

Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский физико-технический институт
(национальный исследовательский университет)»
Физтех-школа Прикладной Математики и Информатики
Кафедра теоретической и прикладной информатики

Направление подготовки / специальность: 03.03.01 Прикладная математика и физика
Направленность (профиль) подготовки: Математическая физика, компьютерные технологии и
математическое моделирование в экономике

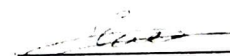
СТАТИЧЕСКИЙ АНАЛИЗ КОДА С ПОМОЩЬЮ ГРАММАТИК С КОНТЕКСТАМИ

(бакалаврская работа)

Студент:
Виноградов Илья Владимирович


(подпись студента)

Научный руководитель:
Ефанов Николай Николаевич,
канд. физ.-мат. наук


(подпись научного руководителя)

Консультант (при наличии):

(подпись консультанта)

Москва 2021

Оглавление

1.	Аннотация	4
2.	Введение	4
3.	Постановка задачи	6
3.1.	Цель работы	6
3.2.	Актуальность	6
3.3.	Формулировка задачи	7
3.4.	Этапы решения	8
	Теоретическая часть	8
	Эмпирическая часть	8
4.	Грамматики	9
4.1.	Конъюнктивные грамматики	10
	Определение	10
	Актуальные примеры	11
4.2.	Грамматики с контекстами	11
	Определение	12
	Актуальные примеры	13
5.	Грамматика	13
5.1.	Обозначения	14
5.2.	Правила вывода	15
	Общие правила	16
	Обращение к переменной	20
	Тонкости объявления переменной и функции	22
	Вызов функции	24
	Возвращаемое значение функции	27
	Использование динамической памяти	27
6.	Численный эксперимент	30
6.1.	Методика проведения эксперимента	30
6.2.	Примеры задач	32
6.3.	Анализ результатов	33
7.	Заключение	34

Список литературы	35
------------------------------------	-----------

1. Аннотация

Данная работа посвящена изучению аппарата грамматик с контекстами и его применению для оптимизации процесса компиляции. Была сформирована задача проверки эффективности данной конструкции в целях улучшения статического анализа.

В рамках поставленной задачи были выявлены определённые синтаксические паттерны языка программирования, для которых была проведена разработка грамматик. Также была проведена адаптация полученных грамматик для входного потока анализатора. Помимо этого были созданы программы, на которых проверялась корректность разработанных конструкций и была произведена их токенизация. В конечном итоге был проведён анализ полученной грамматики на реальных примерах.

На основе проведенных вычислительных экспериментов были сделаны выводы о возможности внедрения данного подхода к семантическому анализу в алгоритмы статического анализа, выявлены достоинства и недостатки данного метода. А также рассмотрены перспективы применения данных конструкций в конгруэнтной области.

2. Введение

В настоящее время статический анализ программ является стремительно развивающимся направлением компьютерной науки и системной инженерии, призванным улучшить качество и соответствие программ как функциональным, так и нефункциональным требованиям. Статический анализ имеет ряд неоспоримых преимуществ перед динамическим:

- экономит ресурсы на построение окружения исполнения и вычислительной инфраструктуры, поскольку не требует компиляции и запуска исходной программы.
- может быть произведён аппаратно-независимым способом, причём широкий класс результатов анализа будет приемлимым для большого числа современных архитектур.
- позволяет идентифицировать характерные ошибки разработки на ранних стадиях.

Тем не менее, существуют множество классов ошибок, неидентифицируемых современными статическими анализаторами по тем или иным причинам. В частности, невозможность отследить все динамические ошибки имеет фундаментальные причины: создание идеального анализатора, определяющего все неточности и не допускающего ложных срабатываний, невозможно в силу теоремы Райса [1]. Несмотря на данное ограничение, возможна разработка решений, выявляющих, в условиях жесткого следования определённым стандартам оформления программ, частные случаи ошибок, которые либо часто встречаются на практике, либо крайне критичны.

Область компиляторных технологий является открытым полем для внедрения инструментов статического анализа. Многие современные компиляторы (Clang [2], GCC [3], etc) включают в себя библиотеки для проведения статического анализа. Причины этого весьма естественны:

- Процесс компиляции исходного кода сам по себе является процессом статического анализа, на основании которого строится промежуточное представление (AST-дерево [4], etc). Из данных представлений может производиться непосредственно генерация бинарного кода для целевой платформы.
- Оптимизационные алгоритмы современного компилятора связаны с детектированием характерных конструкций на промежуточных представлениях, это тоже является статическим анализом программного обеспечения.

