разбор входных файлов: как исходных текстов на языке Tiger, так и файлов с правилами. `frontend`, в свою очередь, содержит подмодули `lexing` и `parsing`, отвечающие за лексический и синтаксический разбор исходных текстов соответственно. Помимо описанных подмодулей во `frontend` также входит код, выполняющий валидацию правил, после успешного их разбора.

Наконец, модуль `linting` содержит весь код, связанный с линтингом. Здесь определены классы `Linter`, `BaseLinter`, `CommentLinter`, `FormatLinter`, `ReplacementLinter`, `ListLinter` и `BinopLinter`, а также класс `LintingError`.

Процесс линтинга разбит на несколько частей-подлинтеров. Общие элементы вынесены в класс `BaseLinter`, а внешний интерфейс предоставляется агрегирующим классом `Linter`. `Linter` содержит экземпляры всех подлинтеров и вызывает их по мере необходимости при проверке исходного текста.

Класс `BaseLinter` определяет два абстрактных метода: `check` и `fix`.

Метод `check` принимает на вход узел синтаксического дерева `node` и набор опциональных параметров (настроек) `kwargs`, контролирующих требования линтера. Метод возвращает список экземпляров класса `LintingError` (класс описан дальше). Если ошибок нет, список должен быть пустым.

Метод `fix` принимает на вход код ошибки `error_code`, узел синтаксического дерева, с которым связана ошибка `node`, словарь `data`, содержащий дополнительную информацию об ошибке, и набор настроек `kwargs`. Метод исправляет синтаксическое дерево так, чтобы оно отвечало предъявляемым требованиям.

Возвращаемые линтером ошибки — экземпляры класса `LintingError`. Класс содержит следующие поля:

— `code` — код ошибки. Это строка-идентификатор для ошибок одного вида.
— `node` — узел, в котором была найдена ошибка.
— `pos` — позиция в тексте, где была найдена ошибка.
— `data` — словарь, содержащий дополнительные сведения об ошибке.

Первый из цепочки вызываемых линтеров — `CommentLinter`. Данный линтер выполняет проверки, связанные с комментариями. Он может удалять любой тип комментариев, если они запрещены, и проверять форматирование предшествующих комментариев (если они разрешены).

Линтер использует следующие коды ошибок:

— `comment:illegal:preceding` — запрещенный предшествующий комментарий.
— `comment:illegal:enclosed` — запрещенный внутренний комментарий.
— `comment:illegal:trailing` — запрещенный последующий комментарий.
— `comment:preceding:line` — ошибка в форматировании предшествующего комментария. Комментарий расположен на той же строке, что и узел, к которому он относится
— `comment:preceding:block` — ошибка в форматировании предшествующего комментария. Комментарий расположен на строке перед узлом и не выровнен с началом узла.

Далее вызывается `ReplacementLinter`. Его метод `check` проверяет возможность применения правила замены к узлу, а метод `fix` осуществляет применение правила. `ReplacementLinter` использует единственный код ошибки: `replacement`.

Третьим по счету является `FormatLinter`. Он выполняет проверку на соответствие форматирования узла форматированию, заданному в файле с правилами. Полный код класса доступен в приложении. `FormatLinter` использует единственный код ошибки: `format`.

Следующим вызывается `ListLinter`. Он проверяет и исправляет форматирование узлов-списков.

Линтер использует следующие коды ошибок:

— `list:dec:same_line` — два определения находятся на одной строке.
— `list:collapse` — найден блок пустых строк (возможно схлопывание).
— `list:empty_line` — найдена пустая строка.
— `list:align` — первый элемент списка, расположенный на новой строке, не выровнен по голове списка.
— `list:separator:adjacent` — найден разделитель, отделенный от последнего элемента одним или более пробелами.
— `list:separator:space` — найден элемент списка, не отделенный от последнего разделителя единственным пробелом.
— `list:width` — список пересекает максимальную длину строки.

Последним управление получает `BinopLinter`. Он проверяет и исправляет форматирование цепочек бинарных операторов. `BinopLinter` использует единственный код ошибки: `binop:width`. Ошибка возникает, когда цепочка операторов пересекает границу максимальной длины строки.

3 Результаты

В третей главе приводится пробный запуск линтера, рассматриваются преимущества и недостатки разработанного инструмента и декларативного подхода к линтингу в целом.

Ниже приводится урезанный вариант отчета о запуске программы (опущен диалог с пользователем).

Файл с правилами имеет следующий вид:

```
[[let @
  in @
  end]]
[[function $(@) =
    $]]
[[$ + $]]

[if $a := $b then $1 else $2] > [if $a = $b then $1 else $2]
```

Исходная программа состоит из единственного выражения `let-in-end`:

```
let
   /* a comment */
  function succ (a:int)
    = a +1
  var a : int := 1
in print(if a := succ(0) then "success" else "failure")
end
```

Результатом запуска утилиты является файл со следующим содержимым:

```
let /* a comment */
    function succ(a:int) =
      a + 1
    var a : int := 1
in print(if a = succ(0) then "success" else "failure")
end
```

Разработанный инструмент успешно справляется с проверкой оформления исходных текстов и в случае необходимости может вносить исправления.

Декларативный подход хорошо подходит для проверки простых конструкций языка и по сравнению со стандартными методами имеет преимущества в виде большой гибкости и простоты определения используемого оформ-

ления. С другой стороны, к более сложным конструкциям, оформление которых зависит от различных условий (например, длины списочных выражений), декларативный подход применим лишь частично. Для них могут быть определены некоторые элементы форматирования, но не все правило целиком.

В связи с этим правила оформления для сложных или плохо описываемых элементов (комментариев, списков, цепочек бинарных операторов) определяются в исходном коде и не подлежат изменению со стороны пользователя. Поэтому tigstyle нельзя назвать полностью декларативным линтером.

Для решения проблем, связанных со сложными конструкциями, возможно введение в язык условных правил, использующих различные варианты оформления в зависимости от контекста (тип родительского узла, ширина форматируемой конструкций и т. п.). Однако описание бесконечных конструкций (списков и цепочек операторов) все равно представляется проблематичным.

Особенные трудности представляет декларативное описание форматирования комментариев. Они могут встречаться в любом месте внутри конструкции, поэтому описание всех возможных положений потребует чрезмерно большого количества правил. Декларативных подход, в том виде, в котором он описан в рамках данной работы, для этой задачи не подходит.

Несмотря на все рассмотренные недостатки декларативного подхода, его явным преимуществом является простота и гибкость описания оформления для отдельного класса конструкций. В связи с этим представляется возможным использовать его как часть более сложного алгоритма.

ЗАКЛЮЧЕНИЕ

В рамках работы был разработан декларативный стилистический линтер tigstyle для языка программирования Tiger. На вход утилита принимает исходный текст программы на языке Tiger и набор правил, декларативно описывающих разрешенные варианты форматирования и возможные замены. Результатом ее работы в зависимости от выбранного режима является список найденных ошибок и, возможно, исправленный код.

Утилита имеет некоторые недостатки, которые были рассмотрены в рамках работы, но в целом успешно справляется с поставленной задачей и в отдельных случаях имеет ряд преимуществ перед альтернативами.