Разработка адаптивного стилевого анализатора программ на языке C++

Автор работы: Уваров Никита

Научный руководитель: Дединский Илья Рудольфович

Введение

Текст любой программы (код) пишется прежде всего для того, чтобы быть скомпилированным, то есть для «прочтения» компилятором. Для этого достаточно соблюдения синтаксических и семантических правил языка программирования. Но после написания и компиляции кода он будет ещё не раз прочитан человеком – как самим разработчиком, спустя некоторое время, так и членами его команды. Чем сложнее (архитектурно или алгоритмически) код, тем тяжелей его читать и тем больше времени требуется на понимание его назначения. Для того, чтобы значительно уменьшить это время, к правильному с точки зрения компилятора коду предъявляется множество дополнительных требований. В данной работе рассматриваются требования, касающиеся *форматирования кода* и *именования переменных*.

Известно, что неопытные разработчики не сразу понимают необходимость следования этим стандартам. Более того, в связи с тем, что они пишут большое количество учебных программ, их «стиль» может менятся от задания к заданию.

Естественным образом возникает необходимость обучения не только умению решать поставленные задачи алгоритмически и технически (то есть владению языком программирования), но и соблюдению общепринятых норм оформления кода.

Качество форматирования кода

Форматированием кода называется расстановка в нём пробельных символов (пробелов, табуляций и переводов строк), так, чтобы они подчёркивали его синтаксическую структуру.

Чтобы получить представление о структуре программы на рис. 2 необходимо, как минимум, заметить фигурные скобки и точки с запятой. При взгляде же на программу на рис. 1 достаточно проследить уровень начала строк, что делается одним взглядом. Чтобы понять, в каком случае выполняется тот или иной блок с одинаковым выравниванием,

достаточно посмотреть на его начало.

Между тем, существует не единственный способ отформатировать программу. Даже основных стилей стилей форматирования (indent styles) больше десятка (например, Allman, K&R, BSD, Whitesmiths, GNU и т.д.). К тому же в пределах проекта стили претерпевают незначительные изменения, удобные данной команде.

Смешение нескольких стилей может создавать значительные неудобства. Например, в случае, когда один из разработчиков модифицирует фрагмент кода, он может случайно или в результате работы его редактора изменить не только строки, в которых содержится исправление, но и соседние. В результате, в некоторых системах контроля версий могут появится неверные сведения об авторе данной строки.

Таким образом, характеристикой качества является не следование какому-то заданному стилю, а всего лишь следование единственному стилю (style consistency) в пределах всей программы.

```
while (true) {
    if (counter > MAX_COUNT) {
        nLoops++;
        counter %= MAX_COUNT;
        if (nLoops > MAX_LOOPS)
            break;
    }
    counter++;
    doWork(counter);
}
```

Рис. 1. Стилистически верный код для стиля K&R

```
while(true) {
if(counter>MAX_COUNT) {
nLoops++;
counter%=MAX_COUNT;
if(nLoops>  MAX_LOOPS)
break;
}
counter ++;
doWork(counter);
}
```

Рис. 2. Код без форматирования

```
while (true) {
    if (C > MC) {
        n++;
        C %= MC;
        if (n > ML)
            break;
    }
    C++;
    d(C);
}
```

Рис. 3. Код с «плохими» именами переменных

Качество имён идентификаторов

С точки зрения компилятора не важно, какая последовательность символов является названием (*идентификатором*) переменной, функции, класса и т.п. При этом если названия переменных и классов подобраны хорошо, в коде программы не требуется комментариев, кроме как в заголовочных файлах. И наоборот – понять, за что отвечает фрагмент на рис. 3

крайне сложно без документации или полного текста программы.

Плохие (не несущие информации о назначении переменной) названия усложняют ориентирование в коде и способствуют появлению ошибок. Если имена объектов в глобальной области видимости имеют распространенные или краткие названия, могут возникнуть неочевидные эффекты скрытия других имён (name shadowing). И то, и другое замедляет разработку программы.

Автоматизированная проверка стиля

С течением времени всё большая часть работы программиста автоматизируется, и задача проверки стиля – не исключение. Наиболее распространены следующие способы проверки:

1. Использование утилит, проверяющих соответствие форматирования стандартам конкретной компании, или некоторому подмножеству заданных правил, таких как **cpplint [9]**, **KWStyle [12]**, **Vera++ [10]**, **cxxchecker [11]**.
2. Сравнение исходный текста программы с результатом работы автоматического форматировщика (такого как **AStyle [14]**).

Недостатки существующих способов проверки

Каждый из перечисленных выше подходов заключается в проверке соответствия программы заданному стилю (либо фиксированному для анализатора, либо настраиваемому). Значит, при организации автоматизированной проверки необходимо настроить анализатор на какой-то единственный стиль.

