Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
Высшего профессионального образования

Московский государственный технический университет имени Н.Э. Баумана

Расчетно-пояснительная записка
к дипломному проекту
НА ТЕМУ:

Проверка корректности расстановки отступов в программе на языке С.

Студент группы ИУ9-82
_____ Разборщикова Анастасия Викторовна
«___» _____ 2019 г.
Преподаватель
_____ Скоробогатов Сергей Юрьевич
«___» _____ 2019 г.

Москва
2019

АННОТАЦИЯ

Работа посвящена созданию программы, выполняющей проверку корректности расстановки отступов в файлах исходных текстов программ на языке C. Разработанное приложение должно принимать на вход файл с исходным кодом и выдавать сообщения об ошибках расстановки отступов. Программа разрабатывается для использования в учебных целях как часть автоматической системы тестирования.

Проведено исследование существующих программ-аналогов, изучено их внутреннее устройство, достоинства и недостатки. На основе полученных данных разработан алгоритм, осуществляющий проверку исходного кода на предмет корректности расстановки отступов в рамках установленных требований. Произведено тестирование на более чем 1000 реальных документов.

Расчетно-пояснительная записка состоит из 10 разделов, содержит 63 страницы и включает в себя 4 таблицы, 22 листинга и 9 рисунков и 1 приложение. В разделах «ВВЕДЕНИЕ» и «Постановка задачи» содержится описание решаемой проблемы, обоснование ее актуальности и определение критериев корректности решения данной задачи. Раздел «Обзор предметной области» содержит описание существующих решений. В разделе «Разработка приложения» приведен алгоритм решения поставленной задачи. В главе «Руководство пользователя» приведена инструкция по установке и использованию разработанной программы. Раздел «Тестирование» содержит примеры входных данных и разбор вывода приложения. В главе «ЗАКЛЮЧЕНИЕ» содержится краткое описание проделанной работы и обозначены направления дальнейшего улучшения приложения. Глава «СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ» содержит описание источников и ссылки, которые упоминаются в данной работе. В «ПРИЛОЖЕНИИ А» находятся листинги, не вошедшие в основной текст записки.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

При разработке программ необходимо уделять внимание не только логическому проектированию программы и ее коррекности, но и стилю оформления кода (coding style). Этой проблеме посвящаются книги и научные статьи. Наименование переменых, длина строк, использование шаблонов (patterns) проектирования, — все это напрямую влияет на понимание программы, возможность распространять ее и избежать проблем с дальнейшей поддержкой, ведь большую часть вренеми при разработке занимает именно чтение кода [1]. Были проведены исследования, доказывающие, что форматирование (стиль расстановки скобок, пробелы и т.п.) влияет на восприятие программы [2].

На сегодняшний день разработано множество стандартов оформления кода, для многих крупных проектов разрабатываются собственные соглашения (например [3], [4]). Однако студенты, только начавшие изучать программирование, часто не знакомы с ними, либо пренебрегают этими правилами и не стремятся к аккуратному оформлению кода.

На кафедре ИУ9 «Теоретическая информатика и компьютерные технологии» факультета «Информатики и систем управления» МГТУ им. Н. Э. Баумана для проверки работ студентов используется автоматическая система тестирования. После проверки программы на корректность и заимствования, она попадает на проверку преподавателю. На этом этапе встает вопрос о читаемости кода, адекватности его оформления. В связи с этим возникла необходимость разработки инструмента, который бы также проверял код на соответствие некоторым правилам форматирования и выдавал студентам сообщения об ошибках, фильтруя поток так, чтобы преподаватель мог проверять только корректно отформатированный код. Сложность задачи состоит в том, чтобы не делать условия корректности слишком жесткими (многие среды разработки программ автоматически форматируют код в соответствии со своими стандартами, которые могут отличаться, при этом являясь корректными) и поддерживать все возможные расширения компилятора GCC (the GNU Compiler Collection), установленного на сервере тестирования.

1 Постановка задачи

В рамках выпускной квалификационной работы ставится задача реализовать программу для проверки корректности расстановки отступов в программах на языке C.

Приложение должно выявлять *плохо отформатированные* программы и выдавать описание найденных ошибок.

***Определение* 1.** *В рамках данной задачи исходный код программы считается плохо отформатированным, если:*

1. *На одном уровне вложенности строки имеют разный отступ (за исключением строк, которые являются продолжением выражения, а также меток, включая метки* `case` *и* `default` *в теле оператора* `switch`*).*
2. *Строки кода на большем уровне вложенности располагаются левее кода на меньшем уровне вложенности. Это правило не относится к меткам (за исключением меток* `case` *и* `default`*, которые должны быть не левее объявления блока* `switch`*) и директивам препроцессора, так как они могут не иметь отступа.*
3. *Строка, являющаяся продолжением утверждения (перенос), располагается не левее предыдущей строки. «Утверждением» в данной задаче считается одна из следующих конструкций: выражения, отделенные точкой с запятой (операции присваивания, вызов функции, объявления переменных и полей структур struct и объединений* `union`*); тело объявления перечисления* `enum`*; имя метки с двоеточием; имя оператора* `if`*,* `while`*,* `for`*,* `switch` *вместе со следующим за ним выражением в скобках.*

Основные требования к программе:

1. Программа должна быть реализована в виде консольного приложения, которой подается на вход исходный код программы на языке C.
2. Программа должна выдавать сообщения об обнаруженных ошибках расстановки отступов.
3. Должна быть реализована возможность управлять тем, насколько подробно выводятся сообщения (т.е. поддерживать различные режимы диагностики).

4. Программа не должна реализовывать слишком жесткие требования к оформлению кода.
5. Программа должна запускаться на платформе GNU/Linux.
6. Должна быть реализована поддержка использования любых расширений компилятора GCC, которые могут использоваться в исходных кодах программ.

Пример, иллюстрирующий различия между кодом с допустимым форматированием и плохо отформатированным, приведен на рисунке 1.

```
1  void ok(void) {
2      int x = 1,
3      n = 5, i;
4      for(i = 0;
5          i < n; i++) {
6        x += x;
7      }
8      if (x % 2 == 0
9           || x > 0) {
10       printf(”Ok!\n”);
11     }
12 }
```

а) Допустимое форматирование.

```
1  void bad(void) {
2      int x = 1,
3    n = 5, i;
4      for(i = 0;
5          i < n; i++) {
6        x += x;
7      }
8        if (x % 2 == 0
9           || x > 0) {
10    printf(”Bad!\n”);
11     }
12 }
```

б) *Плохое* форматирование.

Рисунок 1 — Примеры кода с допустим и плохим форматированием.

Приложение должно стать частью автоматической системы тестирования кафедры «Прикладная математика и компьютерные технологии» и выполнять проверку исходного кода программы на предмет корректности расстановки отступов после прохождения этапов компиляции и тестирования перед отправкой текста преподавателю.

2 Обзор предметной области

2.1 Обработка исходных текстов программ

Для анализа текста его нужно обработать, чтобы получить некоторое представление о его структуре. Для решения разных задач (от преобразования текста программы в исполняемый файл до обработки естественного языка) используются одни и те же подходы.

Введем некоторые понятия, которые будут использоваться в дальнейшем.

***Определение* 2.** *Компиляция (compilation) — процесс перевода (translation) кода с исходного языка программирования в язык, пригодный для выполнения на ЭВМ [5, с. 4].*

Программа, выполняющая компиляцию, называется компилятором.

Как правило, компиляция состоит из двух стадий: анализ (фронтенд, front-end) и синтез (бэкенд, back-end). Нас интересует первая стадия.

***Определение* 3.** *Фронтенд (front-end) — стадия компиляции, производящая синтаксический анализ исходного кода: разбиение текста на значимые единицы и построение по некоторым правилам (грамматике языка) внутреннего промежуточного представления программы (intermediate representation) [5, с. 41].*

Стадия анализа, как правило, состоит из нескольких стадий (см. рисунок 2) [5, с. 41]:

1. Лексический анализ.
2. Синтаксический анализ.
3. Генерация внутреннего представления.
4. Статический анализ (не обязательно).

В процессе *лексического анализа* программа разбивается на *лексемы* (lexem) — лексически значимые единицы. Каждая лексема может быть отнесена к некоторому абстрактному классу (идентификаторы, числовые константы, строки, символы арифметических операций и т.п.). Результатом лексического анализа является последовательность *токенов* (token) — структур, состоящих из лексемы и имени ее абстрактного класса [5, с. 43].

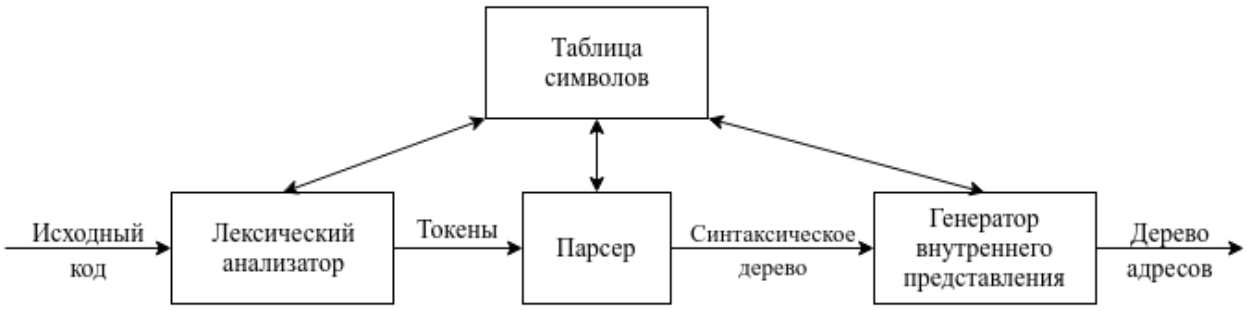

Рисунок 2 — Схема стадии анализа.

Подпрограмма, выполняющая лексический анализ, называется *лексер* (lexer) или *сканер* (scanner).

В результате *синтаксического анализа* строится синтаксическое дерево, в узлах которого располагаются операции, а в их потомках — операнды [5, с. 69].

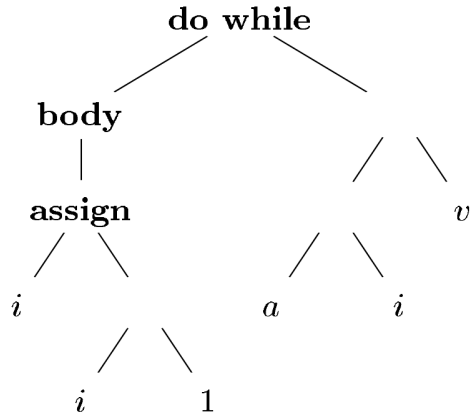

Рисунок 3 — Пример синтаксического дерева (изображение из книги [5, с. 41]).

Подпрограмма, выполняющая синтаксического анализ, называется *парсер* (parser).

***Определение* 4.** *Статический анализ (англ. static analisis) — любой процесс проверки исходного кода без его выполнения [6, с. 3].*

Статический анализ входных данных может быть разделен на две части: *контроль синтаксиса* (syntax checks), во время которого выполняется проверка форматирования кода и *контроль семантики* (semantics checks), определяющий соответствие ввода логике и функциональности приложения [6, с. 120].

В дальнейшем будут рассматриваться статические синтаксические анализаторы — программы и инструменты, выполняющие статическую проверку синтаксиса.

***Определение* 5.** *Линтер (lint, linter), анализатор типа lint (lint-like tool, lint-like checker) — статический анализатор кода. Изначально lint — приложение для статического анализа программ, написанных на языке C, созданное еще в 1988 году [7]. Сегодня же это название часто используется в отношении анализаторов, выполняющих только проверку стиля кода [6, с. 120].*

На сегодняшний день существует много программ для форматирования исходного кода программ на языке C:

1. Системы автоформатирования и статические анализаторы, встроенные в интегрированные среды разработки (IDE, Integrated development environment), такие как CLion [8], QtCreator [9], Eclipse [10] и проч.
2. Автоформатирование в текстовых редакторах, например Sublime Text [11], Notepad++ [12], Kate [13]. Редактор Vim [14] имеет как встроенные решения для автоматической расстановки отступов [15], так и сторонние скрипты для автоформатирования, например vim-autoformat [16] и vim-clang-format [17].

   Инструменты, встроенные в текстовые редакторы и среды разработки, позволяют форматировать текст уже в процессе набора кода.
3. Самостоятельные системы автоформатирования кода (с поддержкой языка C): clang-format [18], Artistic Styler [19], Uncrustify [20], BCPP [21], GC GreatCode [22] и инструмент для преобразования отступов GNU Indent [23]. Эти программы получают на вход файл с исходным кодом и возвращают файл с текстом, преобразованным в соответствии с заданным стилем.
4. Статические синтаксические анализаторы(в этом списке только анализаторы, доступные для запуска на ОС GNU/Linux и поддерживающие языки C/C++): clang-tidy [24] и Clang Static Analyzer [25], cpplint [26], Cppcheck [27], Splint [28]. Эти инструменты анализируют исходный код программы и выдают сообщения об ошибках (некоторые, в том числе, об ошибках форматирования).

Несмотря на обширный список программ с похожей функциональностью, на момент написания данной работы программы, решающей поставленную задачу, найдено не было.

Напомним, что в данной работе ставится задача проверки файла исходного текста программы на единообразие оформления строк в соответствии с

некоторыми критериями: приложение должно получать на вход файл исходного кода программы на языке C и выдавать сообщения об ошибках расстановки отступов. То есть приложение не изменяет полученный файл и не требует соответствия определенному стилю оформления кода.

Очевидно, для этой задачи не подходят системы автоформатирования, так как они преобразуют файл в соответствии с некоторым стилем, не выдавая сообщений об ошибках форматирования. Тем не менее, организация таких систем будет рассмотрена в дальнейшем и использована в разработке программы проверки корректности расстановки отступов, так как эта задача относится к области форматирования кода.

Однако на данном этапе наибольший интерес для нас представляют статические синтаксические анализаторы, так как они могут предоставить требуемую функциональность. Проведем тестирование существующих программ для выявления инструментов, обладающих необходимой функциональностью.

Рассматриваемые анализаторы поддерживают языки C/C++. Создадим небольшую тестовую программу `dummy.c`, содержащую явные ошибки форматирования (листинг 1) и проверим, выявят ли их указанные анализаторы.

Листинг 1 — Текст программы `dummy.c`, содержащий ошибку форматирования.

```
 1 int main(){
 2
 3 int a = 1;      // Ошибка: отступ отличается на одном и том же
 4     if (a ==2)  // уровне вложенности
 5     {
 6  return 2;      // Ошибка: вложенное выражение расположено левее
 7     }           // объемлющего
 8     if (1)
 9         if (2) return 2;  // Ошибка: ветвь else относится
10     else                  // к вложенному оператору if
11         return 3;
12     return 0;
13 }
```

Данная программа содержит 3 ошибки первого типа (см. стр. 44):

1. Строка 4 расположена на том же уровне вложенности, что и строка 3, но имеет больший отступ.
2. Строка 6 расположена по отношению к строке 5 на большем уровне вложенности, однако имеет меньший отступ.
3. В строке 10 оператор `else` находится на одном уровне вложенности с оператором `if` в строке 9, а не в строке 8, следовательно, должен иметь

от же отступ. Пример с такой ошибкой добавлен потому, что в данном случае неправильное восприятие кода может привести к неожиданным результатам выполнения такой программы, и поэтому многие статистические анализаторы содержат инструменты для выявления подобных случаев (потенциальных ошибок).