Если рассмотреть последовательно два вышеуказанных пункта, которые следуют из схемы преобразования программы на этапе компиляции, возникает закономерный вопрос: если необходимость строить и анализировать сложные представления программ возникает в том числе в силу синтаксической неполноты конструкций, используемых в актуальных компиляторах для построения AST-дерева, возможно ли вычислительно эффективное построение промежуточного представления, содержащего элементы программной семантики, подлежащей простому извлечению, на этапе синтаксического разбора кода? Либо же: возможно ли эффективное встраивание статического анализа в синтаксический разбор? Какие конструкции можно будет детектировать таким анализатором?

В данной работе будет предложено частичное решение подобной задачи с помощью грамматик с контекстами. Будут рассмотрены характерные примеры кодовых

конструкций, выявленных посредством инструментов и проведён анализ эффективности.

3. Постановка задачи

Но прежде, чем приступить к формулировке задачи, стоит провести краткое предметное обозрение рабочей области.

- Статический анализ [5] — это анализ программного обеспечения, который производится без реального выполнения исследуемых программ в отличие от динамического анализа. Широко используется в актуальных компиляторах, позволяет выявить ошибки программ.
- Класс грамматик с контекстами [6] — это расширение класса контекстно-свободных грамматик. Только в отличие от них (грамматик, не зависящих от окружающего контекста) данный класс позволяет осуществлять проверку контекста с помощью введённых операторов, благодаря чему способен задать более сложные синтаксические структуры, по сравнению с контекстно-свободными грамматиками, в следствие чего является более мощным классом. Формальное определение будет приведено в конгруэнтной главе.

3.1. Цель работы

Цель данной работы заключается в проверке эффективности такой конструкции как класс грамматик с контекстами в целях оптимизации процесса компиляции, а именно:

- Выявления ошибок исходного кода (в том числе динамических) на этапе статического анализа программы.
- Сохранения полиномиального времени работы алгоритма компиляции.
- Встраивания статического анализа семантики в синтаксический разбор.

3.2. Актуальность

Большинство современных компиляторов, в том числе упомянутые выше, используют контекстно-свободные грамматики для синтаксического анализа программ.

Данный класс грамматик является довольно ограниченным по выразительной способности, то есть существует множество языков, которые данная грамматика не способна задать (например, $a^n b^n c^n$). В связи с этим фактом возникают трудности с анализом ряда программных конструкций, что требует создания дополнительных промежуточных представлений и усложнения работы анализатора.

Как было подмечено прежде, грамматики с контекстами являются более выразительно мощным классом. Поэтому количество подобных невыводимых конструкций существенно меньше. Это позволяет оптимизировать анализ синтаксиса программы.

Более того, в силу увеличения мощности, данный класс грамматик обеспечивает возможность выявления ошибок, которые прежде могли быть обнаружены только на этапе динамического анализа. Данный факт значительно улучшает кооректность работы программы при запуске, в том числе и работы с памятью.

Также, асимптотика времени работы статического анализатора, использующего грамматики с контекстами сохраняет полиномиальную сложность и равняется $O(n^3)$.

На основании трёх упомянутых преимуществ можно заключить, что данная задача является актуальной и практичной в современном применении теории компиляции.

3.3. Формулировка задачи

В рамках описанной цели работы была поставлены следующие задачи:

- **Разработать грамматику с контекстами для статического анализа программы**

Данная грамматика должна порождать язык, описывающий базовые конструкции языка программирования. Также она должна позволять извлекать элементы семантики конструкций и не выводить программы с ошибками. Делать это за полиномиальное время.

- **Протестировать данную грамматику**

Выявить её преимущества и недостатки при проверке на реальных программах. Проверить теоретическую эффективность и корректность её работы на практике.

В качестве языка программирования был выбран C++ в силу своей синтаксической простоты и строгости, наглядности и актуальности по сравнению с другими языками.

3.4. Этапы решения

В целях решения поставленной задачи, было проведёно исследование и эксперимент, в которых были выполнены следующие подзадачи.

Теоретическая часть

1. Выделение паттернов программирования

Определить базовые конструкции программ, в которых может быть допущена ошибка. Языки, образованные на основании синтаксиса данных примеров должны принадлежать классу грамматик с контекстами и не принадлежать контекстно-свободным грамматикам.