Такой подход, например, в образовательном процессе, не эффективен: он не развивает у учащихся чувства стиля, не даёт осознания необходимости следования стилю. Напротив, учащиеся начинают считать эту проверку пустой формальностью, пользуются автоматическими форматировщиками перед сдачей программ. При создании же этих программ они пишут код на подобии приведённого на рис. 2 и 3. В результате, во-первых, снижается их собственная продуктивность (хотя они этого зачастую не понимают), а во-вторых, получаемый код низкого качества не может быть использован в реальных проектах без серьёзной модификации.

В результате, преподавателям приходится проверять стиль вручную, что трудоемко,

поскольку число программ велико (около 400 на всего одной практической работы одного потока в МФТИ), вносит в проверку субъективный элемент (в отличие от объективных критериев – правильности работы на тестах и отсутствия ошибок работы с памятью), а также позволяет обнаружить далеко не все стилевые ошибки.

Идея адаптивного анализа

Описанная проблема была бы решена, если бы перед проверкой стиля анализатор мог бы автоматически определить его в данной программе, на основании преобладания способов оформления тех или иных конструкций. Создание такого анализатора позволило бы ещё сильнее автоматизировать процесс рецензирования кода (code review) как в образовательном процессе, так и в промышленном программировании.

Цель работы

Разработать инструмент проверки стиля, не требующего задания стиля перед работой, а определящего стиль на основе анализа преобладающих способов оформления синтаксических конструкций и имён переменных в программе.

Требования

1. *Адаптивность*, то есть возможность автоматического определения преобладающего в программе стиля, а затем нахождения отклонений от него.
2. *Устойчивость* к стилистическим ошибкам и возможность их диагностики.
3. *Непривязанность к конкретному языку*. Этого удалось добиться, разделив анализатор на модуль, осуществляющий сбор информации о программе и зависящий от языка (front end) и модуль, непосредственно выполняющий анализ, не зависящий от языка (back end).

Постановка задачи

Разработать подход для адаптивного автоматического анализа стиля форматирования и именования переменных. Реализовать прототип, анализирующий стиль программ на языке C++ (поскольку именно языкам семейства C/C++ учат на ранних курсах большинства технических вузов).

Необходимость синтаксического анализа

Оказалось, что в любом стиле есть конструкции, расстановку пробелов в которых невозможно объснить на основании только лексических соображений. Например, лексема "`:`" (двоеточие) в языке C++ может выступать как в роли части тернарного оператора "`?:`", так и в списке инициализаторов конструктора. Понятно, что в этих случаях она форматируется по-разному. Аналогично, код в пределах операторов `if`, `for` и т.п. часто сдвинут относительно содержащей оператор области видимости. Для анализа таких сдвигов необходимо уметь выделять операторы цикла и ветвления.

Сама по себе такая необходимость не требует синтаксического анализа. Например, форматировщик **AStyle** анализирует такие конструкции только на основании потока лексем. Однако такой подход чрезвычайно сложно масштабируется и не обеспечивает поддержку произвольного исходного кода (например, не обрабатывает код, интенсивно использующий макросы препроцессора).

В результате, было решено использовать данные синтаксического анализа при анализе стиля.

Модель для анализа стиля форматирования

Анализ форматирования, т. е. расстановки пробелов, являлся наиболее сложной частью проекта, поскольку существует большое количество распространённых нарушений стиля (с формальной точки зрения), которые тем не менее являются нормальным явлением. Наиболее распространёнными являются следующие исключительные ситуации:

1. Написание нескольких операторов на одной строке, так называемые «one-line statements». Например:

```
if (nArguments != 2) return 0

if (cached[argument]) return retrieveCached(argument); // обычно «return» располагается на отдельной строке
```

2. Напротив, вставка перевода строки в длинных конструкциях (перенос). При этом перенесенные строки могут быть дополнительно выделены отступом:

```
if (getHolidaysInMonth(month) + getWeekendsIntMonth(month) >=
       getWorkDaysRequired(month) - getWorkTreshold(month) &&
           useExceptions) // перенос длинного условия оператора «if»
```

3. Выравнивание нескольких строк для подчёркивания в них повторяющихся элементов:

```
const char* file_signature    = "BCDE";
const char* section_signature = "SE"  ; // выравнивание объявлений
const char* header_signature  = "HE"  ; // в виде таблицы
```

4. Смешение чистого и препроцессированного кода, комментарии:

```
#define ifdebug(...) if(debugEnabled(__VA_ARGS__))
ifdebug(__LINE__)
{
     // использование препроцессора
     // для создания собственного условного оператора
}
```

5. Расстановка пустых строк программистом.

```
int nEntries;
scanf("%d", &nEntries);

int nRequests;          // предыдущая строка - пустая
scanf("%d", &nRequest); // она отделяет логические связанные группы
                        // операторов
```

Поведение стилевого анализатора в таких случаях должно было быть максимально толерантным. Иными словами, для использования в образовательных целях, необходимо было минимизировать число ложных отказов (false negatives), возможно, ценой повышения числа ложных принятий (false positives).