2.2 clang-tidy и Clang Static Analyzer

Многие из перечисленных инструментов (clang-format, clang-tidy и Clang Static Analyzer, vim-clang-format) основаны на Clang — проекте, предоставляющем фронтенд (см. определение 3 на с. 8) и инструменты для обработки языков семейства C (включая C/C++) [29].

2.2.1. clang-tidy

Clang-tidy — анализатор типа lint для диагностики и исправления таких ошибок в программах, как нарушение стиля и некорректное использование интерфейса, а так же ошибок, которые могут быть обнаружены с помощью статического нализа [24]. Это консольное приложение, которое принимает на вход файл исходного текста на языке C++ и возвращает сообщения об обнаруженных ошибках. Полученные в результате анализа исправления можно сохранить в некотором файле (если указан ключ `-export-fixes=<filename>` при запуске), а потом автоматически применить предлагаемые изменения к входному файлу с помощью инструмента `clang-apply-replacements`; либо применить правки сразу, используя опции `fix` и `fix-errors`.

Запустим clang-tidy с максимальным набором проверяемых параметров, задав параметр `-checks=*`. Результат запуска приведен в листинге 2.

Как видно из вывода приложения, из трех ошибок форматирования анализатор обнаружил только одну: разный отступ в ветвях оператора `if-else` в строке 10. Этой ошибке посвящены два сообщения. Приведем их описание:

– «`/home/nastasia/Diploma/dummy.c:10:5: предупреждение: различные отступы для 'if' и соответствующего ему 'else'`»;
– «`/home/nastasia/Diploma/dummy.c:10:9: предупреждение: утверждение должно быть обрамлено скобками`»;

Листинг 2 — Результат запуска программы clang-tidy.

```
 > clang-tidy -checks=* dummy.c
Error while trying to load a compilation database:
Could not auto-detect compilation database for file "dummy.c"
No compilation database found in /home/nastasia/Diploma or any parent directory
fixed-compilation-database: Error while opening fixed database: No such file or directory
json-compilation-database: Error while opening JSON database: No such file or directory
Running without flags.
15 warnings generated.
/home/nastasia/Diploma/dummy.c:8:9: warning: converting integer literal to bool, use bool literal instead [modernize-use-bool-literals]
    if (1)
        ^
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:8:9: warning: implicit conversion 'int' -> bool [readability-implicit-bool-conversion]
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:8:11: warning: statement should be inside braces [google-readability-braces-around-statements]
    if (1)
          ^
           {
/home/nastasia/Diploma/dummy.c:9:13: warning: converting integer literal to bool, use bool literal instead [modernize-use-bool-literals]
        if (2) return 2;  // Ошибка: ветвь else относится
            ^
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:9:13: warning: implicit conversion 'int' -> bool [readability-implicit-bool-conversion]
note: this fix will not be applied because it overlaps with another fix
/home/nastasia/Diploma/dummy.c:9:15: warning: statement should be inside braces [google-readability-braces-around-statements]
        if (2) return 2;  // Ошибка: ветвь else относится
               ^
                {
/home/nastasia/Diploma/dummy.c:10:5: warning: add explicit braces to avoid dangling else [clang-diagnostic-dangling-else]
    else                  // к вложенному оператору if
    ^
/home/nastasia/Diploma/dummy.c:10:5: warning: different indentation for 'if' and corresponding 'else' [readability-misleading-indentation]
/home/nastasia/Diploma/dummy.c:10:9: warning: statement should be inside braces [google-readability-braces-around-statements]
    else                  // к вложенному оператору if
        ^
         {
```

Открыв список проверяемых условий (ckecks) [30], обнаруживаем, что среди них только одно правило `readability-misleading-indentation` относится к проверке отступов, и оно задает ограничения только на оформление оператора `if-else`: ветви одного условия должны иметь одинаковый отступ, и тела блоков должны выделяться фигурными скобками. Сообщения об этих

ошибках мы видим в выводе приложения, другие же ошибки отступов были проигнорированы.

Clang-tidy имеет очень гибкие настройки стиля форматирования. Эта утилита использует те же настройки форматирования, что и clang-format. Настройки можно передавать, например, используя файл `./clang-format` (программа будет выполнять поиск файла с таким именем в ближайшей родительской директории) или в виде строки в формате JSON. Подробное описание ключей для задания стиля форматирования приведено в документации clang-format [31].

Каждый ключ из описания стиля форматирования кода, относящийся к отступам, задает определенную ширину отступа для каждого случая (отступ содержания блока в теле условных операторов, оступ метки `case` относительно оператора `switch` и т. д.). Однако в данной работе необходимо не осуществить проверку исходного кода программы на соответствие определенному стилю, а проверить единообразие расстановки отступов для *любого* стиля (который может отличаться в различных файлах и заранее не известен).

2.2.2. Clang Static Analyzer

Clang Static Analyzer анализирует исходный код на этапе компиляции. Согласно информации на сайте проекта [25], статический анализатор должен расширять возможности компиляторов и «идти дальше», сообщая о потенциальных ошибках времени выполнения. Также сообщается, что из-за глубины анализа и сложности алгоритмов статический анализ может занимать намного больше времени, чем компиляция программы, поэтому для ускорения работы Clang Static Analyzer имеет ограничения на количество проверок и, в связи с этим, может выявлять не все ошибки. Также возможны случаи ложных срабатываний.

В отличие от clang-tidy, который является линтером и проверяет стиль оформления кода, Clang Static Analyzer не имеет соответствующих флагов в настройке. Этот инструмент больше подходит для анализа всего проекта, что становится возможным благодаря «встраиванию» анализатора в процесс сборки.

Попробуем запустить этот анализатор и посмотреть, сможет ли он найти по крайней мере ошибку в строке 10, которая может привести к неверным результатам выполнения программы.

Анализ запускается командой `scan-build`, поставленной перед командами для сборки и компиляции проекта. Результат запуска приведен в листинге 3. Ни одной ошибки найдено не было (`no bugs found`).

Листинг 3 — Clang Static Analyzer.

```
1 > scan-build gcc dummy.c
2 scan-build: Using '/usr/lib/llvm-6.0/bin/clang' for static analysis
3 scan-build: Removing directory '/tmp/scan-build
    -2019-06-09-203732-25240-1' because it contains no reports.
4 scan-build: No bugs found.
```

2.3 cpplint

Согласно информации, приведенной на странице проекта на сайте GitHub [26], инструмент осуществляет проверку, что файл исходного текста на C++ соответствует стандарту форматирования Google [3]. Относительно отступов в стандарте указывается только то, что размер отступа по умолчанию равняется двум пробелам.

В описании к программе также указано, что для проверки используются регулярные выражения[1] в связи с чем не все ошибки могут быть выявлены, а также случаются ложно-положительные срабатывания.

Анализатор cpplint принимает на вход только файлы исходного кода на языке C++, поэтому изменим расширение тестового файла, переименовав `dummy.c` в `dummy.cc` (возможность анализа файлов, содержащих тексты программ на языке C++ в данном случае возможно и даже избыточно, так как язык C++, с некоторыми исключениями, является надмножеством языка C [33]).

Программа распространяется в виде Python-скрипта. Запустим проверку. Согласно документации, создаем в том же каталоге, где находится файл `dummy.cc`, конфигурационный файл `CPPLINT.cfg` и добавим в вывод все фильтры, в названиях которых присутствует `readability` и `whitespace`. Содержание конфигурационного файла приведем в листинге 4.

[1] Регулярное выражение (англ. regular expression, regexp) — вид текстового шаблона (text pattern), который может использоваться во многих современных приложениях и языках программирования, например для проверки, что входной текст соответствует шаблону, для выполнения поиска частей текста, соответствующих шаблону, в большем блоке текста, для замены текста, соответствующего шаблону, на другой текст, разбивать текст на блоки в соответствии с шаблоном и т.д. [32]

Листинг 4 — Содержание файла `CPPLINT.cfg`.

```
filter=+whitespace/indent,+build/class,+build/c++11,+readability/
    ↪ alt_tokens,+readability/braces,+readability/casting,+readability/
    ↪ check,+readability/constructors,+readability/fn_size,+readability/
    ↪ inheritance,+readability/multiline_comment,+readability/
    ↪ multiline_string,+readability/namespace,+readability/nolint,+
    ↪ readability/nul,+readability/strings,+readability/todo,+
    ↪ readability/utf8,+whitespace/blank_line,+whitespace/braces,+
    ↪ whitespace/comma,+whitespace/comments,+whitespace/
    ↪ empty_conditional_body,+whitespace/empty_if_body,+whitespace/
    ↪ empty_loop_body,+whitespace/end_of_line,+whitespace/ending_newline
    ↪ ,+whitespace/forcolon,+whitespace/indent,+whitespace/line_length,+
    ↪ whitespace/newline,+whitespace/operators,+whitespace/parens,+
    ↪ whitespace/semicolon,+whitespace/tab,+whitespace/todo
```

Запустим анализ с максимальным уровнем вывода (`--verbose=0`). Результат запуска приведен в листинге 5.

Листинг 5 — Результат запуска программы cpplint.

```
 > python cpplint.py --verbose=0  dummy.cc
dummy.cc:0:  No copyright message found.  You should have a line:
    "Copyright [year] <Copyright Owner>"  [legal/copyright] [5]
dummy.cc:1:  Missing space before {  [whitespace/braces] [5]
dummy.cc:2:  Line ends in whitespace.  Consider deleting these extra
    spaces.  [whitespace/end_of_line] [4]
dummy.cc:2:  Redundant blank line at the start of a code block should be
     deleted.  [whitespace/blank_line] [2]
dummy.cc:5:  { should almost always be at the end of the previous line
    [whitespace/braces] [4]
dummy.cc:6:  Weird number of spaces at line-start.  Are you using a 2-
    space indent?  [whitespace/indent] [3]
dummy.cc:9:  Line ends in whitespace.  Consider deleting these extra
    spaces.  [whitespace/end_of_line] [4]
dummy.cc:9:  Else clause should be indented at the same level as if.
    Ambiguous nested if/else chains require braces.  [readability/
    braces] [4]
Done processing dummy.cc
Total errors found: 8
```

Приведем перевод сообщений об интересующих нас ошибках:

– «`dummy.cc:6: Странное количество пробелов в начале строки. Вы используете отступы величиной в два пробела?`»;

– «`dummy.cc:9: Ветвь else должна иметь отступ, равный отступу if. Двусмысленные вложенные последовательности if/else требуют обрамления скобками.`»;

Как мы видим, из трех ошибок в тестовом файле (см. стр. 11) были найдены две (в строках 6 и 9), причем из описания ошибки в строке 6 неясно, в чем она заключается.

Таким образом, данный инструмент не может использоваться для решения поставленной задачи, несмотря на то, что среди приложений, рассмотренных в данной работе, cpplint обнаружил наибольшее количество ошибок.

2.4 Cppcheck

Согласно информации на странице проекта [27], Cppcheck — статический анализатор кода на языке C/C++, предоставляющий возможность обнаружения ошибок и нацелен на выявление неопределенного поведения (undefined behaviour) и *опасные конструкции кода*. Цель проекта - обнаруживать только настоящие ошибки в коде, иными словами, уменьшить количество ложных срабатываний. [27]

Программа получает на вход файл исходного текста программы и возвращает сообщения об ошибках, записывая при этом результат анализа в виде XML-файла с расширением `.dump`. Эти данные содержат промежуточное представление программы, которое можно в дальнейшем анализировать дополнительными скриптами (addons). На сайте проекта не было найдено скрипта, который относился бы к решаемой задаче.

В справочной информации (на сайте проекта либо в командной строке в результате запуска команды `cppcheck --doc`) не было найдено флагов, запускающих проверки, относящихся к стилю оформления кода или расстановке отступов

Проверим, обнаружит ли анализатор хотя бы потенциальную ошибку в строке 10 тестового файла (см. листинг 1 на с. 11), которую обнаружили анализаторы Cpplint и clang-tidy.

Результат запуска приводится в листинге 6.

Листинг 6 — Результат запуска программы Cppcheck.

```
1 > cppcheck dummy.c
2  Checking dummy.c ...
```

В процессе анализа не было найдено ни одной ошибки. Таким образом, статический анализатор Cppcheck не подходит для решения поставленной задачи.

2.5 Splint

Splint (Secure Programming Lint) — инструмент для статического анализа программ, написанных на языке C, на предмет уязвимостей безопасности и ошибках в коде. При необходимой настройке может быть использован как лучшая версия упомянутой ранее утилиты lint. На сайте программы также утверждается, что если в программу добавлены аннотации[1], Splint может выполнять более глубокую проверку кода, нежели любой стандартный линтер [28] (при этом не указывается, какие линтеры считаются стандартными).

Проверим, какие ошибки в тестовом файле обнаружит данный анализатор. В документации к проекту перечислены флаги и режимы запуска. Среди флагов к отступам и форматированию относятся либо проверки форматных строк в функциях типа `printf`, либо к форматированию и расстановке отступов в выводе самой утилиты. Среди режимов запуска выберем режим `strict`, выполняющий максимальное количество проверок кода.

Результат запуска приложения приведен в листинге 7.

Листинг 7 — Результат запуска программы Splint.

```
splint +strict dummy.c
Splint 3.1.2 --- 20 Feb 2018

dummy.c:1:5: Function main declared without parameter list
  A function declaration does not have a parameter list. (Use -noparams
  to inhibit warning)
dummy.c: (in function main)
dummy.c:8:9: Test expression for if not boolean, type int: 1
  Test expression type is not boolean or int. (Use -predboolint to
  inhibit warning)
dummy.c:9:13: Test expression for if not boolean, type int: 2
dummy.c:9:23: Body of if clause of if statement is not a block: return 2
  If body is a single statement, not a compound block. (Use -ifblock to
  inhibit warning)
dummy.c:11:16: Body of else clause of if statement is not a block:
    return 3
dummy.c:11:16: Body of if statement is not a block:
                  if (2) return 2 else return 3

Finished checking --- 6 code warnings
```

Как видно из листинга, ни одна из требуемых ошибок (см. стр. 2.1) не была обнаружена. Тем не менее, для строки 9 было выведено предупреждение

[1] Аннотации в языке C — расширение языка, которое позволяет явно описывать поведение функции [34]. Аннотации не входят в стандарт языка C, являются платформо-зависимым решением и могут отличаться для разных компиляторов (ср. аннотации Microsoft VC++ [34] и Clang [35]).

об отсутствии фигурных скобок в теле оператора `if`: «`dummy.c:9:23: Тело ветви else конструкции if не является блоком: return 2. Тело if состоит из единственного утверждения, не составной блок`».

Исходя из полученных результатов, делаем вывод, что утилита Splint не пригодна для решения поставленной задачи поиска ошибок в расстановке отступов.