2. Разработка для них грамматик

Сформулировать правила вывода синтаксиса, используя математический аппарат грамматик с контекстами.

3. Разработка грамматики для программы

Написать общие правила для вывода программы в целом. Эти правила не обязательно должны принадлежать классу грамматик с контекстами.

4. Представление грамматики в бинарной нормальной форме

Адаптация грамматики для входного потока анализатора.

Эмпирическая часть

5. Разработка корректных и ошибочных программ

Написать программы, которые бы могли подтвердить или опровергнуть правильность тестируемой грамматики. Поиск всевозможных ошибочных синтаксических конструкций.

6. Токенизация программ

Осуществить представление программы как строки токенов. Адаптация программы для входного потока анализатора.

7. Тестирование грамматики с помощью анализатора

Устранить ошибки, модифицировать грамматику. Выявить преимущества и недостатки грамматик с контекстами в статическом анализе.

Но прежде, чем перейти к аналитической и практической реализации поставленных задач, стоит сформировать понимание используемых структур и сформулировать математический аппарат применяемых конструкций.

4. Грамматики

В данной главе будут представлены формальные определения грамматик, используемых в теоретической составляющей данной работы, а также разобраны примеры, иллюстрирующие прагматичность данных конструкций.

Грамматикой [7] называется способ описания языка, то есть выделение определённого множества слов, заданных над конечным алфавитом.

В теории формальных языков грамматика — это четвёрка $G = (\Sigma, N, R, S)$, где

- Σ — алфавит определяемого языка, набор терминальных символов;
- N — конечный набор вспомогательных символов (нетерминальных символов в терминологии Хомского), используемых при определении правил вывода строк;
- R — конечный набор правил вывода строк, вида:

«левая часть» \rightarrow «правая часть», где:

«левая часть» — непустая последовательность терминалов и нетерминалов, содержащая хотя бы один нетерминал

«правая часть» — любая последовательность терминалов и нетерминалов или пустая строка

- $S \in N$ — стартовая аксиома, представляет корректно сформированные синтаксические формы языка.

Существует огромное число всевозможных классов грамматик и их классификаций, это в первую очередь связано с видом правил вывода, но в данной работе нас будут интересовать лишь конкретные грамматики, а именно: класс конъюнктивных грамматик и его расширение — класс грамматик с контекстами. Но обратимся сначала к первым.

4.1. Конъюнктивные грамматики

Широко известен факт, что контекстно-свободные грамматики, к сожалению, не являются замкнутым классом относительно пересечения. Это приводит к созданию новых, более сложных конструкций, обладающих этим свойством.

Таким классом являются конъюнктивные грамматики, для которых введён специальный оператор пересечения множеств выводимых слов «&».

Определение

Конъюнктивная грамматика [9] — это четвёрка $G_{conjunctive} = (\Sigma, N, R, S)$, где

- Σ — алфавит определяемого языка, набор терминальных символов;
- N — конесный набор вспомогательных символов (нетерминальных символов в терминологии Хомского), используемых при определении правил вывода строк;
- R — конечный набор правил вывода строк, вида:

$$A \rightarrow \alpha_1 \& \dots \& \alpha_k;$$

- $S \in N$ — стартовая аксиома, представляет корректно сформированные сентенциальные формы языка.

Здесь правила вида $A \rightarrow \alpha_1 \& \dots \& \alpha_k$ означают, что любое слово, задаваемое языком A должно принадлежать каждому из языков, задаваемых α_i . Или более формально: $L_{G_{conjunctive}}(A) = \bigcap_{i=1}^k L_{G_{conjunctive}}(\alpha_i)$.

Мощность множества языков описываемых этой грамматикой больше мощности класса контекстно-свободных языков. Доказательство этого факта основывается на том, что при $k = 1$ конъюнктивная грамматика является контекстно-свободной. Данный факт позволяет задать более сложные синтаксические конструкции, разбор одной из них будет произведён в следующем параграфе.

Актуальные примеры

Например, язык следующих слов $\{wvw \mid w \in \{a, b\}^*, v \notin \Sigma\}$, по лемме о накачке для контекстно-свободных языков, не принадлежит данному классу. Но тем не менее, возможно построить конъюнктивную грамматику которая его задаёт.