Для достижения такого эффекта решено было разбить анализ на две стадии:

1. Первая стадия должна выявить все места, в которых, возможно, реализуются исключительные ситуации, проверить их по специальным (облегчённым) правилам и исключить из дальнейшего анализа.
2. На второй стадии анализатор работает с оставшейся частью кода, применяя к ней общий механизм. Масштабируемость и адаптивность планировалось реализовать с помощью этой стадии.

Поскольку в общем случае многие части программы могут быть исключены из

анализа, стилевой единицей было решено принять *промежуток* между соседними лексемами в программе, то есть последовательность пробельных символов и переводов строки. Существуют промежутки двух типов:

1. Не содержащие перевода строки. Значением таких промежутков является количество пробелов в них. Заметим, что ещё до работы анализатора следуют избавиться от символов табуляции. Например, путём запуска анализатора с несколькими значениями пробелов в табуляции и выбора лучшего результата.
2. Содержащие перевод строки. В этом случае значением промежутка является *относительный* сдвиг двух соседних строк. Так, если после закрывающей скобки оператора **`for`** начало следующей строки сдвинуто на четыре пробела относительно начала предыдущей, значение промежутка равно четырём. Заметим, что оно может быть и отрицательным, если следующая строка начинается левее предыдущей.

Любой промежуток можно считать одним из перечисленных, если:

1. Избавиться от пустых строк на первой стадии.
2. Игнорировать пробелы в конце строк (или выдавать предупреждение).

Входом второй стадии является последовательность промежутков, некоторые из которых были исключены из анализа на первой стадии. Так, на второй стадии не анализируются:

1. Промежутки между лексемами, одна из которых не соответствует препроцессированной лексеме. В результате, макросам препроцессора разрешено произвольным образом влиять на форматирование программы. Это позволяет принимать рассмотренные выше конструкции, в которых макросы реализуют отсутствующие в языке операторы.
2. Промежутки, про которые установлено, что они являются частью переноса длинного оператора, частью выравнивания похожих строк или той частью однострочной конструкции, в которой обычно ставится перенос строки.
3. Промежутки между лексемами, тип одной из которых не поддерживается анализатором. Это позволяет расширять список поддерживаемых конструкций постепенно.

Для анализа на второй стадии было принято за аксиому утверждение: «значение

промежутка полностью определяется типом образующих его лексем». При этом тип лексемы содержит как лексическую, так и синтаксическую информацию. Например, рассмотренная ранее лексема “**:**” в составе различных конструкций будет иметь различный тип.

То, как синтаксическая информация влияет на типы лексем, обеспечивает масштабируемость системы, возможность обучения её исключительным ситуациям. Например, некоторые стили отделяют скобки вызова функции от её имени пробелом: “**f (x)**”, но только в том случае, если у функции есть хотя бы один аргумент. Чтобы добавить поддержку такого стиля, достаточно создать дополнительный тип для одной из скобок, который будет присваиваться в случае, если аргументов нет.

Впрочем, оставались случаи, в которых принятая гипотеза не работала. Такими были промежутки после последней лексемы в области видимости, не ограниченной фигурными скобками:

```
for (int i = 0; i < nCalls; i++)
    if (something)
        statement();
nextStatement(); // При переходе на эту строку уровень отступа
                 // уменьшился на 8 пробелов, но если бы не было операторов
                 // if или for, этого бы не произошло.
                 // Конец области видимости является невидимым модификатором
                 // лексемы “;”, объясняющим уменьшение отступа.
```

Для разрешения и этих случаев к типам лексем может быть добавлен набор *невидимых модификаторов*, возникающих в такой ситуации.

Таким образом, первым шагом второй стадии является назначение типов лексем и модификатором с использованием синтаксической информации.

Формализация задачи анализа форматирования

После того, как назначены типы всех лексем, а некоторые из них получили невидимые модификаторы окончания области видимости, уже можно выдать результат анализа больших программ: достаточно проверить, что одним и тем же парам типов лексем соответствует одно и то же значение промежутка.