2.6 Выводы обзорной части

Для удобства анализа полученных данных сведем результаты в таблицу. Напомним, что каждому из рассмотренных статических синтаксических анализаторов на вход был подан файл `dummy.c` (см. листинг 1 на стр. 11). Файл содержит в себе три ошибки расстановки отступов, которые влияют на качество восприятия кода и, согласно заданию, должны быть выявлены, а пользователю должны быть показаны сообщения, описывающие суть ошибки (описание ошибок см. на стр. 11).

Сводка результатов приведена в таблице 1.

Таблица 1 — Сравнение рассмотренных статических синтаксических анализаторов на предмет выявления ошибок в тестовом файле `dummy.c`

Анализатор	Разные отступы в блоке	Несоответствие отступа уровню вложенности	Разный отступ ветвей оператора `if-else`
clang-tidy	—	—	Предупреждение об отсутствии скобок. Указание на двусмысленность выражения.
Clang Static Analyzer	—	—	—
cpplint	—	Предупреждение о «странном» отступе. Причина неясна.	Сообщение об ошибке отступа. Напоминание о скобках.
Cppcheck	—	—	—
Splint	—	—	Предупреждение об отсутствии скобок.

Проанализировав полученные результаты, можно сделать следующий вывод: несмотря на то, что статические анализаторы могут предоставить тре-

буемую функциональность (а именно проверять корректность расстановки отступов в зависимости от уровня вложенности, см. раздел «Постановка задачи») и некоторые действительно частично ее предоставляют (clang-tidy, cpplint), они не могут быть использованы для решения по следующим причинам:

1. Статические анализаторы разрабатываются для определения потенциальных ошибок времени выполнения программы, и задача проверки форматирования не является приоритетной, поэтому анализ отступов либо не удовлетворяет требованиям данного задания, либо отсутствуют вовсе.
2. Если проверка стиля оформления кода выполняется, она производится в соответствии с некоторым установленным стандартом. Такое решение удобно при встроенной функции автодополнения в среде разработки и целесообразно для больших проектов, требующих непрерывной поддержки. Однако в среде тестирования учебных программ, которой пользуются начинающие программисты, требование жестко соблюдать стиль форматирования исходного текста программы будет выставлять слишком высокий порог прохождения тестирования, так как будут отклоняться даже хорошо отформатированные программы, но написанные в другом стиле.

Таким образом возникает необходимость разработки нового инструмента для проверки корректности расстановки отступов с учетом того, что эта программа будет использоваться в обучающих целях.

2.7 Программная архитектура существующих статических анализаторов и утилит форматирования

Рассмотрим принципы построения статического анализатора cpplint, который показали наилучший результат среди рассмотренных.

Помимо этого интерес также представляет Clang, на котором основан статический анализатор clang-tidy. В семействе утилит, основанных на Clang, изучим устройство программы clang-format, которая, в отличие от clang-tidy, предназначенной для автоматического форматирования исходных текстов про-

грамм. Как было указано выше, сама эта программа не может быть применена для решения поставленной задачи, так как в результате работы выдает отформатированный по заданным правилам код, а не сообщения об ошибках. Однако изучение ее внутренней организации может быть полезно при проектировании приложения, проверяющего корректность форматирования кода.

Все из перечисленных утилит являются свободными программами[1], поэтому конфликта с условиями использования программ при анализе и модификации их исходного кода не возникает.

Cpplint — анализатор типа lint для проверки исходного кода на соответствие стилю Google Style Guides. Программа распространяется в виде скрипта на языке Python, на момент написания работы содержащий порядка 6 тыс. строк. Несмотря на то, что документации об архитектуре приложения найти не удалось, возможно получить представление о работе программы, читая ее исходный код, размещенный на платформе GitHub [26].

В функции `main()` происходит чтение аргументов командной строки, и для каждого файла вызывается функция `ProcessFile`, в качестве аргументов принимающая имя файла, флаг, показывающий, насколько подробным должен быть вывод и список дополнительных проверок. В этой функции программа разбивается на строки, они предобрабатываются (проверяются символы перевода строки и т.п.).

Вызывается функция `ProcessFileData`, где удаляются строки кода с метками «`// NOLINT`» и «`// NOLINTNEXTLINE`» (экранирование с целю избежать ложно-положительных срабатываний на некоторых участках кода), после чего к полученным данным последовательно проверяются фильтры по умолчанию (проверка на наличие сообщения об авторских правах, проверка формата написания заголовков), а затем вызывается построчная обработка, где будут также применены дополнительные фильтры, указанные при вызове программы.

В функцию обработки строк `ProcessLine`, кроме уже перечисленных аргументов, передается внутреннее состояние программы в виде стека, в котором хранятся объект класса `NestingState`, содержащий данные о фигурных скобках.

В программе различаются *три типа скобок*:

[1] Термин «свободная программа» означает, что у пользователей есть четыре существенных свободы: 1) выполнять программу, 2) изучать и править программу в виде исходного текста, 3) перераспространять точные копии и 4) распространять измененные версии .

1. Ограничивающие тело в объявлении структуры или класса.
2. Заключающие область видимости имен (namespace).
3. Все остальные.

Таким образом программа хранит информацию об уровне вложенности, на котором происходит анализ в данный момент.

Функции проверок основаны на регулярных выражениях. Программа ищет заранее заданные шаблоны и в случае их обнаружения выдает соответствующее сообщение. Пример такой функции, выявляющей некорректное увеличение указателя, приведен в листинге 8.

Листинг 8 — Функция обнаружения некорректного увеличения указателя в исходном коде анализатора cpplint.

```
# Matches invalid increment: *count++, which moves pointer instead of
# incrementing a value.
_RE_PATTERN_INVALID_INCREMENT = re.compile(
    r'^\s*\*\w+(\+\+|--);')


def CheckInvalidIncrement(filename, clean_lines, linenum, error):
  """Checks for invalid increment *count++.

  For example following function:
  void increment_counter(int* count) {
    *count++;
  }
  is invalid, because it effectively does count++, moving pointer, and
  should be replaced with ++*count, (*count)++ or *count += 1.

  Args:
    filename: The name of the current file.
    clean_lines: A CleansedLines instance containing the file.
    linenum: The number of the line to check.
    error: The function to call with any errors found.
  """
  line = clean_lines.elided[linenum]
  if _RE_PATTERN_INVALID_INCREMENT.match(line):
    error(filename, linenum, 'runtime/invalid_increment', 5,
          'Changing pointer instead of value (or unused value of
   operator*).')
```

На странице проекта отмечается, что из-за использования регулярных выражений не все отклонения могут быть выявлены, а также возможны ложные срабатывания. В начале файла с исходным кодом программы есть сопроводительный комментарий с описанием назначения инструмента и известных проблем. В частности там указано, что программа не всегда корректно обрабатывает комбинации «//» и «/*» внутри строковых констант. Там же подчеркивается,

что инструмент разрабатывался в образовательных целях и задача обнаружения всех возможных ошибок не ставилась. В данной работе также ставится задача разработать обучающий инструмент, но так как от результатов его работы зависит успешность прохождения студентом тестирования, такие ошибки, как некорректный разбор строковых констант, недопустимы.

Рассмотрим теперь утилиту clang-format. Схему работы можно найти в презентации Дэниэла Джаспера (Daniel Jasper) [36] на странице проекта LLVM, частью которого является Clang.

Вначале воспроизводятся первые стадии компиляции: лексический и синтаксический анализ (см. определения на стр. 8). Эти стадии выполняются с помощью инструментов, предоставляемых Clang.

Также программа разбивается на цельные строки (не содержащие переносов выражений, unwrapped lines), каждому переносу строки присваивается штраф (penalty), программа перестраивается так, чтобы штрафы были минимальными. Процесс обработки приведен на рисунке .

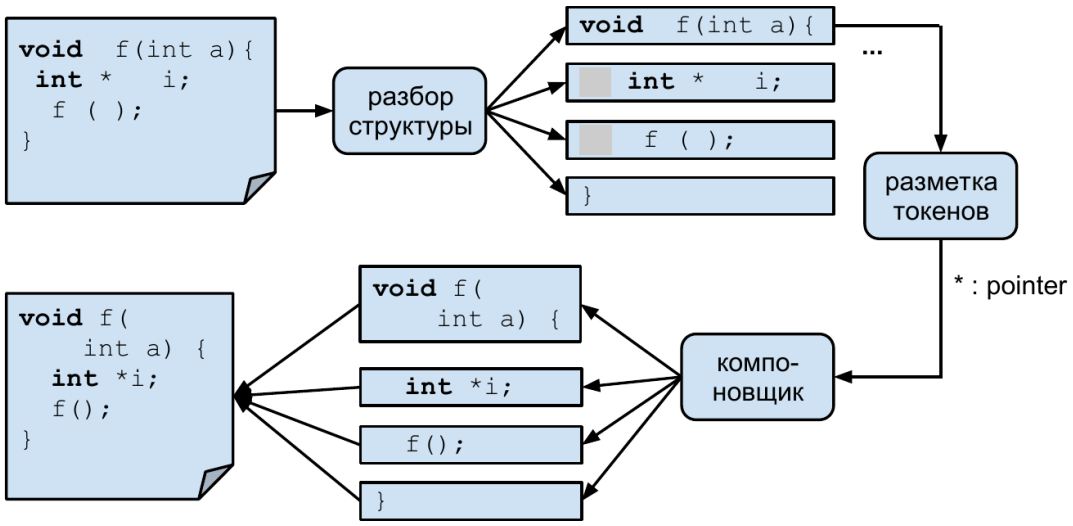

Рисунок 4 — Процесс обработки файла программой clang-format [36].

Из разобранных примеров видно, что задачу форматирования текста можно решать различными способами. При этом используется информация о том, на каком уровне вложенности находится то или иное выражение. Для этих целей cpplint использует стек, где хранится информация о типе блока, а в clang-format — синтаксическое дерево, которое полностью отражает синтаксическую структуру программы.

3 Разработка приложения

Исходя из анализа существующих программ и условий поставленной задачи (в частности, необходимость проверки соответствия отступа уровню вложенности), можно сделать вывод, что программа должна хранить: а) разбиение программы на строки с сохранением информации об отступах и б) информацию о вложенности выражений.

Как было показано при рассмотрении архитектуры анализатора cpplint, способ построения структуры языка, основанный на поиске шаблонов с помощью регулярных выражений, недостаточно надежен. Поэтому решено было воспроизвести первые стадии компиляции, как это было сделано в приложении clang-format, но изменить структуру для хранения внутреннего представления с синтаксического дерева (структуры, удобной для вычисления выражений), на более подходящую.

Разработанное приложение обрабатывает файл с исходным кодом по следующему алгоритму:

1. Лексический анализ.
2. Синтаксический анализ.
3. Построение внутреннего представления входных данных.
4. Анализ полученного внутреннего представления на предмет ошибок (см. раздел «Постановка задачи»).
5. Формирование сообщений об обнаруженных ошибках в зависимости от режима вывода.

3.1 Выбор структуры данных для внутреннего представления и разработка алгоритма

Пусть в результате синтаксического анализа построено синтаксическое дерево (назовем его `ST`). Для проверки равенства отступов во всех строках на одном уровне вложенности необходимо получить все элементы с одинаковой длиной пути от корневой вершины. Для этого придется производить обход дерева в ширину и накапливать информацию о токенах, расположенных на одном уровне вложенности. Однако никакой информации об их взаимном расположении в тексте дерево не дает, так как, например, для операций присваивания,

символ операции в синтаксическом дереве будет располагаться выше, чем операнды, в то время как в реальном тексте он будет находится между ними и при переносе на другую строку должен будет располагаться не левее, чем начало выражения (см. рис. 5).

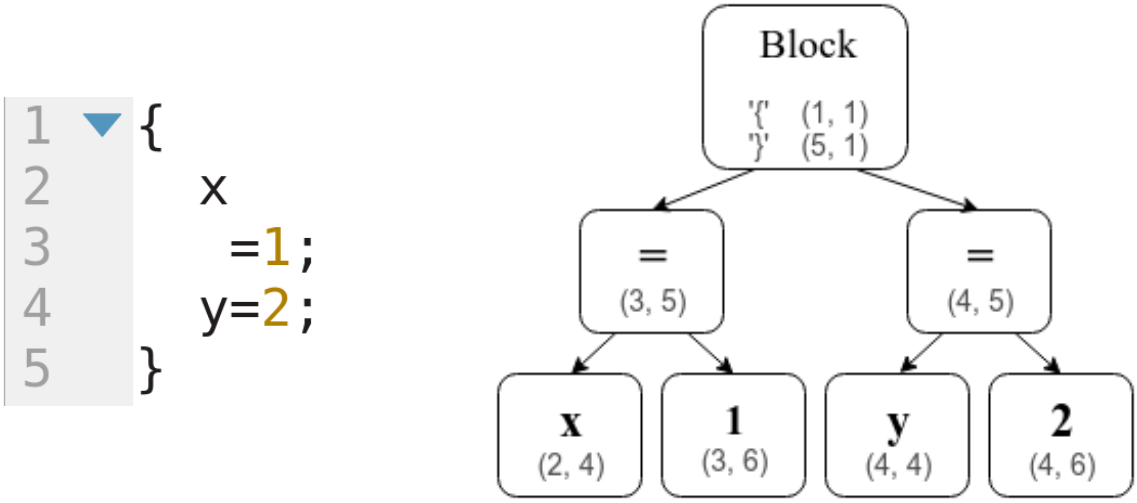

Рисунок 5 — Пример участка кода и его синтаксического дерева (в скобках указаны номер строки и столбца).

Рассмотрим, как можно преодолеть данные ограничения. Для хранения информации об отступах на каждом уровне вложенности необходимо выделить дополнительный массив (назовем его `indents_arr`). Опишем возможный алгоритм.

Алгоритм 1.

Шаг 1. При обходе синтаксического дерева `ST` (не важно, в ширину или в глубину) извлечь следующий элемент.

Шаг 2. Проверить, является ли данный токен первым элементов в строке (для этого необходимо дополнить лексический анализ так, чтобы для каждого токена хранить эту информацию).

- Если является — перейти на следующий шаг.
- Если не является — вернуться к шагу 1.

Шаг 3. Извлечь из массива `indents_arr` информацию об отступе на текущем уровне вложенности.

- Если данный элемент массива пуст — записать информацию об отступе для текущего токена и перейти на шаг 1.
- Если не пуст и значение в массиве совпадает с величиной отступа для данного токена — вернуться на шаг 1.

– Иначе (если значения отличаются) сформировать сообщение об ошибке и вернуться на шаг 1.

Структуры данных, использованные в данном алгоритме, показаны на рисунке 6.