Грамматика для языка $\{wvw \mid w \in \{a, b\}^*, v \notin \Sigma\}$ [9]

$$S \rightarrow V \& D$$

$$A \rightarrow XAX \mid vEa$$

$$B \rightarrow XBx \mid vEb$$

$$X \rightarrow a \mid b$$

$$V \rightarrow XVX \mid v$$

$$D \rightarrow aA \& aD \mid bB \& bD \mid vE$$

$$E \rightarrow XE \mid \varepsilon$$

Нетерминал V задаёт строки, состоящие из двух частей равной длины, разделённые центральным маркером. Более формально: $L(C) = \{w_1vw_2 \mid w_1, w_2 \in \{a, b\}^*, |w_1| = |w_2|\}$ Нетерминал D задаёт язык $L(D) = \{svys \mid s, y \in \{a, b\}^*\}$. Пересечение данных языков образует слова, из нужного языка. $s = \varepsilon$.

Данный пример будет иметь практическое применение при построении грамматики программы.

4.2. Грамматики с контекстами

Теперь будет рассмотрен самый актуальный для данной работы класс грамматик. Определение грамматик с контекстами требует подробное определение понятия контекст, что и будет представлено далее.

Пусть имеется строка w , полученная конкатенацией трёх подстрок x, y, z , то есть $w = xyz$, тогда

- левым контекстом подстроки y называется подстрока x
- левым расширенным контекстом подстроки y называется подстрока xy
- правым контекстом подстроки y называется подстрока z
- правым расширенным контекстом подстроки y называется подстрока yz

Как было сказано прежде, грамматики с контекстами позволяют осуществлять проверку контекста. Это свойство заложено в самой структуре грамматики и осуществляется с помощью операторов проверки ($<$, \leq , $>$, \geq). Чтобы лучше это понять, следует рассмотреть следующий пример:

$$S \rightarrow CA \mid A \mid CCA$$

$$A \rightarrow B \ \& \ < \ C$$

$$B \rightarrow b$$

$$C \rightarrow c$$

Если бы в этой грамматике не было конструкции $\& <$, то она задавала бы 3 слова-строки «cb, b, ccb». Но во втором правиле введён оператор проверки левого контекста, который позволяет воспользоваться этим правилом только тогда, когда левый контекст (слева от нетерминала A в сентенциальной форме) является языком C . То есть, слово $w = cb$ может быть выведено с помощью этой грамматики, но ни слово $w' = b$, ни слово $w'' = ccb$ не может быть выведено из этой грамматики с контекстами.

Класс грамматик с контекстами делится на грамматики с односторонним контекстом, тех которые включают проверку либо левого либо правого расширенного и/или не расширенного контекста, и грамматики с двусторонним контекстом. Выразительная способность последних больше, поэтому будем рассматривать их более подробно.

Определение

Грамматика с двусторонним контекстом — это четвёрка $G_{context} = (\Sigma, N, R, S)$, где

- Σ — алфавит определяемого языка;
- N — конесный набор вспомогательных символов (нетерминальных символов в терминологии Хомского), используемых при определении правил вывода строк;
- R — конечный набор правил вывода строк, вида:

$$A \rightarrow \alpha_1 \ \& \ \dots \ \& \ \alpha_k \ \& \ < \ \beta_1 \ \& \ \dots \ \& \ \beta_m \ \& \ \leq \ \gamma_1 \ \& \ \dots \ \& \ \gamma_n \ \&$$

$$\ \& \ > \ \delta_1 \ \& \ \dots \ \& \ \delta_{m'} \ \& \ \geq \ \kappa_1 \ \& \ \dots \ \& \ \kappa_{n'},$$

где $A \in N, k > 1, m, n, m', n' \geq 0$ и $\alpha_i, \beta_i, \gamma_i, \delta_i, \kappa_i \in (\Sigma \cap N)^*$;

- $S \in N$ — стартовая аксиома, представляет корректно сформированные синтаксические формы языка.

Актуальные примеры

Рассмотрим язык $L = a^n b^n c^n$. Данный язык не является контекстно-свободным в силу леммы о накачке. Докажем это, проверив выполнение её отрицания.

$$\forall n \exists w = a^n b^n c^n, w \in L, |w| > n : \forall xuyvz = w, |uyv| < n, |uv| > 0 \exists i = 0, i \in \{\mathbb{N} \cup 0\} : xu^i y v^i z = xyz \notin L.$$

Разобьём слово на три части, каждая из которых состоит из одного и того же терминального символа. Действительно в силу того, что $|uyv| < n$ данный фрагмент разбиения не захватит одну из трёх частей слова. Следовательно, когда произойдёт уничтожение u, v , хотя бы в одной части количество терминалов уменьшится, и хотя бы в одной останется неизменным. Что сформирует слово, не принадлежащее данному языку.