Но такой подход не будет работать для маленьких программ, поскольку в них слишком мало промежутков. Может получится, что каждому промежутку соответствует уникальная пара типов, так как количество типов, которые могут соответствовать одной и той же лексеме, достаточно велико. В таком случае анализатор примет программу, даже если в ней есть стилевые ошибки.

Поэтому было решено перейти к следующей формализации задачи. Будем считать, что у каждого типа также есть значение (в пробелах), которое должно следовать непосредственно перед (значение префикса) и после (значение суффикса) лексемы данного типа. Например, в большинстве стилей идентификаторы не отделяются пробелами сами по себе (оба значения равны нулю), а операторы, наоборот, отделяются с двух сторон (оба значения равны единице). Или, допустим, значение суффикса открывающей фигурной скобки часто равно четырём (относительный сдвиг, связанный с началом области видимости). Значение промежутка, таким образом, должно быть равно сумме значений суффикса и префикса типов лексем, соответствующих промежутку. Каждый тип невидимого модификатора также имеет собственное значение, участвующее в сложении.

Итак, можно считать, что есть некоторый набор целочисленных переменных (суффиксы и префиксы различных типов), а каждый промежуток, встречающийся в программе, есть линейное уравнение на эти переменные. При этом коэффициенты в полученных уравнениях равны единице (кроме свободного члена). Также известно, что переменные принимают небольшие значения (не более 4 по модулю для большинства стилей),

Задача переформулируется следующим образом: **«найти значения переменных, удовлетворяющие максимальному количеству данных целочисленных линейных уравнений»**.

Преимущества описанной модели анализа форматирования

1. Для улучшения гибкости анализатора достаточно добавить новые типы лексем.
2. Анализатор не только принимает верно отформатированные программы, но и старается наиболее лояльно оценить программы, содержащие ошибки. В случае, если количество ошибок невелико (по сравнению с размером программы), будут указаны ровно те промежутки, которые их содержат.
3. Удаление из анализа некоторых промежутков в результате первой стадии модели,

очевидно, не ухудшает результат работы анализатора (поскольку удаление уравнений из системы не может ухудшить её разрешимость).

Решение полученной оптимизационной задачи

Полученная задача была решена с помощью перебора с отсечением и использованием парадигмы «разделяй-и-властвуй». А именно, заметим, что зафиксировав и подставив значение некоторой переменной, система может распасться на несколько независимых, в которых можно произвести перебор по отдельности. Использование этого отсечения позволило решать получаемые системы за приемлемое (меньше секунды) время.

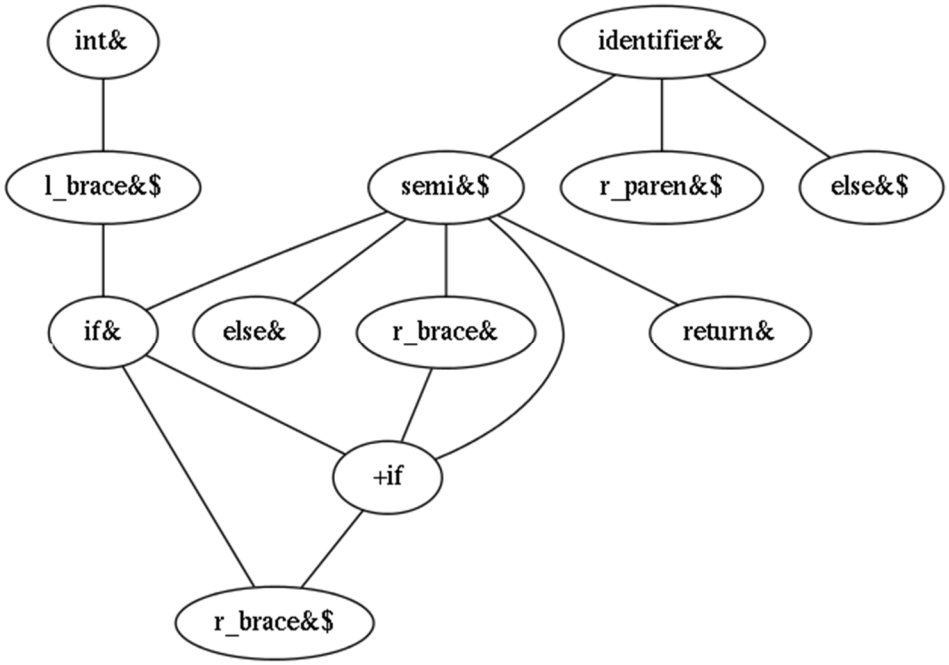

Рис. 4. Граф зависимости между стилевыми переменными

Поскольку полученные линейные уравнения состоят из двух переменных (редко - из большего числа, когда добавляются невидимые модификаторы), можно построить граф зависимостей между переменными. Тогда выгодно начинать перебор с вершин, после удаления которых граф распадается на максимальное количество компонент связности. Пример графа зависимостей, автоматически визуализированного анализатором, приведен на рис. 4.