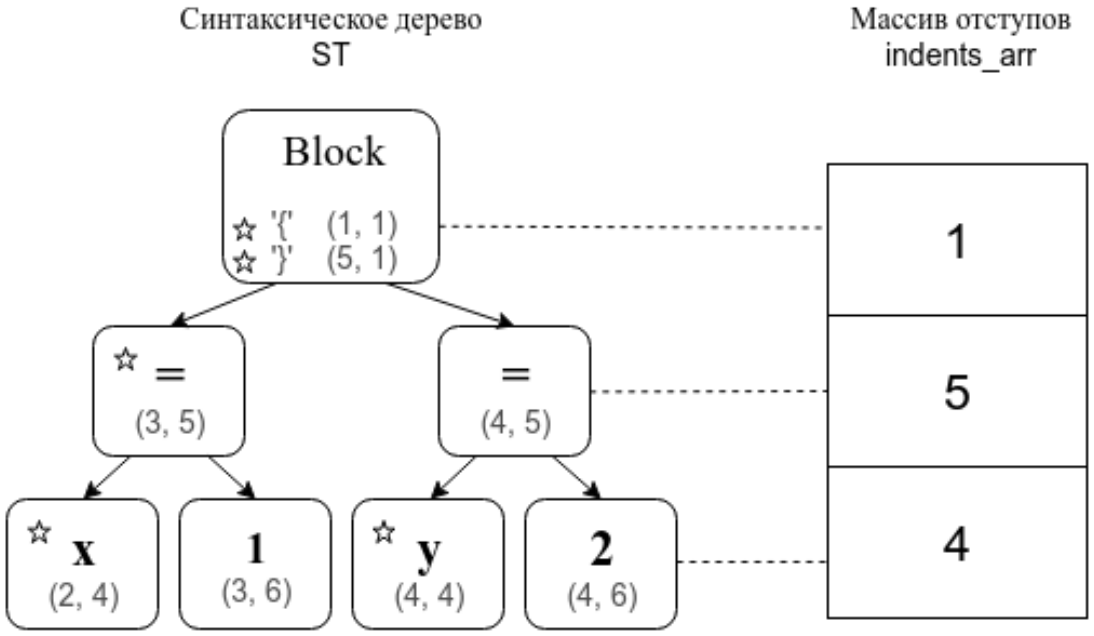

Рисунок 6 — Структуры данных для первой версии алгоритма (звездочкой отмечены элементы, являющиеся первыми в строке, координаты элемента указаны в скобках в формате «строка, столбец»).

Этот алгоритм выполняет проверку на то, что на одном уровне вложенности все строки имеют одинаковый отступ. Однако при такой архитектуре неясно, как выполнять проверку корректности постановки отступа при переносе выражения (новая строка должна начинаться не левее предыдущей). В таком случае для данного токена необходимо будет подниматься вверх по синтаксическому дереву до тех пор, пока не будет найден корень выражения, к которому принадлежит, а затем выполнять обход от найденной вершины вниз, чтобы найти начало выражения.

Значит, в программе необходимо хранить информацию о разбиении на строки (и в clang-format, и в cpplint также есть этап построчного анализа). Возможны различные варианты решения данной задачи, например хранить массив строк (назовем его `lines`), где каждая строка представлена последовательностью элементов, содержащих токены и указатели на расположение этих токенов в синтаксическом дереве. Для хранения информации об отсту-

пах на каждом уровне вложенности как и прежде будет использоваться массив `indents_arr`. Визуализация структур данных, использованных в алгоритме, приведена на рисунке 7.

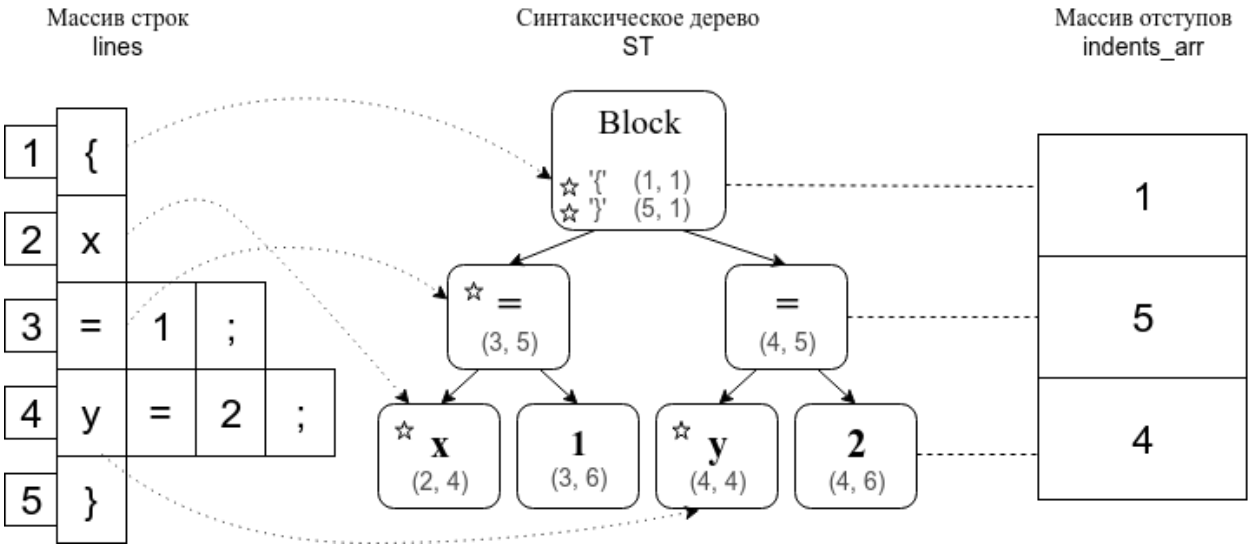

Рисунок 7 — Структуры данных для второй версии алгоритма (звездочкой отмечены элементы, являющиеся первыми в строке, координаты элемента указаны в скобках в формате «строка, столбец»).

Попытаемся дополнить алгоритм, используя введенную структуры данных.

Алгоритм 2.

Шаг 1. Выполнить алгоритм 1 для проверки того, что на одном уровне вложенности строки имеют одинаковый отступ.

Шаг 2. Начать проверку того, что при переносе выражения продолжение строки находится не левее, чем ее начало: перейти на новую строку и извлечь ее первый элемент.

Шаг 3. Проверить предыдущую строку на предмет переноса выражения: выбрать из нее последний элемент.

Если элемент предыдущей строки относится к тому же утверждению[1], что и данный токен, сравнивнить отступы в этих строках.

– Если отступ в текущей строке меньше, чем в предыдущей строке, сгенерировать сообщение об ошибке и перейти на шаг 2.

[1]Напомним, что в постановке задачи «утверждением» считается одна из следующих конструкций: выражения, отделенные точкой с запятой (операции присваивания, вызов функции, объявления переменных и полей структур struct и объединений union); тело объявления перечисления enum; имя метки с двоеточием; имя оператора if, while, for, switch вместе со следующим за ним выражением в скобках.

– Иначе — перейти на шаг 2.

Если последний элемент предыдущей строки является корректным завершением утверждения либо находится на предыдущем уровне вложенности, считать, что переноса не было и перейти на шаг 2.

Очевидно, что алгоритм 2 обладает существенным недостатком: неясно, как определять границы утверждений на шаге 3.

Кроме того, как видно на рисунках 7 и 6, при использовании синтаксического дерева непросто проанализировать соотношение величины отступа на данном уровне вложенности к отступам на бóльших и меньших уровнях вложенности (как было отмеечно ранее, в синтаксическом дереве оператор равенства имеет уровень вложенности меньше, чем его операнды x и y, однако при переносе должен располагаться правее первого операнда).

Эти трудности возникают из-за того, что грамматическая структура языка, которую отражает синтаксическое дерево, основана на рекурсивных правилах и вложенных конструкциях, в то время как для решения задачи требуется анализировать последовательность строк, то есть линейную, а не рекурсивную структуру. Поэтому для решения задачи с использованием разработанной архитектуры нужно будет добавлять все больше и больше массивов (например для хранения границ утверждений, которые могут быть перенесены и т.п.).

Однако окончательно отказаться от использования синтаксического дерева нельзя, так как помимо линейного списка строк необходимо хранить информацию об уровнях вложенности.

Для решения данной задачи решено было скомбинировать линейную структуру массива строк и информацию об уровне вложенности в следующем виде:

1. Отказаться от явного хранения синтаксического дерева. Реализовать парсер, основанный на методе рекурсивного спуска, в виде несколько модифицированного конечного автомата. Синтаксический анализатор будет изменять свое состояние, начиная разбор нового правила. Новое состояние будет храниться в стеке состояний (назовем его *метасостоянием*) до тех пор, пока разбор правила не будет окончен (этим реализуемый метод похож на алгоритм «перенос-свертка»).

   Таким образом можно легко выделить границы утверждений и текущий уровень вложенности, на котором находится каждый токен.

2. Не выполнять разбор правил глубже уровня утверждений. Считать утверждение последовательностью символов, имеющих один уровень вложенности.
3. Выделить отдельное состояние `STATEMENT` для переноса утверждения.
4. В результате синтаксического анализа генерировать массив строк, где для каждой строки будет сохранено метасостояние, в котором находился парсер в момент начала разбора данной строки.

Визуализация полученной структуры — массива строк с сохраненным метасостоянием, представлена на рисунке 8.

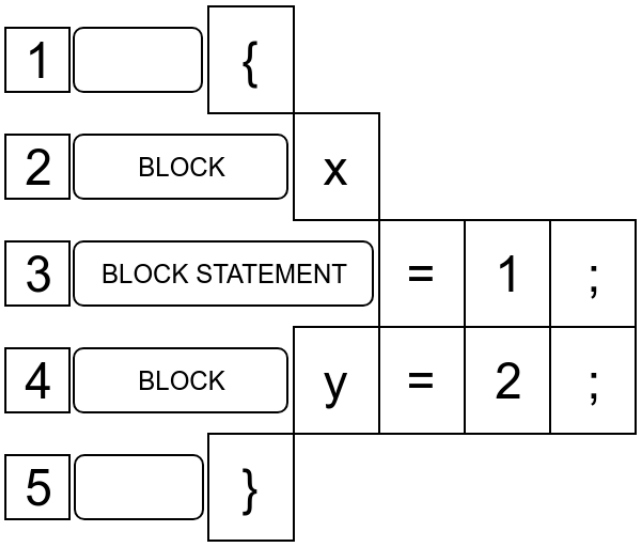

Рисунок 8 — Массив строк с сохраненным метасостоянием парсера.

Как видно на рисунке 8, полученная структура естественным образом представляет задачу в виде, близком к ее представлению человеком: чем больше уровень вложенности (чем больше элементов в метасостоянии строки), тем больше должен быть отступ. Также эта структура удобна для дальнейшего анализа на ЭВМ, так как в ней уже собрана вся необходимая информация.

Теперь анализ на равенство отступов на одном уровне вложенности можно проводить без использования дополнительного массива, отсортировав строки по метасостояниям и собрав всю необходимую информацию за один проход.

Проверка на корректность расстановки отступов для переносов также тривиально: для каждой строки, верхнее состояние которой равно `STATEMENT` достаточно сравнить ее с предыдущей строкой и генерировать ошибку, если отступ первой оказался меньше.

Для проверки корректности величины отступов также выполняется построчное сканирование, и если длина метасостояния одной строки больше ме-

тасостояния другой строки, первая считается вложенной и отступы этих строк сравниваются. Если отступ в первой строке (вложенной) меньше, чем во второй, необходимо генерировать ошибку, иначе переходить к следующей строке.

Состояния могут быть следующими:

BLOCK — обозначает, что строка находится в блоке: в теле функции, в теле операторов `if`,`else`, `do`, `while`, `for`, простом блоке.

STATEMENT — используется внутри утверждений (заголовках функций, объявлений переменных, операторов присваивания).

SWITCH — используется внутри тела оператора `switch`. Это состояние было выделено отдельно от состояния `BLOCK`, так как внутри блоков `switch` форматирование может отличаться от прочих операторов (см. определение 1 в разделе «Постановка задачи» и рисунок 9 на с. 49 в разделе «Руководство пользователя»).

CASE — это состояние ставится перед метками `case` и `case`, чтобы обозначить, что они имеют собственный уровень вложенности внутри блока `switch`.

CASE_STATEMENT — состояние, обозначающее разбор блока кода, следующего за меткой `case` или `default`.

LABEL — это состояние, как и `CASE`, добавляется перед пользовательской меткой, чтобы обозначить для нее собственный уровень вложенности.

3.2 Лексический анализ

Лексический анализ текста проводился с помощью реализованного вручную объектно-ориентированного сканера.

В результате работы лексического анализатора программа разбивается на токены — объекты класса `Token`, хранящие следующие данные:

- Тип токена.
- Образ лексемы — вид лексемы в тексте. В реализованном приложении для хранения этого поля использован тип данных `std::string_view`, позволяющий хранить только указатель на подстроку в строке с полным текстом программы, что позволило избежать дополнительных накладных расходов на хранение образов лексем.

– Поле `start` — координата первого символа токена.

– Поле `follow` — координата первого символа, следующего за токеном.

Координта представляет собой кортеж из трех чисел — индекса символа в тексте анализируемой программы, номера строки и номера столбца.

На этапе лексического анализа комментарии удаляются. Также удаляются пробелы, кроме тех, что которые находятся в начале следующей строки.

Символ перевода строки и следующие за ним пробельные символы объединяются в токен типа `NEWLINE`, который хранит информацию об отступе в строке.

Лексический анализатор реализован в виде класса `Scanner`, который хранит в себе следующие поля:

– Строка с текстом анализируемой программы `prog`.

– Отображение слов в ключевые слова.

– Указатель на начало считываемой лексемы в программе `start_ptr`.

– Указатель на текущий символ в в программе `cur_ptr`.

– Координата `start` лексемы `start_pos`.

– Координата текущего символа `cur_pos`.

– Текущий символ `cur_char`.

– Длина считанного участка следующей лексемы `token_len`.

– Список сообщений об ошибках `errors_list`.

– Сохраненное внутреннее состояние `state`, содержащее копии полей `start_ptr`, progcur_ptr, `start_pos`, `cur_pos`, `token_len` и `errors_list`.

Класс `Scanner` предоставляет следующие возможности:

– Извлечение следующего токена из потока ввода (текста программы) в функции `nextToken`.

– Чтение следующего токена из потока ввода без его извлечения в функции `peekToken` (внутреннее состояние сканера не изменяется, т.е. повторный вызов функции `peekToken` или `nextToken` вернет тот же самый результат).

– Сохранения текущего состояния в переменную или во внутреннее поле `state` и восстановление состояния из сохраненной копии.

Реализованный сканер в целом является классическим лексическим анализатором (прицины построения см. напрмер, в [5, сс. 111–152]), за исключением процесса обработки символов переносов строк и следующих за ними пробе-

лов (как правило, для компиляции программы в машинный код все пробельные символы, включая символы перевода строк, исключаются на этапе лексического анализа).

3.3 Разработка грамматики

Одной из самых сложных задач в данной работе была разработка грамматики языка C, которая не разбирает выражения внутри утверждений (что считается утверждением см. в разделе «Постановка задачи»), но при этом удовлетворят всем корректным программам на языке C и поддерживает нестандартные расширения компилятора GCC (что требуется по условию задачи).

За основу была взята грамматика из стандарта языка C от 2017 года ISO/IEC9899:2017 [37, с. 338].

Начиная с правила верхнего уровня (с которого стартует алгоритм синтаксического разбора), будем рекурсивно спускаться вниз, выбирая правила, которые будут включены в разработанную грамматику.