Символ “+” означает, что вершина соответствует невидимому модификатору; символ “**$**” – что переменная соответствует префиксу некоторого типа лексем (промежуток образуется двумя типами, от одного из них в линейном уравнении участвует переменная-

префикс, от другого – переменная-суффикс); символ “**&**” – что промежутки данной компоненты связности содержат перевод строки.

Модель анализа стиля именования переменных

Данная задача оказалась гораздо проще, поскольку существует не так много способов именования переменных. Каждое имя состоит из одного или нескольких слов. В зависимости от стиля:

1. Отдельные слова могут разделяться символом подчёркивания (`variable_Name`) или писаться слитно (`variableName`).
2. Первое слово может начинаться с прописной (`VariableName`) или строчной буквы (`variableName`).
3. Последующие слова могут начинаться с прописной буквы (`variable_Name`), строчной буквы (`variable_name`) или писаться целиком прописными буквами (`VARIABLE_NAME`).
4. Наконец, переменная может начинаться с префикса венгерской нотации (hungarian notation), например: `ISomeInterface`, `CSomeClass`, `m_classMember`, `g_globalVariable`.

Разумеется, стиль именования идентификаторов зависит от их роли в программе. Поэтому каждый идентификатор, определённый в программе, анализатор относит к одному из классов в зависимости от:

1. Назначения идентификатора (является ли он названием класса, интерфейса, функции, переменной, константы или макроса?).
2. Пространства имён идентификатора.

После этого небольшим перебором определяется наиболее подходящий стиль каждого из классов, а не подходящие под него имена объявляются ошибочными.

Логическая структура анализатора, возможность поддержки нескольких языков

Легко видеть, что анализатор стиля форматирования (на второй стадии) и анализатор именования идентификаторов принимают на вход ограниченное количество информации о

программе:

- на вход второй стадии анализатора форматирования необходимо подать список обрабатываемых промежутков и типы лексем (некоторые – с модификаторами).
- на вход анализатору имён подаётся список идентификаторов и тип (область видимости и роль) каждого идентификатора.

Таким образом, логически стилевой анализатор разделён на модуль, отвечающий за сбор информации о программе (front end) и модуль, реализующий непосредственный анализ (back end). Такая архитектура позволяет реализовать несколько сборщиков информации для разных языков программирования (например, для C++ и для C), но использовать один и тот же back end для её обработки.

Сложности получения синтаксической информации о программах на C++

Поскольку грамматика языка C++ является Тьюринг-полной [4], существует не так много решений, реализующих разбор программ на этом языке. Потребовалось несколько попыток, чтобы найти инструменты, подходящие для получения необходимой информации.

Первая попытка: библиотека VivaCore

В конце 2012 года была разработана первая версия анализатора, использовавшая библиотеку VivaCore [5] для получения необходимой информации. Поскольку рассматриваемая библиотека работала только под Windows, нацеливалась на совместимость с Visual Studio, не обеспечивала полной поддержки языка C++ и была коммерческой (что привело к её закрытию для свободного использования в 2013 году), от её использования пришлось отказаться.

Вторая попытка: модификация исходного кода компилятора GCC

Второй попыткой была модификация исходного кода GCC [6] для получения в результате его работы дерева разбора программы. С одной стороны, в случае успеха такой подход гарантировал бы полную совместимость с большинством анализируемых программ. Но результатом работы фронт-энда GCC является абстрактное синтаксическое дерево,

которое, например, не содержит «лишних» с точки зрения семантики скобок и т.п. Стилевому же анализатору эта информация необходима - оптимальным входом для него является дерево разбора (parse tree). Различие между двумя типами деревьев показано на рис. 5.