Далее оригинальные грамматические правила из спецификации будут записываться в том виде, в котором они приведены в документе ISO/IEC9899:2017. Правила построенной грамматики будут записываться в расширенной форме Бэкуса-Наура (Extended Backus-Naur Form), или РБНФ. Правила обозначаются *нетерминальным символом*, или *нетерминалом*. Символы, которые не могут быть раскрыты далее (то есть, токены) именуются *терминальными символами* или «терминалами». Существует множество различных вариаций РБНФ [38], в данной работе будут использоваться следующие обозначения:

- терминальные символы (токены) заключены в одиночные кавычки «'»;
- символы (терминальные и нетерминальные) в правилах группируются заключением в круглые скобки «(» и «)» для определения приоритета операций (будем считать, что один элемент, не заключенный в скобки, также образует группу);
- символ «*», расположенный после группы, обозначает ее последовательное повторение 0 или более раз;
- символ «+» после группы обозначает ее повторение 1 или более раз;
- символ «?» после группы обозначает ее повторение 0 или 1;

– символ «|» между двумя группу обозначает альтернативу (при применении правила может быть использована либо левая, либо правая группа);
– комментарии заключаются между последовательностями «.*» и «*/».

Чтобы отличать новые правила от оригинальных, будем записывать первые прописными буквами.

Начнем составлять грамматику. Все, что находится внутри утверждения, будем считать простой последовательностью слов. Зафиксируем правило 1:

```
WORD-SEQUENCE = WORD *.
```
(1)

Словом считается токен любого из перечисленных типов: квадратные скобки, знаки унарных и бинарных операций и пунктуации, идентификаторы, числовые и строковые константы, буквенные литералы, звездочка (независимо от контекста), символ «?» и эллипс «...».

Покажем на примере, как производился выбор. Начнем с правила верхнего уровня `translation-unit`, которое имеет вид: «`translation-unit: external-declaration | translation-unit external-declaration`».

Следующее правило «`external-declaration: function-definition | declaration`» описывает объявление функции `function-definition` или объявление переменных и типов данных `declaration`. Нетерминальный символ `function-definition` раскрывается в большое количество правил, описывающих типы данных в возвращаемом значении и аргументах. Для проверки корректности переносов строк достаточно знать, что заголовок функции вплоть до открывающей фигурной скобки является утверждением, поэтому спецификаторы и объявления типа возвращаемого значения, а также объявления параметров можно абстрагировать нетерминалом `WORD-SEQUENCE`. Зафиксируем правило 2:

```
FUNCTION-DEFINITION = WORD-SEQUENCE IDENT
                      '(' WORD-SEQUENCE ')' BLOCK.
```
(2)

Где правило `BLOCK` состоит из последовательности сложных утверждений `STATEMENT`:

```
BLOCK = '{' STATEMENT * '}' .
```
(3)

Главная задача — редуцировать разбор выражений, которые могут быть перенесены на другую строку. Поэтому для объявлений переменных declaration, арифметических выражений внутри блоков и пр. создадим правило SIMPLE-STATEMENT, которое описывает все утверждения, оканчивающиеся фигурными скобками, а также объявление функции:

```
SIMPLE-STATEMENT = WORD-SEQUENCE ';'
    | WORD-SEQUENCE /* если при сканировании WORD-SEQUENCE
        обнаружилось, что оно удовлетворяет правилу
                                    FUNCTION-DEFINITION */.
```
(4)

Правило FUNCTION-DEFINITION было перенесено с верхнего уровня translation-unit на уровень разбора утверждения, так как компилятор GCC поддерживает возможность объявления функций в теле других функций и объявление функции можно считать таким же утверждением, как объявление переменной или цикла.

Рекурсивно спускаемся по правилам грамматики до уровня statement («утверждение»). Оно имеет следующий вид:

```
statement:
        labeled-statement
        compound-statement
        expression-statement
        selection-statement
        iteration-statement
        jump-statement.
```
(5)

Практически каждое из правил нетерминала 5 представляет собой утверждение, которое является последним уровнем разбора в нашей системе. Заменяем compound-statement на уже упомянутое правило 3 BLOCK, а expression-statement на упрощенный SIMPLE-STATEMENT (правило 4). Удаляем правило jump-statement, которое теперь так же попадает в правило SIMPLE-STATEMENT. Остальные подправила, к которым относятся объявления

операторов `if-else`, `while`, `do`, `for`, `switch`, оставим почти без изменений, заменив только в их объявлениях `expression` на `WORD-SEQUENCE` (правило 1).

В таблице 2 приведен пример преобразования для правила `selection-statement`: нетерминал `expression`, который при разборе выражения может быть рекурсивно раскрыт в сколь угодно большое количество правил, заменяется правилом 1 `WORD-SEQUENCE`, которое представляет любое выражение как последовательность слов. В листинге 9 приведен результат всех преобразований.

Таблица 2 — Преобразование правил грамматики языка C (жирным шрифтом выделен замененный нетерминал).

Исходная грамматика	Преобразованная грамматика

`selection-statement:`
`  if ( expression ) statement`
`  if ( expression ) statement`
`  else statement`
`  switch ( expression )`
`  statement`
	`SELECTION-STATEMENT =`
	`'if' '(' WORD-SEQUENCE ')'`
	`  STATEMENT ('else' STATEMENT)?`
	`|`
	`'switch' '(' WORD-SEQUENCE ')'`
	`  STATEMENT`

Для правильной интерпретации фигурных скобок «{» и «}», необходимо выяснить, в каких случаях они могут использоваться. Анализируя правила грамматики из спецификации, находим, что фигурные скобки встречаются в следующих правилах и только в них:

1. Блок (правило `compound-statement` как одна из альтернатив в правиле `statement`).
2. Тело функции (правило `compound-statement` внутри `function-definition`).
3. Список инициализации (фигурными скобками обрамлен нетерминал `initializer-list` в одной из альтернатив правила `postfix-expression` и в правиле `initializer`).
4. Объявление структуры или объединения (фигурные скобки обрамляют `struct-declaration-list` в правиле `struct-or-union-specifier`).
5. Объявление перечисления (аналогично предыдущим случаям, в фигурные скобки заключен нетерминал `enumerator-list`).

Заметим, что списки инициализации и тело перечисления в установленных условиях представляют собой последовательность слов `WORD-SEQUENCE` (см. правило 1). Выделим правило `INITIALIZER-LIST` как

послледовательность слов, заключенную в скобки и добавим этот нетерминал как альтернативу слову `WORD` в правило `WORD-SEQUENCE`. Теперь список инициализации является такой же частью утверждений, как, например, идентификатор или константа. Также добавим в правило 1 еще одну альтернативу — будем также считать словом последовательность слов в скобках (выделения выражений в скобках в рекурсивное правило необходимо для проверки правильности расстановки скобок в программе). К словам отнесем и объявления структур, объединений и перечислений, так как они используются в выражениях так же, как типы данных (в операторе приведения типа, объявлении переменной и пр.), которые считаются словом. Новые правила приведены листинге 10.

Листинг 9 — Сокращенная грамматика языка C. Часть 1.

```
1  TRANSLATION_UNIT =  STATEMENT *
2
3  STATEMENT = BLOCK
4            | SIMPLE_STATEMENT
5            | ITERATION_STATEMENT
6            | SELECTION_STATEMENT
7            | LABELED_STATEMENT
8
9  BLOCK = '{' STATEMENT* '}'
10
11 SIMPLE_STATEMENT = WORD_SEQUENCE ';'
12                  | WORD_SEQUENCE /* если при сканировании WORD_SEQUENCE
13                                     обнаружилось, что сканировалось
14                                     правило FUNCTION_DEFINITION */.
15
16 ITERATION_STATEMENT = 'for' '(' WORD_SEQUENCE ';' WORD_SEQUENCE ';'
17                                  WORD_SEQUENCE ')' STATEMENT
18                    | 'while' '(' WORD_SEQUENCE ')' STATEMENT
19                    | 'do' STATEMENT 'while' '(' WORD_SEQUENCE ')' ';'
20
21 SELECTION_STATEMENT = 'if' '(' WORD_SEQUENCE ')' STATEMENT
22                       ('else' STATEMENT) ?
23                    | 'switch' '(' WORD_SEQUENCE ')' STATEMENT
24
25 LABELED_STATEMENT = IDENT ':' STATEMENT
26                  | 'case' WORD_SEQUENCE ':' STATEMENT
27                  | 'default' ':' STATEMENT
```

В результате вышеописанных преобразований исходная грамматика языка C из стандарта [37, с. 338], включающая порядка 70 правил, была преобразована в грамматику из 12 правил (см. листинги 9 и 10). Полученная грамматика имеет высокий уровень абстракции, благодаря чему устойчива к использованию нестандартных расширений и не генерирует информацию, излишнюю для решения данной задачи.

Листинг 10 — Сокращенная грамматика языка C. Часть 2.

```
WORD = LBRACKET     /* [ */
     | RBRACKET     /* ] */
     | LANGLE       /* < */
     | RANGLE       /* > */
     | COLON        /* : */
     | COMMA        /* , */
     | DOLLAR       /* $ */
     | DOT          /* . */
     | ELLIPSIS     /* ... */
     | IDENT        /* идентификатор [A-ZA-Z_][A-ZA-Z_0-9]* */
     | STRING       /* строковая константа ".*" */
     | CHAR         /* буквенный литерал  */
     | NUM          /* числовая константа [0-9A-FA-FUULLXX]+(\. */
     | INCDEC       /* ++, -- */
     | OPERATOR     /* +, -, /, %, ~, |, ^, <<, >>, !, &&, || */
     | ARROW        /* -> */
     | STAR         /* * */
     | ASSIGNOP     /* =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>= */
     | COMPAREOP    /* >, <, !=, <=, =>, == */
     | AMPERSAND    /* & */
     | QUESTIONMARK /* ? */
     | KEYWORD      /* ключевые слова */

WORD-SEQUENCE = (WORD
                  | '(' WORD-SEQUENCE ')'
                  | INITIALIZER-LIST
                  | UNION-STRUCT-ENUM-DEFINITION)*
              | FUNCTION-DEFINITION

FUNCTION-DEFINITION = WORD-SEQUENCE IDENT '(' WORD-SEQUENCE ')' BLOCK

INITIALIZER-LIST = '{' WORD-SEQUENCE '}'

UNION-STRUCT-ENUM-DEFINITION = ('struct' | 'union') IDENT? BLOCK
                             | ('struct' | 'union') IDENT
                             | 'enum' IDENT? INITIALIZER-LIST
                             | 'enum' IDENT
```

Недостатком такого подхода является то, что данной грамматике будет удовлетворять множество текстов, не являющихся корректными программами на языке C, так как данная грамматика, по сути, описывает надмножество языка C. Однако это свойство не имеет значения при решении данной задачи, так как, по условию, на вход анализатора должны подаваться программы, прошедшие стадии компиляции и тестирования.

3.4 Синтаксический анализ

Синтаксический анализ выполняется парсером, который работает по схеме, описанной в разделе «Выбор структуры данных для внутреннего представления и разработка алгоритма».

Синтаксический анализатор выполнен в виде класса `Parser`, который хранит следующие данные:

– указатель на объект класса `Scanner`;
– текущий токен из потока ввода;
– стек состояний (текущее метасостояние);
– таблицу строк с сохраненными метасостояниями;
– список обнаруженных синтаксических ошибок.

Метасостояние представлено вектором (массивом с динамически изменяемым размером) констант `VectorState`, который на каждом шаге может быть изменен добавлением элемента в конец или извлечением последнего элемента.

Синтаксический анализ осуществляется методом рекурсивного спуска [5], то есть для каждого правила нетерминального символа грамматики определена функция, осуществляющая его разбор. Каждая функция производит считывание следующего элемента из потока ввода (лексический анализ производится параллельно, следующий токен извлекается из потока ввода по требованию парсера) и производит следующие действия:

– изменяет метасостояния парсера;
– добавляет новый элемент в таблицу строк;
– проверяет, что текущий токен соответствует ожидаемому терминалу в правиле;
– для каждого нетерминала правила вызывается функция его разбора.

Пример реализации функции для нетерминала `BLOCK` представлена в листинге 11.

Реализованный парсер осуществляет самовосстановление при ошибках. Если при проверке оказалось, что тип токена не соответствует ожидаемому, производится сканирование потока до тех пор, пока требуемый символ не будет найден, в список ошибок записывается соответствующее сообщение, и разбор продолжается (простейшая схема восстановления при ошибках).

Листинг 11 — Пример реализации функции для разбора нетерминала BLOCK.

```
// BLOCK = '{' STATEMENT* '}'
void Parser::parse_block(int level)
{
    // LOG – вывод отладосной информации
    LOG(level, std::string(" ") + __func__ + std::string(", first = ") +
     std::string(token));
    Coords fragment_start = token.start();

        // CHECK_TOKEN – проверерить, что текущий символ равен '{',
        // если нет, сканировать поток, пока не достигнем символа '{'
    CHECK_TOKEN({lexem::LBRACE}, {lexem::LBRACE});
    pushCase(Rules::Cases::BLOCK);  // Добавить статус в метасостояние
    nextToken();  // Считать следующий токен
    while (token.notEOF() && token != lexem::RBRACE) {
        parse_statement(level + 1);  // Считать последовательность
                                     // правил STATEMENT
    }
    popCase();  // Удалить статус из метасостояния
    // CHECK_TOKEN – проверерить, что текущий символ равен '}'
    CHECK_TOKEN({lexem::RBRACE}, {lexem::RBRACE});
    nextToken();

    // Вывод отладочной информации
    Coords fragment_end = token.start();
    LOG(0, GREEN_TEXT(get_image(fragment_start, fragment_end)));
    LOG(level, std::string(" ") + __func__ + std::string(", next = ") +
   std::string(token) << "\n\n");
}
```

Таблица строк является массивом структур Line, в которых хранится следующая информация:

– номер строки;

– величина отступа;

– массив токенов;

– метасостояние.

Добавление элементов в таблицу происходит на этапе запроса парсером следующего символа из потока ввода: если при сканер вернул символ перевода строки, к таблице добавляется новая пустая строка, в нее копируется текущее метасостояние парсера и величина отступа. Если считанный токен не является символом перевода строки, он добавляется в конец последней строки таблицы.

Следует отметить, что разработанная грамматика не позволяет в некоторых случаях делать выбор правила при сканировании только одного следующего символа. Например, в функции STATEMENT правило LABELED-STATEMENT начинается с идентификатора, и первым символом правила SIMPLE-STATEMENT также может быть идентификатор. В самом правиле SIMPLE-STATEMENT необхо-

димо различать, когда считывается обыкновенное утверждение, а когда объявление функции. Для этого в функции считывания этих правил были добавлены некоторые контекстные ограничения:

1. В функции, выполняющей разбор правила `STATEMENT`, если текущим символом является идентификатор, из потока ввода считывается (без извлечения) следующий токен. Если он является двоеточием, выполняем разбор правила `LABELED-STATEMENT`, иначе `SIMPLE-STATEMENT`.
2. В функции разбора нетерминала `SIMPLE-STATEMENT` храним информацию о считанной части правила, и если она удовлетворяет шаблону `WORD-SEQUENCE IDENT '(' WORD-SEQUENCE ')' BLOCK`, далее считываем правило `FUNCTION-DEFINITION`, иначе, продолжаем считывать простое выражение (в частности, интерпретируем фигурные скобки как ограничители списка инициализации, а не блока).