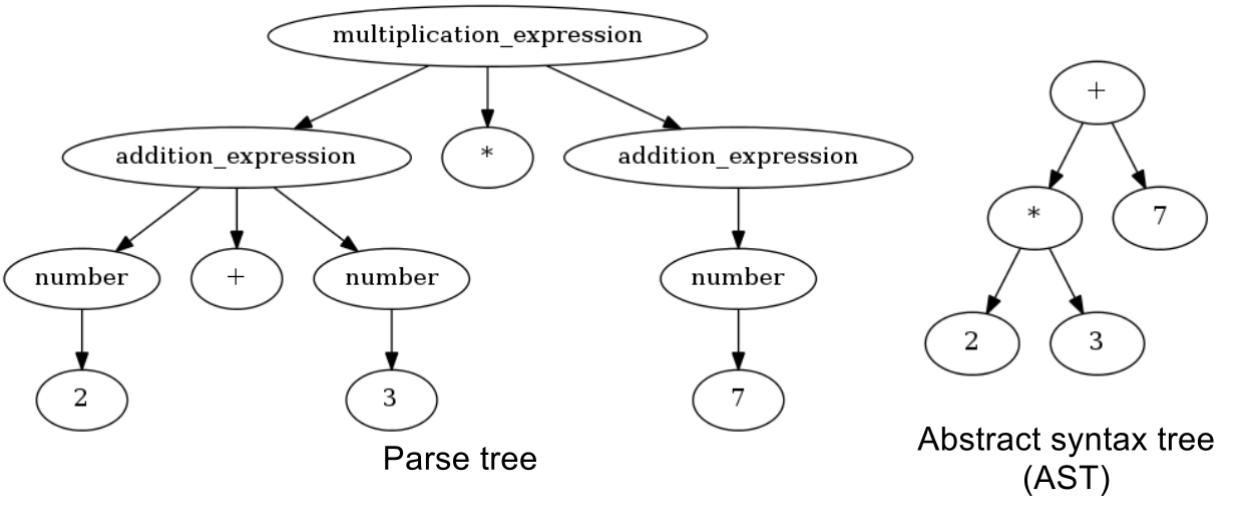

Рис. 5. Дерево разбора и абстрактное синтаксическое дерево для выражения 2*3+7

Синтаксический анализатор GCC использует метод рекурсивного спуска (recursive descent parser). В файле `parser.c` находятся около 200 функций, отвечающих за разбор каждого нетерминала (их названия начинаются с `cp_parser`). Перед каждой функцией приведён комментарий с фрагментом грамматики, описывающим нетерминал. Функции рекурсивно вызывают друг друга, возвращая разобранные вершины абстрактного синтаксического дерева (AST) в виде указателей на структуры `cp_tree`. Перед работой синтаксического анализатора производится полный разбор файла на лексемы, которые затем будут являться листьями дерева разбора. В процессе работы синтаксический анализатор поддерживает указатель на последнюю лексему, не отнесённую ни к одному правилу, т. е. ещё не присоединённую к дереву.

Был придуман способ получить именно дерево разбора, в том самом виде, в котором оно неявно строится синтаксическим анализатором в процессе работы. Для этого пригодилось почти однозначное соответствие функций фронт-энда GCC и нетерминалов грамматики C++: можно построить дерево разбора, зная, в каком порядке функции синтаксического анализатора вызывали друг друга, а также какие лексемы были уже прочитаны, т. е. добавлены к дереву, в момент этих вызовов. При этом каждая лексема (а

они образуют листья дерева разбора) будет отнесена к самой глубокой функции разбора из тех, что были в стеке вызовов на момент чтения лексемы.

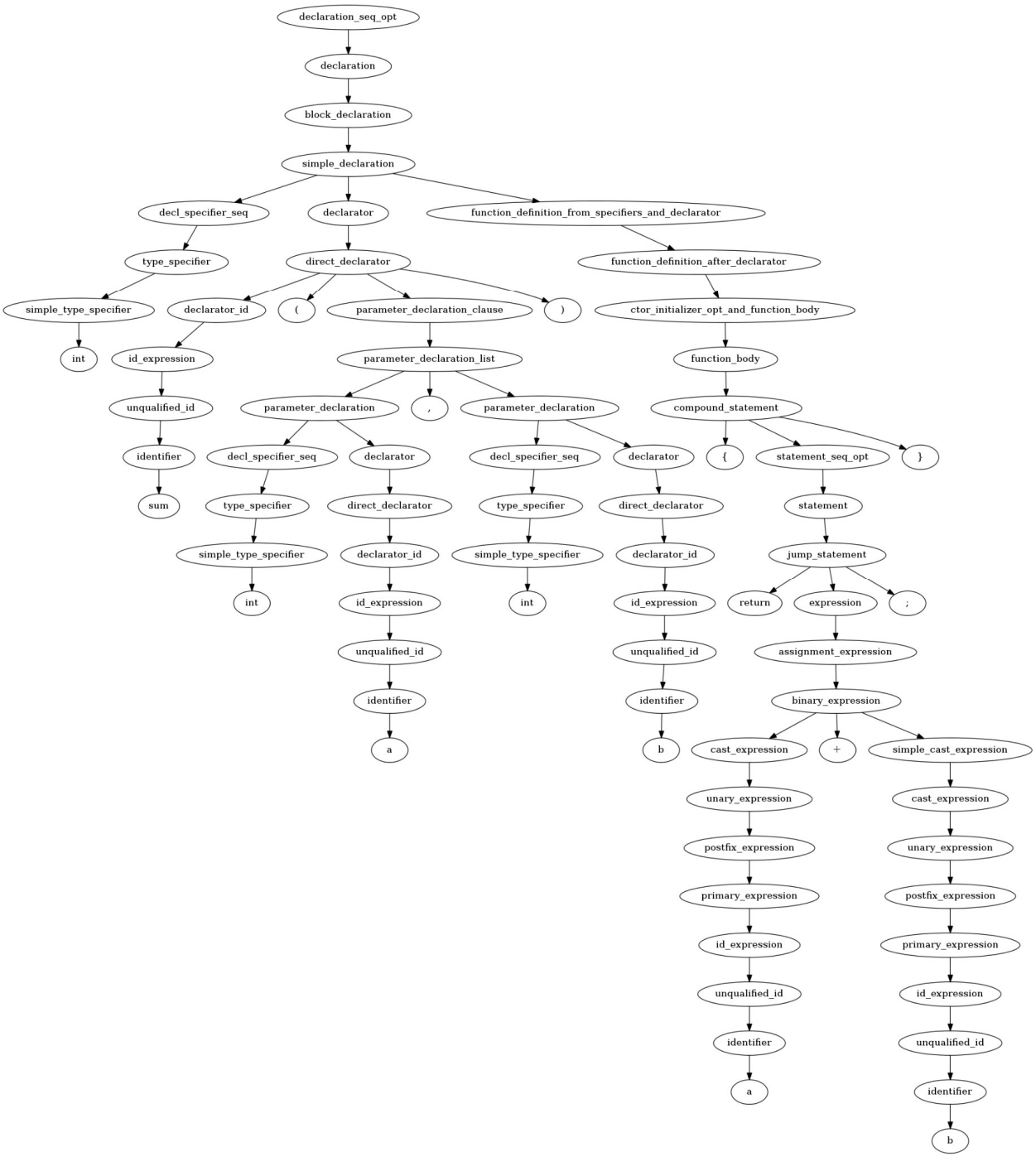

Рис. 6. Дерево разбора для программы “`int sum(int a, int b) { return a + b; }`”

Конечно же, полученное дерево не будет, строго говоря, являться деревом разбора, ведь его вершинами являются имена функций разбора, а не имена нетерминалов, а также, вообще говоря, одному нетерминалу может соответствовать несколько функций разбора.

Однако получаемое таким образом дерево пригодно для выполнения поставленных задач, так как отражает структуру программы, а небольшой доводкой его можно превратить в дерево, изоморфное «настоящему» дереву разбора. На рис. 6 приведен пример дерева разбора, как оно строится компилятором GCC. В получаемом дереве могут присутствовать специфичные для компилятора вершины (например, вершина, которую строит функция-помощник `cp_parser_consume_semicolon_at_end_of_statement`), но их число не велико.

Для получения информации о вызовах в начало каждой функции парсера GCC было втсавлено создание временного объекта, в конструкторе и деструкторе которого выполнялась запись соответствующего события. Поскольку количество функций превышает две сотни, для модификации была написана вспомогательная программа, использовавшая Boost::Wave для обнаружения реализаций функций в файле `parser.c` GCC.

Полученное дерево уже подходило для решения поставленных задач, но создавало неудобства, вызванные тем, что иногда требуется знать чуть больше, чем предоставляет чистое дерево (например, тип переменной). К тому же, предложенный подход создавал большие трудности при работе с кодом, опирающемся на препроцессор.

Третья попытка: библиотеки компилятора Clang

Оптимальным способом получения необходимой информации стало API компилятора Clang, поскольку оно позволяет получить доступ к большинству структур самого компилятора (лексической, синтаксической и семантической информации), при этом библиотека поддерживает всё, что поддерживает сам компилятор (то есть, новейший стандарт C++11).

Удобство обнаруженной библотеки позволило разработать программу, анализирующую стиль кода на языке C++ (стиль форматирования и именования переменных).

Пример работы прототипа, интеграция в Ejudge

В результате работы анализатора форматирования все проанализированные промежутки подсвечиваются зелёным и красным светом в зависимости от правильности форматирования промежутка (согласно обнаруженному автоматически стилю). Выводом анализатора является HTML-файл, содержащий подсвеченную программу и сообщения от

анализатора имён.

Программа рис. 7 написана в стиле K&R, имена переменных в ней не отделяются от скобок вызова функций. После того, как программист решает изменить стиль на Whitesmiths (рис. 8), отделять имена функций от аргументов, но не отделять оператор сравнения, анализатор способен распознать новый стиль и указать на изменившиеся ошибки. Наконец, фрагмент на рис. 9 содержит признаки смешения стилей, поэтому большое количество промежутков отмечено, как ошибочные (красным).