3.5 Проверка корректности расстановки отступов

После того, как в результате синтаксического анализа была построена таблица строк с сохраненными метасостояниями, если не было обнаружено синтаксических ошибок, построенная таблица передается в объект класса `Analyzer`, который выполняет проверку расстановки отступов.

Проверка происходит в несколько этапов:

1. Предобработка: удаление пустых строк.
2. Сбор статистики: строки группируются по метасостояниям.
3. Проверка на то, что строки с меньшим уровнем вложенности имеют меньший отступ.
4. Проверка равенства расстановки отступов для всех строк с одинаковым метасостоянием.

Анализатор действует по алгоритму, описанному в разделе «Выбор структуры данных для внутреннего представления и разработка алгоритма».

Все найденные анализатором ошибки отступов заносятся в список сообщения об ошибках и после окончания обработки печатаются в стандартный поток вывода, имя файла исходного текста с найденными ошибками печатается в файл со статистикой обработки, программа переходит к обработке следующего файла.

Если во время анализа был обнаружен символ табуляции, процесс останавливается, в список ошибок заносится соответствующее сообщение, все сообщения выводятся в стандартный поток вывода, имя обрабатываемого файла печается в файл с именами ошибочных программ с пометкой «(tab.)» начинается обработка следующего файла.

3.6 Техническая реализация

Программа была реализована на языке C++, таким образом она может быть скомпилирована с помощью компилятора GCC, который уже установлен на сервере тестирования, на котором предполагается использовать данное приложение, без необходимости установки дополнительных инструментов.

В программе не было использовано платформо-зависимых решений, что дает возможность при необходимости запускать ее на различных платформах.

Для разбора файла с правилами обработки внутренних состояний анализатора в формате «`.json`» использовалась сторонняя библиотека nlohmann/json [39], представленная в виде заголовочного файла, который включается в исходный код проекта.

Для разбора аргументов командной строки была выбрана библиотека jarro2783/cxxopts [40], которая также распространяется в виде заголовочного файла.

Включенные в исходный код сторонние решения помещены в папку `include` в корневой директории проекта.

4 Руководство пользователя

4.1 Установка и запуск

Программа распространяется в виде исходного кода на платформе GitHub [41]. Для запуска необходимо иметь следующее ПО:

1. ОС GNU/Linux (компиляция и запуск приложения тестировались на платформах Gentoo Linux и Linux Mint, а также Windows 8, Windows 10 и MacOS High Sierra).
2. Компилятор GCC [42] версии не ниже 7 (либо другой компилятор языка C++, поддерживающий стандарт C++17),
3. CMake [43] версии от 3.10.2.

В папке с исходным кодом проекта `c-format-checker` находится скрипт `compile.bash`. (Данный скрипт написан для исполнения в командной оболочке Bash ОС GNU/Linux и может потребовать модификации для корректной работы на другой платформе. Также команды будут отличаться для разных компиляторов. Дальнейшие команды приведены для ОС GNU/Linux с установленными утилитами Cmake, make, компилятором GCC 7) Из папки `c-format-checker` в терминале необходимо выполнить следующие команды (листинг 12):

Листинг 12 — Сборка проекта.

```
1 $ chmod +x compile.bash  # Сделать файл исполняемым.
2 $ ./compile.bash         # Запустить сборку.
```

В результате выполнения скрипта в папке появятся директории `build` (содержит временные файлы `CMake`) и `bin`. В папке `bin` находится исполняемый файл `c-format-checker` (для упрощения вызова приложения можно переместить исполняемый файл в нужную директорию, либо добавить путь к папке `bin` в системную переменную `PATH` и вызывать программу как системную утилиту).

Программа является консольным приложением. Оно имеет следующие параметры вызова (листинг 13):

Листинг 13 — Формат вызова приложения.

```
1 > c-format-checker [-f] ФАЙЛЫ [-q | -v | -d]
```

Программе на вход подаются имена файлов, разделенные пробелом, а также необязательный параметр, обозначающий степень детализированности сообщений об ошибках:

-f, --file Файл исходного текста программы на языке C (явное указание ключа `-f` не обязательно).

-q, --quiet Выводить кратко описание ошибок (по умолчанию).

-v, --verbose Выводить развернутые описания ошибок.

-d, --debug Выводить отладочную информацию.

--help Помощь.

Каждый файл должен содержать исходный код программы на языке C. Список файлов для анализа подается подряд, не разбивается другими ключами. Благодаря возможностям языка командной оболочки Bash, можно запускать обработку файлов, выполнив поиск по сложному запросу (см. листинг 14):

Листинг 14 — Пример вызова приложения.

```
1 # Обработка всех файлов в папке "011 Подсчёт слов в строке"
2 # в режиме verbose:
3 $ c-format-checker "011 Подсчёт слов в строке"/* -v
4 ...
5 # Обработка всех файлов в папке "src" с расширением .c в режиме debug:
6 $ c-format-checker -d \$(find src -name *.c)}
```

Ключи -q, -v, -d, указывающие, насколько подробным должен быть вывод приложения, не могут встречаться в списке аргументов более одного, в противном случае будет выведено сообщение с подсказкой, и программа завершится с кодом ошибки 1. Если ни один из ключей не передан, вывод производится в режиме `verbose`.

Каждый файл из переданного списка обрабатывается программой. Вывод сообщений по умолчанию производится в стандартный поток (консоль). В результате выполнения скрипта в директории, из который производился запуск приложения, появится файл `cfc-statistics.txt`, содержащий список файлов, в которых были обнаружены некорректные отступы. Если в файле встречается хотя бы одна *непустая*[1] строка кода с отступом, в котором содержится символ табуляции, напротив имени этого файла в документе `cfc-statistics.txt` ставится пометка «(tab.)».

[1]Здесь под непустой строкой подразумевается строка, содержащая текста помимо пробельных символов, комментариев и директив препроцессора. Указанные элементы игнорируются.

4.2 Вывод приложения и сообщения об ошибках

Программа выделяет два вида ошибок при расстановке отступов:

1. Строка имеет иной отступ, чем строки выше на том же уровне вложенности, либо находится левее относительно выражения, расположенного на верхнем уровне вложенности по отношению к данной.
2. Использование табуляций (так как нет единого стандарта о ширине символа при отображении).

Если программа обнаружила только ошибки первого вида, сообщения о них выводятся в соответствующем режиме в стандартный поток вывода и в файл `cfc-statistics.txt` записывается имя файла.

Если в процессе анализа найдена хотя бы одна ошибка второго вида, выдается соответствующее сообщение, в файл `cfc-statistics.txt` записывается имя файла с пометкой «(tab.)».

4.2.1. Ключ --quiet

В режиме `quiet` (является форматом вывода по умолчанию) для каждого файла в стандартный поток выводится краткая сводка об обнаруженных некорректных отступах, а также сообщения об ошибках, возникших в процессе выполнения программы. Об ошибках отступов сообщается в следующем формате: «<имя файла> (строка <номер>): <сообщение>.»

«Сообщение» — предложение из следующего списка:

– Использование табуляции.
– Различные отступы на одном уровне вложенности.
– Ошибка отступа при переносе выражения.
– Вложенное выражение левее объемлющего.

Пример вывода приложения приведен в листинге 15.

Листинг 15 — Запуск приложения в режиме `quiet`.

```
1 > c-format-checker 'Test 1'/* -q
2 ...
3 Test 1/X.c (строка 20): Различные отступы на одном уровне вложенности.
4 Test 1/X.c (строка 21): Использование табуляции.
5
6 Test 1/Y.c (строка 10): Использование табуляции.
7
8 ...
```

Если ошибок не обнаружено, программа ничего не выводит и завершается с кодом 0.

4.2.2. Ключи --verbose и --debug

В режиме `verbose` в стандартный поток выводятся развернутые сообщения о некорректных отступах и сообщения об ошибках. Сообщения имеют следующий вид: «<имя файла> (строка <номер>): ошибка: отступ ширины <число>: <образ>. <Подсказка>».

«Образ» — печатное представление символов отступа: «␣» для пробела и «\t» для табуляции. При вычислении величины отступа ширина табуляции в программе берется равной четырем пробелам.

«Подсказка» — это сообщение, описывающее ошибку. Если в отступе в начале строки встретился символ табуляции, выдается сообщение «Использование пробелов и табуляций в одном отступе» или «Использование табуляций (ранее использовались пробелы)».

Если ошибка связана с тем, что какая-то строка имеет отступ, не равный таковому для строки выше на том же уровне вложенности, подсказка имеет вид: «Ранее на том же уровне вложенности в строке <номер сроки выше> отступ ширины <число>: <образ отступа>.»

Если при переносе выражения на следующую строку продолжение оказалось левее начала выражения, дополнительно выводится сообщение «Продолжение выражения левее предыдущей строки.» Пример вывода приложения в режиме `verbose` приведен в листинге 16.

Листинг 16 — Запуск приложения в режиме `verbose`.

```
1 > c-format-checker 'Test 1'/* -v
2 Test 1/X.c (строка 20): ошибка: отступ ширины 8: <␣␣␣␣␣␣␣␣>. Ранее на
    том же уровне вложенности в строке 19 отступ ширины 4: <␣␣␣␣>.
3 Test 1/X.c (строка 21): ошибка: отступ ширины 4: <\t>. Использование
    табуляций (ранее использовались пробелы).
4
5 Test 1/Y.c (строка 10): ошибка: отступ ширины 8: <\t␣␣␣␣>.
    Использование пробелов и табуляций в одном отступе..
```

Соответствие выводимых сообщений в режимах `quiet` и `verbose` при различных ошибках приведено в таблице 3.

В режиме `debug` в стандартный поток вывода выводится вся отладочная информация.

Таблица 3 — Сопоставление сообщений об ошибках в режимах `quiet` и `verbose`.

Режим `quiet`	Режим `verbose`
”Вложенное выражение левее объемлющего.”	”Вложенное выражение левее объемлющего.”
”Ошибка отступа при переносе выражения.”	”Продолжение выражения левее предыдущей строки.”
”Использование табуляции.”	”Использование пробелов и табуляций в одном отступе.”
	”Использование табуляций (ранее использовались пробелы).”
”Различные отступы на одном уровне вложенности.”	Ранее на том же уровне вложенности в строке <номер сроки выше> отступ ширины <число>: <образ отступа>.

Эти данные содержат список токенов и вывод правил грамматики, полученные в результате лексического и синтаксического разбора, а также промежуточные таблицы анализатора, содержащие уровень вложенности, на котором находится каждая строка файла. Сообщения о найденных ошибках отступов выводятся после отладочной информации в таком же формате, как в режиме `verbose`.

4.2.3. Ключ --help

Если программе передан аргумент `–help`, выводится справочная информация (см. листинг 17) и программа завершается с кодом 0. В случае, если параметры были переданы некорректно, программа завершается с кодом 1 («Некорректные аргументы командной строки.»), показав пользователю сообщение с подсказкой: «`Использование: c-format-checker [-f] ФАЙЛЫ [-q | -v | -d]. Запустите «c-format-checker –help» для более подробного описания.`»

Если в процессе работы программы была обнаружена лексическая или синтаксическая ошибка, в стандартный поток вывода будет отправлено сообщение с именем файла, координатами ошибки в файле, ее описанием и функцией, в которой возникло исключение (см. пример в листинге 18). Далее программа переходит к анализу следующего файла. (Однако не следует целенаправлен-

Листинг 17 — Справочное сообщение (запуск приложения с ключом «--help»).

```
НАЗВАНИЕ
      c-format-checker - проверка файла исходных текстов программ на языке C на корректность расстановки отступов.

СИНТАКСИС
      c-format-checker [-f] ФАЙЛЫ [-q | -v | -d]

ОПИСАНИЕ
      Программа сканирует файл и проверяет корректность расстановки отступов
ОПЦИИ
      -f, --file    Файлы исходного текста программы на языке C ключ( -f не требуется, если этот аргумент идет первым).
      -q, --quiet   Выводить кратко описание ошибок по( умолчанию).
      -v, --verbose Выводить развернутые описания ошибок.
      -d, --debug   Выводить отладочную информацию.
          --help    Помощь.

СТАТУС ВЫХОДА
      0   Нормальный выход.
      1   Некорректные аргументы командной строки.
      2   Ошибка чтениязаписи/ в файл.
```

но использовать данную программу для проверки корректности синтаксиса, так как она производит разбор программы по упрощенной грамматике, следовательно, не все ошибки могут быть обнаружены).

Листинг 18 — Пример сообщения об ошибке на этапе синтаксического анализа.

```
Test 2/SE.c: Ошибка на этапе синтаксического анализа: [c-format-checker/src/frontend/parser/parse_rules.cpp, line 327] Error at (9, 78)-(9, 78) - expected SEMICOLON, got END_OF_FILE.
```

4.3 Критерии корректности форматирования кода

Программа считается корректной, если на одном уровне вложенности все строки имеют одинаковый отступ, вложенные выражения располагаются не левее строк с объемлющими выражениями, при переносе строки продолжение выражения располагается не левее его начала. Также программа не должна содержать символов табуляции в начале строки (при этом не накладывается никаких ограничений на их использование в середине или конце строки). Запрет на использование табуляций не накладывается на «пустые» строки, т.е. содержащие только пробельные символы, комментарии и директивы препроцессора.

Следующим уровнем вложенности считаются:

1. Блок.
2. Тело функции.
3. Тело конструкций условных переходов `if`, `else`, `switch`.
4. Тело цикла `for`, `while`, `do`.
5. Пользовательская метка, метки `case` и `default`.
6. Блок кода, следующий за метками `case` и `default` внутри тела `switch`.
7. Тело объявления структуры `struct`, объединения `union` и перечисления `enum`.

Не считаются следующим уровнем вложенности:

1. Список инициализации.
2. Перенос выражения на следующую строку.
3. Выражения в скобках.
4. Условные выражения в конструкциях `if`, `switch`, `for`, `while`.

Случаи, в которых ослабляется требование равенства всех отступов на одном уровне вложенности:

1. Пользовательские метки (к ним не относятся метки `case` и `default`) могут располагаться левее или правее основного блока кода.
2. При переносе выражения на другую строку ослабляется требование равенства отступов. Перенос считается корректным, если новая строка начинается не левее предыдущей.
3. Внутри блока `switch` метки `case` и `default` могут располагаться левее прочего кода в том же блоке, но на одном уровне друг с другом и не левее самого оператора `switch`.
4. Внутри блока `switch` код, который предшествует самой первой метке `case`, имеет собственный уровень вложенности и не обязан совпадать по ширине отступа с метками или последовательностью кода после них.

Примеры корректого и некорректного форматирования тела оператора `switch` приведены на рисунке 9.