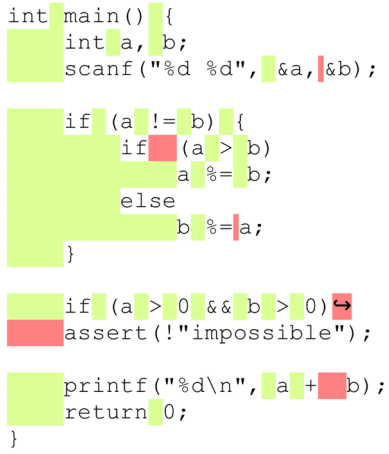

Рис. 7. Стиль «K&R»

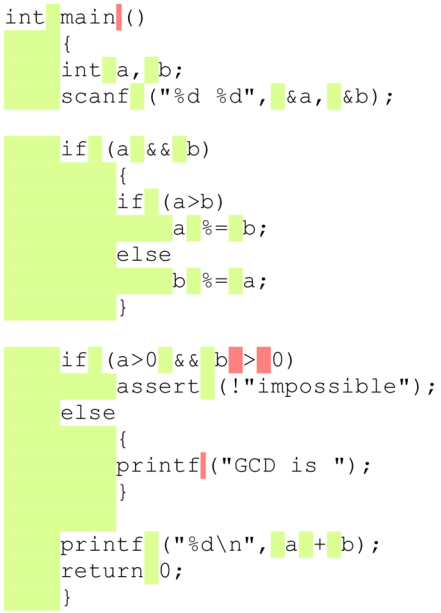

Рис. 8. Стиль «Whitesmiths»

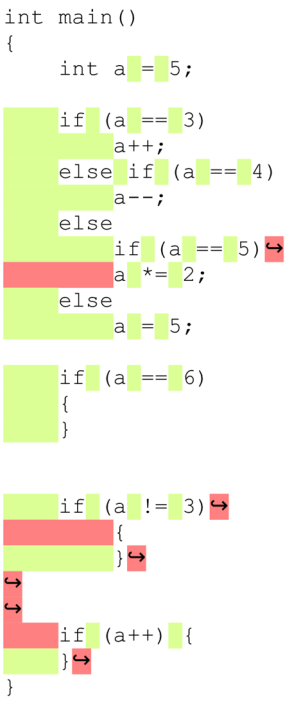

Рис. 9. Смешение стилей

В результате работы анализатора имён генерируется список переменных, имена которых с точки зрения адаптивного анализатора не соответствуют основному стилю для группы, к которой отнесена данная переменная.

Анализатор стиля можно запускать автоматически на каждой посылке, сделанной в автоматизированную систему проверки программ ejudge [13]. Для этого необходимо указать переменную ejdguge `style_checker_cmd`, после чего вывод стилевого анализатора будет доступен по ссылке, а в случае, если процент соответствия обнаруженному стилю будет ниже заданного, программа не будет допущена к проверке на тестах.

Результаты работы

В результате работы:

1. Предложена модель для анализа стиля форматирования и именования переменных в программе на произвольном языке программирования.
2. Исследован вопрос получения информации о синтаксической структуре программы на языке C++. Предложен и реализован способ получения дерева разбора с помощью компилятора GCC.
3. Реализован прототип, проверяющий стиль программы на языке C++. Реализована интеграция полученного прототипа в систему автоматизированной проверки решений Ejudge.

Список литературы

1. Herb Sutter, Andrei Alexandrescu. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison-Wesley.
2. Google Style Guide (https://code.google.com/p/google-styleguide/)
3. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман Компиляторы: принципы, технологии и инструментарий, 2 изд. — М.: Вильямс, 2008.
4. Todd L. Veldhuizen. C++ Templates are Turing Complete.
5. VivaCore, открытая библиотека анализа C/C++/C++11 кода. http://www.viva64.com/ru/vivacore-library/
6. GNU Compiler Collection, GNU C Compiler. http://gcc.gnu.org/
7. Clang LibTooling. http://clang.llvm.org/docs/Tooling.html
8. Valgrind Home. http://valgrind.org/
9. Cppcheck – a tool for static C/C++ code analysis. http://cppcheck.sourceforge.net/
10. cxxchecker – C++ source-code style check. http://gna.org/projects/cxxchecker/
11. Vera++ – programmable verification and analysis tool for C++. http://www.inspirel.com/vera/
12. KWStyle – The Source Style Checker. http://kitware.github.io/KWStyle/
13. ejudge contest management system. https://ejudge.ru/
14. Artistic Style. http://astyle.sourceforge.net/