Из приведенного рисунка видно, что в пределах заданных правил существует множество различных вариантов корректного форматирования кода.

```
1 switch(x)
2 {
3     int i=4;
4     f(i);
5 case 0:
6     i=17;
7 default:
8     i=2;
9 }
```

а) Корректно (стиль из стандарта ISO/IEC9899:2017 [37, с. 109])

```
1 switch(x)
2 {
3     int i=4;
4     f(i);
5     case 0:
6         i=17;
7     default:
8         i=2;
9 }
```

б) Корректно (форматирование среды разработки CLion)

```
1 switch(x)
2 {
3     int i=4;
4 f(i);
5     case 0:
6           i=17;
7     default:
8         i=2;
9 }
```

в) Некорректно

Рисунок 9 — Пример корректного (а, б) и некорректного (в) форматирования тела оператора `switch`.

5 Тестирование

Тестирование проводилось в несколько этапов:

1. Тестирование на искусственно созданных простых примерах в режиме debug.
2. Тестирование на большом объеме реальных данных.
3. Анализ программы сторонними инструментами на предмет ошибок и утечек памяти.

Программа была проанализирована следующими инструментами (многие из них были описаны в обзоре предметной области):

1. Clang Static Analyzer показал наличие одной неиспользуемой переменной;
2. clang-tidy не выявил проблем в коде;
3. Cppcheck также не выдал сообщений об ошибках или предупреждений;
4. Valgrind [44] при запуске на реальных данных (см. раздел «Тестирование на реальных данных») не обнаружил ошибок или утечек памяти;
5. Анализатор кода SonarQube выдал следующее заключение: обнаружено 0 ошибок (bugs), уровень безопасности A (уязвимостей не обнаружено), 0% дублирования кода [45].

5.1 Тестирование на искусственных примерах

Тест 1: Ошибки уровня вложенности

Покажем, как программа определяет ошибки отступов, связанные с уровнем вложенности. Пример программы, где встречаются сразу две ошибки: вложенное выражение левее объемлющего и разный отступ на одном уровне вложенности — представлен в листинг 19. Полный вывод приложения в режиме debug представлен в листинге 24 в Приложении А.

Рассмотрим часть вывода приложения, относящуюся к стадии анализа отступов (листинг 20). Программа корректно определила уровень вложенности каждой строки. После сортировки строк по уровню вложенности несоответствие отступов в строках 3 и 5 становится очевидным.

Листинг 19 — Пример программы, содержащей ошибки уровня вложенности `nested.c`.

```
1 int main() {
2     if (x < 2)
3   printf(”Hello”);
4     else
5         return 0;
6     return 1;
7 }
```

Программа вернула сообщение об обеих ошибках: о нарушении правила «строки на большем уровне вложенности должны иметь больший отступ» и о неравенстве отступов на одном уровне вложенности.

Листинг 20 — Результат анализа программы `nested.c`.

```
1  ...
2  line:  indent [state] {tokens}
3  1:  0 [ ] {int, main, (, ), {, }
4  2:  4 [ BLOCK ] {if, (, x, <, 2, ), }
5  3:  2 [ BLOCK IF_ELSE_WHILE_DO ] {printf, (, ”Hello”, ), ;, }
6  4:  4 [ BLOCK ] {else, }
7  5:  8 [ BLOCK IF_ELSE_WHILE_DO ] {return, 0, ;, }
8  6:  4 [ BLOCK ] {return, 1, ;, }
9  7:  0 [ ] {}, }
10 8:  0 [ ] {<EOF>, }
11
12 line: [state] indent type coords: image
13 1:  [ ] 0 NEWLINE (1, 1)–(1, 1): <>
14 7:  [ ] 0 NEWLINE (7, 1)–(7, 1): <>
15 2:  [ BLOCK ] 4 NEWLINE (2, 1)–(2, 5): <␣␣␣␣>
16 4:  [ BLOCK ] 4 NEWLINE (4, 1)–(4, 5): <␣␣␣␣>
17 6:  [ BLOCK ] 4 NEWLINE (6, 1)–(6, 5): <␣␣␣␣>
18 3:  [ BLOCK IF_ELSE_WHILE_DO ] 2 NEWLINE (3, 1)–(3, 3): <␣␣>
19 5:  [ BLOCK IF_ELSE_WHILE_DO ] 8 NEWLINE (5, 1)–(5, 9): <␣␣␣␣␣␣␣␣>
20
21 nested.c (строка 3): ошибка: отступ ширины 2: <␣␣>. Вложенное выражение
       левее объемлющего.
22 nested.c (строка 5): ошибка: отступ ширины 8: <␣␣␣␣␣␣␣␣>. Ранее на том
       же уровне вложенности в строке 3 отступ ширины 2: <␣␣>.
```

Тест 2: Пример `dummy.c` Проверим, действительно ли разработанный инструмент лучше, чем рассмотренные анализаторы кода, приведенные в обзоре.

Запустим программу в режиме `debug` для файла `dummy.c` из листинга 1 на странице 11. Вывод приложения приведен в листинге 21 (из листинга исключнгы сообщения, сгенерированные на этапе синтаксического анализа).

Мы видим, что, в отличие от рассмотренных анализаторов, реализованная программа обнаружила все заложенные в файл ошибки: различие отступов на одном уровне вложенности в строках 4 и 5 (так как программа счита-

Листинг 21 — Результат анализа программы `dummy.c`.

```
...
line:  indent [state] {tokens}
1:  0 [ ] {int, main, (, ), {, }
3:  0 [ BLOCK ] {int, a, =, 1, ;, }
4:  4 [ BLOCK ] {if, (, a, ==, 2, ), }
5:  4 [ BLOCK ] {{, }
6:  1 [ BLOCK BLOCK ] {return, 2, ;, }
7:  4 [ BLOCK ] {}, }
8:  4 [ BLOCK ] {if, (, 1, ), }
9:  8 [ BLOCK IF_ELSE_WHILE_DO ] {if, (, 2, ), return, 2, ;, }
10:  4 [ BLOCK IF_ELSE_WHILE_DO ] {else, }
11:  8 [ BLOCK IF_ELSE_WHILE_DO IF_ELSE_WHILE_DO ] {return, 3, ;, }
12:  4 [ BLOCK ] {return, 0, ;, }
13:  0 [ ] {}, }
15:  0 [ ] {<EOF>, }

line: [state] indent type coords: image
1:  [ ] 0 NEWLINE (1, 1)–(1, 1): <>
13:  [ ] 0 NEWLINE (13, 1)–(13, 1): <>
3:  [ BLOCK ] 0 NEWLINE (3, 1)–(3, 1): <>
4:  [ BLOCK ] 4 NEWLINE (4, 1)–(4, 5): <␣␣␣␣>
5:  [ BLOCK ] 4 NEWLINE (5, 1)–(5, 5): <␣␣␣␣>
7:  [ BLOCK ] 4 NEWLINE (7, 1)–(7, 5): <␣␣␣␣>
8:  [ BLOCK ] 4 NEWLINE (8, 1)–(8, 5): <␣␣␣␣>
12:  [ BLOCK ] 4 NEWLINE (12, 1)–(12, 5): <␣␣␣␣>
6:  [ BLOCK BLOCK ] 1 NEWLINE (6, 1)–(6, 2): ▫<>
9:  [ BLOCK IF_ELSE_WHILE_DO ] 8 NEWLINE (9, 1)–(9, 9): <␣␣␣␣␣␣␣␣>
10:  [ BLOCK IF_ELSE_WHILE_DO ] 4 NEWLINE (10, 1)–(10, 5): <␣␣␣␣>
11:  [ BLOCK IF_ELSE_WHILE_DO IF_ELSE_WHILE_DO ] 8 NEWLINE (11, 1)–(11,
   9): <␣␣␣␣␣␣␣␣>

dummy.c строка( 4): ошибка: отступ ширины 4: <␣␣␣␣>. Ранее на том же
   уровне вложенности в строке 3 отступ ширины 0: <>.
dummy.c строка( 5): ошибка: отступ ширины 4: <␣␣␣␣>. Ранее на том же
   уровне вложенности в строке 3 отступ ширины 0: <>.
dummy.c строка( 6): ошибка: отступ ширины 1: <␣>. Вложенное выражение
   левее объемлющего.
dummy.c строка( 7): ошибка: отступ ширины 4: <␣␣␣␣>. Ранее на том же
   уровне вложенности в строке 3 отступ ширины 0: <>.
dummy.c строка( 8): ошибка: отступ ширины 4: <␣␣␣␣>. Ранее на том же
   уровне вложенности в строке 3 отступ ширины 0: <>.
dummy.c строка( 10): ошибка: отступ ширины 4: <␣␣␣␣>. Ранее на том же
   уровне вложенности в строке 9 отступ ширины 8: <␣␣␣␣>.
dummy.c строка( 12): ошибка: отступ ширины 4: <␣␣␣␣>. Ранее на том же
   уровне вложенности в строке 3 отступ ширины 0: <>.
```

ет первую встреченную строку «эталонной», все остальные строки на том же уровне вложенности были также отмечены как ошибочные), некорректный отступ вложенного утверждения по отношению к объемлющему в строке 6 и различные уровни вложенности для ветвей оператора `if-else` в строке 9.

Случай, когда тело условного оператора или цикла не заключено в скобки, но отступы расставлены так, будто несколько выражений относятся к этому бло-

ку, в реальных тестовых данных встречался не единожды. Благодаря тому, что разработанный инструмент сообщает об ошибке в случаях, когда утверждения на одном уровне вложенности имеют разный отступ, он может использоваться как дополнительная помощь студентам в поиске ошибок в своих программах.

Тест 3: Перенос утверждения, метка case и тернарный оператор

Как было сказано ранее (см. стр. 39), на этапе синтаксического анализа распознавание меток производится с помощью эвристики по шаблону «`IDENT :`». Приведем пример, показывающий, что распознавание производится корректно, и такие конструкции, как тернарный оператор «`?:`» в условии метки `case`, распознаются правильно.

Листинг 22 — Пример программы, содержащий одновременно метки и тернарные операторы `label.c`.

```
1  void main(void)
2  {
3      if (1 ? 1 : 0)
4          if1  // Имя метки и двоеточие – одно утверждение
5            :  // перенос
6          printf(”if 1 printf1\n”);  // printf(...) является одним
       утверждением с меткой => относится к оператору if
7
8      switch(1) {
9          case 1 ? 1  // Тернарный оператор в метке
10                : 0: {
11             printf(”hello\n”);
12         }
13     }
14 }
```

Для данного примера программа не показала наличие ошибок, так как текст полностью удовлетворяет требованием корректности программы в постановке задачи.

Как и в предыдущем примере, приведем только часть вывода приложения (полный текст содержит более 300 строк), относящуюся к синтаксическому разбору тернарных операторов (листинг 23). Угловыми скобками показана глубина стека вызовов, разбираемое выражение для наглядности выделено символом «@» (так как он не является зарезервированным символом языка C).

По аналогичной схеме (ручной анализ вывода приложения в режиме `debug`) проводились, в частности, следующие проверки:

– корректность разбора операторов `if-else`, `while`, `do-while`, `for`, `switch` с пустым условием и/или телом;

Листинг 23 — Результат анализа программы `labels.c`.

```
...
>>>>>>> parse_word_sequence, next = NUM (3, 17)–(3, 18): <0>


@1 ? 1 : 0@
>>>>>>> parse_word_sequence, next = RPAREN (3, 18)–(3, 19): <)>

...

>>>>>>> parse_statement, first = CASE (9, 9)–(9, 13): <case>
>>>>>>>> parse_labeled_statement, first = CASE (9, 9)–(9, 13): <case>
>>>>>>>>> parse_word_sequence, first = IDENT (9, 14)–(9, 15): <A>
>>>>>>>>> parse_word_sequence, first = IDENT (9, 18)–(9, 19): <B>
@B  // Тернарный оператор в метке
                : @
>>>>>>>>> parse_word_sequence, next = IDENT (10, 18)–(10, 19): <C>


@A ? B  // Тернарный оператор в метке
                : C@
>>>>>>>>> parse_word_sequence, next = COLON (10, 19)–(10, 20): <:>

...

@case A ? B  // Тернарный оператор в метке
                : C: {
            printf(”hello\n”);
        }
    @
>>>>>>>> parse_labeled_statement, next = RBRACE (13, 5)–(13, 6): <}>
```

– корректность разбора тех же операторов, если их тело состоит из одного утверждения или блока;
– распознавание объявления функции в файле и в теле другой функции;
– распознавание объявлений типов данных `struct`, `enum`, `union` в файле и внутри выражения;
– выявление переносов строк внутри объявлений вышеописанных операторов и типов данных;
– корректность исключения директив препроцессора, в том числе разбивающих утверждения на строки;
– случай нескольких функций иобъявлений типов данных в файле;
– корректность разбора списков инициализации, в том числе в случаях, когда перед ними стоит выражение в скобках (случай, в построенной грамматике похожий на шаблон объявления функции);
– различные варианты расстановки отступов в блоках `switch` (см. рис. 9 в разделе «Руководство пользователя»);

– выявление случаев некорректного переноса строки в арифметических выражениях, списках инициализации, заголовках функций, объявлений операторов `if-else`, `while`, `do-while`, `for`, `switch` и типов данных `struct`, `enum`, `union`.

5.2 Тестирование на реальных данных

Для тестирования кафедрой были предоставлены реальные решения, присланные студентами на сервер тестирования. Каждое решение представляет собой файл с расширением «`.c`», файлы сгруппированы по задачам. Всего было получено 5 архивов, для каждого из которых была запущена пакетная обработка. Файлы содержали в среднем от 20 до 100 строк в зависимости от задачи. Время выполнения измерялось с помощью утилиты командной строки `time`, из предоставляемых ей метрик использовалось реальное время выполнения.

Таблица 4 — Результат тестирования программы на реальных примерах.

Задача	Файлов всего	Обнаружено файлов с ошибкой	Из них с ошибкой использования табуляции	Время выполнения, с
011 Подсчёт слов в строке	989	619	453	0,342
091 Фибоначчиевы строки	353	207	136	0,284
094 Наибольший простой делитель	397	260	160	0,328
095 Кратчайшая суперстрока	939	706	633	1,208
148 Полином	436	230	157	0,254

После выполнения программы для каждой директории, результаты, сохраненные в файле `cfc-statistics.txt` анализировались вручную. Из полученного списка документов выбирались случайные 20 файлов, не содержащих ошибку табуляции, и проверялись на соответствие реальных ошибок и обнаруженных программой. Таким образом было проверено на корректность работы порядка 100 файлов.

Среди обрабатываемых данных встречались документы, для которых программа обнаружила синтаксическую ошибку. В архивах «095 Кратчайшая

суперстрока» и «011 Подсчёт слов в строке», файлы были помечены информацией о статусе решения, и в них все файлы с обнаруженной синтаксической ошибкой содержали пометку `DRAFT` или `FAILED_TESTS` (что говорит о том, что присланное решение не прошло тесты). В оставшихся архивах было мало файлов с синтаксическими ошибками (до 10), и все они также были проверены вручную на наличие найденных ошибок.

Очевидно, что зависимость времени выполнения от количества документов нелинейная. Скорость анализа зависит от количества строк кода в каждом документе, а также сложности использованных в тексте тестируемой программы синтаксических конструкций. Тем не менее из полученных данных можно судить о том, что программа имеет высокую скорость выполнения, что является положительным показателем при использовании на сервере, обрабатывающем большие объемы данных.

ЗАКЛЮЧЕНИЕ

В данной работе стояла задача создать программу для проверки корректности расстановки отступов в исходных текстах программ на языке C. Данный инструмент разрабатывался как часть системы автоматического тестирования с целью снизить нагрузку на проверяющего, допуская до ручной проверки только хорошо отформатированный и легко читаемый код.

В процессе исследования было выяснено, что для решения данной задачи существующие инструменты не подходят по ряду причин. Был проведен подробный разбор схемы работы существующих анализаторов кода и разработан алгоритм решения поставленной задачи.

Результатом работы стало приложение, позволяющее проводить пакетный анализ файлов исходных текстов программ с несколькими режимами детализированности вывода сообщений и собирать статистику по ним.

К положительным качествам реализованной программы можно отнести устойчивость анализатора к использованию в исходном тексте нестандартных расширений, которая была достигнута за счет переработки грамматики языка C, а также высокую скорость работы.

В качестве дальнейших направлений работы можно указать, во-первых, усовершенствование критериев корректности расстановки отступов, так, чтобы поддерживать больше стилей оформления кода, а во-вторых, продолжение тестирования разработанного приложения как в ручном режиме, так и в автоматическом.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Lee T., Lee J. B., In H. P. A Study of Different Coding Styles Affecting Code Readability // International Journal of Software Engineering and Its Applications. Gothenburg, Sweden, 2013. Vol. 7, no. 5. P. 413–422. Access: https://pdfs.semanticscholar.org/1727/4fef400424fe4876cd23edb8318e4944b203.pdf.

[2] Dos Santos R. M., Gerosa M. A. Impacts of Coding Practices on Readability // ICPC'18: 26th IEEE/ACM International Conference on Program Comprehension. Gothenburg, Sweden, 2018. 9 p. Access: https://www.ime.usp.br/~gerosa/papers/ICPC2018-Legibility.pdf.

[3] Google C++ Style Guide [Электронный ресурс]. Режим доступа: https://google.github.io/styleguide/cppguide.html.

[4] Qt Coding Style [Электронный ресурс]. URL: https://wiki.qt.io/Qt_Coding_Style (дата обращения: 30.05.2019).

[5] Compilers : principles, techniques, and tools. Second Edition / A. V. Aho, M. S. Lam, R. Sethi et al. Vrije Universiteit, Amsterdam, 2007. 1010 p.

[6] Chess B., West J. Security programming with static analisis. Addison-Wesley, 2007. 587 p. Access: https://www.e-reading.club/bookreader.php/142130/Secure_programming_with_Static_Analysis.p

[7] Kunst F. Lint, a C Program Checker. Vrije Universiteit, Amsterdam, 1988. 24 p. Access: http://tack.sourceforge.net/olddocs/lint.pdf.

[8] CLion: A Cross-Platform IDE for C and C++ by JetBrains [Электронный ресурс]. Режим доступа: https://www.jetbrains.com/clion/.

[9] Qt | Cross-platform software development for embedded & desktop [Электронный ресурс]. Режим доступа: https://www.qt.io/.

[10] Enabling Open Innovation & Collaboration | The Eclipse Foundation [Электронный ресурс]. Режим доступа: https://www.eclipse.org/.

[11] Sublime Text - A sophisticated text editor for code, markup and prose [Электронный ресурс]. Режим доступа: https://www.sublimetext.com/.

[12] Notepad++ Home [Электронный ресурс]. Режим доступа: https://notepad-plus-plus.org/.

[13] Kate - Advanced Text Editor - KDE.org [Электронный ресурс]. Режим доступа: https://kde.org/applications/utilities/kate/.

[14] welcome home: vim online [Электронный ресурс]. Режим доступа: https://www.vim.org/.

[15] vim tips and tricks: indenting [Электронный ресурс]. Режим доступа: https://www.cs.oberlin.edu/~kuperman/help/vim/indenting.html.

[16] Chiel92/vim-autoformat: Provide easy code formatting in Vim by integrating existing code formatters [Электронный ресурс]. Режим доступа: https://github.com/Chiel92/vim-autoformat.

[17] rhysd/vim-clang-format: Vim plugin for clang-format, a formatter for C, C++, Obj-C, Java, JavaScript, TypeScript and ProtoBuf [Электронный ресурс]. Режим доступа: https://github.com/rhysd/vim-clang-format.

[18] ClangFormat — Clang 9 documentation [Электронный ресурс]. Режим доступа: https://clang.llvm.org/docs/ClangFormat.html.

[19] Artistic Style — Index [Электронный ресурс]. Режим доступа: http://astyle.sourceforge.net/.

[20] Uncrustify — Source Code Beautifier for C, C++, C#, ObjectiveC, D, Java, Pawn and VALA [Электронный ресурс]. Режим доступа: http://uncrustify.sourceforge.net/.

[21] BCPP — C(++) Beautifier [Электронный ресурс]. Режим доступа: https://invisible-island.net/bcpp/.

[22] GC GreatCode Download | SourceForge.net [Электронный ресурс]. Режим доступа: https://sourceforge.net/projects/gcgreatcode/.

[23] Indent - GNU Project - Free Software Foundation [Электронный ресурс]. Режим доступа: https://www.gnu.org/software/indent/.

[24] Clang-Tidy — Extra Clang Tools 9 documentation [Электронный ресурс]. Режим доступа: https://clang.llvm.org/extra/clang-tidy/.

[25] Clang Static Analyzer [Электронный ресурс]. Режим доступа: https://clang-analyzer.llvm.org/.

[26] cpplint at gh-pages - google/styleguide [Электронный ресурс]. Режим доступа: https://github.com/google/styleguide/tree/gh-pages/cpplint.

[27] Cppcheck - a took for static C/C++ analisis [Электронный ресурс]. Режим доступа: http://cppcheck.sourceforge.net/.

[28] Splint Home Page [Электронный ресурс]. Режим доступа: https://splint.org/.

[29] Clang: a C language family frontend for LLVM [Электронный ресурс]. Режим доступа: https://clang.llvm.org/index.html.

[30] clang-tidy — Clang-Tidy Checks — Extra Clang Tools 9 documentation [Электронный ресурс]. Режим доступа: https://clang.llvm.org/extra/clang-tidy/checks/list.html.

[31] cpplint at gh-pages - google/styleguide [Электронный ресурс]. Режим доступа: https://clang.llvm.org/docs/ClangFormatStyleOptions.html.

[32] Goyvaerts J., Levithan S. Regular Expressions Cookbook, Second Edition. 2012. 595 p. Access: https://doc.lagout.org/programmation/Regular

[33] Stroustrup B. The C++ Programming Language. Fourth Edition. 2013. 1347 p. Access: https://anekihou.se/programming/2.

[34] Annotation Overview | Microsoft Docs [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2008/ms182033(v=vs.90).

[35] Source Annotations [Электронный ресурс]. Режим доступа: https://clang-analyzer.llvm.org/annotations.html.

[36] clang-format. Automatic formatting for C++ (Daniel Jasper - djasper@google.com) [Электронный ресурс]. Режим доступа: https://llvm.org/devmtg/2013-04/jasper-slides.pdf.

[37] INTERNATIONAL STANDARD. Programming languages – C. ISO/IEC9899:2017 [Электронный ресурс]. Режим доступа: https://web.archive.org/web/20181230041359if_/http://www.open-std.org/jtc1/sc22/wg14/www/abq/c17_updated_proposed_fdis.pdf. (дата обращения: 2.06.2019).

[38] Zaytsev Vadim. BNF WAS HERE: What Have We Done About the Unnecessary Diversity of Notation for Syntactic Definitions // SAC'12. Riva del Garda, Italy, 2012. 6 c. Access: https://www.grammarware.net/text/2012/bnf-was-here.pdf.

[39] nlohmann/json - JSON for Modern C++ [Электронный ресурс]. Режим доступа: https://nlohmann.github.io/json/.

[40] jarro2783/cxxopts - Lightweight C++ command line option parser [Электронный ресурс]. Режим доступа: https://github.com/jarro2783/cxxopts.

[41] Исходный код приложения c-format-checker [Электронный ресурс]. Режим доступа: https://github.com/anastasiarazb/c-format-checker.

[42] GCC, the GNU Compiler Collection [Электронный ресурс]. Режим доступа: https://gcc.gnu.org/.

[43] CMake [Электронный ресурс]. Режим доступа: https://cmake.org/.

[44] Valgrind Home. Режим доступа: http://www.valgrind.org/.

[45] Sonar - Continuous Code Quality | c-format-checker - anastasiarazb. Режим доступа: https://sonarcloud.io/dashboard?id=anastasiarazb_c-format-checker.

ПРИЛОЖЕНИЕ А

Листинг 24 — Вывод программы в режиме `debug` при анализе файла `nested.c` (см. листинг 19).

```
Tokens:

0:  0 [ ] {\n, int, main, (, ), {, \n␣␣␣␣␣, if, (, x, <, 2, ), \n␣␣,
    printf, (, "Hello", ), ;, \n␣␣␣␣␣, else, \n␣␣␣␣␣␣␣␣␣, return, 0, ;, \
    n␣␣␣␣␣, return, 1, ;, \n, }, \n, <EOF>, }
Parse top level
> parse_statement, first = TYPENAME (1, 1)–(1, 4): <int>
>> parse_simple_expr, first = TYPENAME (1, 1)–(1, 4): <int>
>>> parse_word_sequence, first = TYPENAME (1, 1)–(1, 4): <int>
>>>> parse_word_sequence, first = RPAREN (1, 10)–(1, 11): <)>
@@
>>>> parse_word_sequence, next = RPAREN (1, 10)–(1, 11): <)>


>>>> parse_block, first = LBRACE (1, 12)–(1, 13): <{>
>>>>> parse_statement, first = IF (2, 5)–(2, 7): <if>
>>>>>> parse_selection_statement, first = IF (2, 5)–(2, 7): <if>
>>>>>>> parse_word_sequence, first = IDENT (2, 9)–(2, 10): <x>
@x < 2@
>>>>>>> parse_word_sequence, next = RPAREN (2, 14)–(2, 15): <)>


>>>>>>> parse_statement, first = IDENT (3, 3)–(3, 9): <printf>
>>>>>>>> parse_simple_expr, first = IDENT (3, 3)–(3, 9): <printf>
>>>>>>>>> parse_word_sequence, first = IDENT (3, 3)–(3, 9): <printf>
>>>>>>>>>> parse_word_sequence, first = STRING (3, 10)–(3, 17): <"Hello
    ">
@"Hello"@
>>>>>>>>>> parse_word_sequence, next = RPAREN (3, 17)–(3, 18): <)>


@printf("Hello")@
>>>>>>>>> parse_word_sequence, next = SEMICOLON (3, 18)–(3, 19): <;>


@printf("Hello");
    @
>>>>>>>> parse_simple_expr, next = ELSE (4, 5)–(4, 9): <else>


@printf("Hello");
    @
>>>>>>> parse_statement, next = ELSE (4, 5)–(4, 9): <else>


>>>>>>> parse_statement, first = KEYWORD (5, 9)–(5, 15): <return>
>>>>>>>> parse_simple_expr, first = KEYWORD (5, 9)–(5, 15): <return>
>>>>>>>>> parse_word_sequence, first = KEYWORD (5, 9)–(5, 15): <return>
@return 0@
>>>>>>>>> parse_word_sequence, next = SEMICOLON (5, 17)–(5, 18): <;>
```

Продолжение листинга 24, часть 1.

```
@return 0;
    @
>>>>>>>> parse_simple_expr, next = KEYWORD (6, 5)–(6, 11): <return>


@return 0;
    @
>>>>>>> parse_statement, next = KEYWORD (6, 5)–(6, 11): <return>


@if (x < 2)
  printf(”Hello”);
    else
        return 0;
    @
>>>>>> parse_selection_statement, next = KEYWORD (6, 5)–(6, 11): <return
    >


@if (x < 2)
  printf(”Hello”);
    else
        return 0;
    @
>>>>> parse_statement, next = KEYWORD (6, 5)–(6, 11): <return>


>>>>> parse_statement, first = KEYWORD (6, 5)–(6, 11): <return>
>>>>>> parse_simple_expr, first = KEYWORD (6, 5)–(6, 11): <return>
>>>>>>> parse_word_sequence, first = KEYWORD (6, 5)–(6, 11): <return>
@return 1@
>>>>>>> parse_word_sequence, next = SEMICOLON (6, 13)–(6, 14): <;>


@return 1;
@
>>>>>> parse_simple_expr, next = RBRACE (7, 1)–(7, 2): <}>


@return 1;
@
>>>>> parse_statement, next = RBRACE (7, 1)–(7, 2): <}>


@{
    if (x < 2)
  printf(”Hello”);
    else
        return 0;
    return 1;
}
@
>>>> parse_block, next = END_OF_FILE (8, 1)–(8, 1): <<EOF>>
```

Продолжение листинга 24, часть 2.

```
@int main() {
    if (x < 2)
  printf(”Hello”);
    else
        return 0;
    return 1;
}
@
>>> parse_word_sequence, next = END_OF_FILE (8, 1)–(8, 1): <<EOF>>


@int main() {
    if (x < 2)
  printf(”Hello”);
    else
        return 0;
    return 1;
}
@
>> parse_simple_expr, next = END_OF_FILE (8, 1)–(8, 1): <<EOF>>


@int main() {
    if (x < 2)
  printf(”Hello”);
    else
        return 0;
    return 1;
}
@
> parse_statement, next = END_OF_FILE (8, 1)–(8, 1): <<EOF>>


line:  indent [state] {tokens}
1:  0 [ ] {int, main, (, ), {, }
2:  4 [ BLOCK ] {if, (, x, <, 2, ), }
3:  2 [ BLOCK IF_ELSE_WHILE_DO ] {printf, (, ”Hello”, ), ;, }
4:  4 [ BLOCK ] {else, }
5:  8 [ BLOCK IF_ELSE_WHILE_DO ] {return, 0, ;, }
6:  4 [ BLOCK ] {return, 1, ;, }
7:  0 [ ] {}, }
8:  0 [ ] {<EOF>, }

line: [state] indent type coords: image
1:  [ ] 0 NEWLINE (1, 1)–(1, 1): <>
7:  [ ] 0 NEWLINE (7, 1)–(7, 1): <>
2:  [ BLOCK ] 4 NEWLINE (2, 1)–(2, 5): <␣␣␣␣>
4:  [ BLOCK ] 4 NEWLINE (4, 1)–(4, 5): <␣␣␣␣>
6:  [ BLOCK ] 4 NEWLINE (6, 1)–(6, 5): <␣␣␣␣>
3:  [ BLOCK IF_ELSE_WHILE_DO ] 2 NEWLINE (3, 1)–(3, 3): <␣␣>
5:  [ BLOCK IF_ELSE_WHILE_DO ] 8 NEWLINE (5, 1)–(5, 9): <␣␣␣␣␣␣␣␣>

nested.c (строка 3): ошибка: отступ ширины 2: <␣␣>. Вложенное выражение
   левее объемлющего.
nested.c (строка 5): ошибка: отступ ширины 8: <␣␣␣␣␣␣␣␣>. Ранее на том
   же уровне вложенности в строке 3 отступ ширины 2: <␣␣>.
```