Однако этот язык $\{a^n b^n c^n \mid n > 0\}$ можно задать с помощью грамматик с контекстами.

$$S \rightarrow aSc \mid B \& \leq X$$

$$B \rightarrow bB \mid b$$

$$X \rightarrow aXb \mid ab$$

Нетерминал X задаёт все строки вида $a^n b^n$, где $n > 0$. Без контекстного оператора, символ S будет задавать все строки вида $a^n b^m c^n$, где также $n > 0$. Однако правило $S \rightarrow B \& \leq X$ гарантирует, что префикс строки будет являться языком $a^k b^k$ для какого-либо $k > 0$. Именно наличие проверки контекста (левого в данном случае) позволяет сформировать грамматику для этого языка.

Проведённый анализ вышеперечисленных классов грамматик и примеров с ними позволяет рассмотреть грамматику, разработанную для анализа программ: построения дерева вывода и выявление ошибок.

5. Грамматика

Данная грамматика с контекстами разработана для статического анализатора. Она должна выполнять ряд задач, а именно: порождать правильный синтаксис языка программирования C++, не выводить программы с некорректным синтаксисом,

а также распознавать некоторые динамические ошибки программ. Конкретизация ошибок и грамматики и будет представлена в этой главе.

5.1. Обозначения

Прежде, чем ввести правила вывода грамматики следует остановиться на принятых обозначениях.

Данная грамматика включает всебя совокупность правил, состоящих из нетерминальных и терминальных символов. В целях упрощения интуитивного понимания правил вывода, эти символы данной грамматики — это слово, написанное без разделительных символов (пробелов), возможно, состоящее из конкатенации двух или нескольких слов английского языка.

Любое слитно написанное слово, начинающееся с большой буквы, является нетерминальным символом. Слово, не выделенное жирным, означает, что грамматика для данного нетерминала не является сложной и требует нескольких правил. В целях упрощения структурного понимания грамматики, разбор многих таких конструкций будет опущен.

Также следует заметить, что любой символ, начинающийся с прописной английской буквы «t» означает токенизированное представление описываемого объекта с возможными отступами и пунктуационными символами, в случаях, где это необходимо. Так например, «tInt» означает токен типа переменных: "int" , «tDeletion» — "delete" , tSemicolon — ";" , «tEqual» — "=", «tOpenBrace» — "{", «tCloseParenthesis» — ")" и т. д.

В данной грамматике существует нетерминальный символ наличия пробела и записывается он так: " "Введём некоторые базовые правила вывода для нижеиспользуемых нетерминалов.

Нетерминал **Letter** задаёт алфавит названия переменных и функций.

Letter \rightarrow "a" ... "z" | "A" ... "Z" | "_" ...

Letters \rightarrow **Letter** **Letters** | ε

Digit \rightarrow "0" ... "9"

Digits \rightarrow **Digit** **Digits** | ε

LetterOrDigit \rightarrow **Letter** | **Digit**

LettersOrDigits \rightarrow **LetterOrDigit** **LettersOrDigits** | ε

Также стоит вывести понятие некоторых винарных и унарных операторов над различными типами.

$\text{IntegerUnaryOperator} \rightarrow "-"$

$\text{IntegerBinaryOperator} \rightarrow "+" \mid "-" \mid "*" \mid "/" \mid "\%"$

$\text{FloatBinaryOperator} \rightarrow "+" \mid "-" \mid "*" \mid "/"$

$\text{BooleanBinaryOperator} \rightarrow "!=" \mid "<" \mid ">" \mid "<=" \mid ">=" \mid "-"$

$\text{BooleanUnaryOperator} \rightarrow "!"$

Также хотелось бы подчеркнуть семантическую особенность следующих нетерминалов:

- **Identifier** отвечает за объявление первой переменной в некотором участке видимости программы. Выводит корректное название переменной.
- **ValidIdentifier** отвечает за обращение к переменной в некотором участке видимости программы. Выводит корректное название переменной.
- **InvalidIdentifier** отвечает за объявление всех переменных, кроме первой в некотором участке видимости программы. Выводит корректное название переменной.

5.2. Правила вывода

Разработка грамматики в первую очередь характеризуется разработкой правил вывода, поскольку терминальные символы программы известны — токенизированное представление фундаментальных конструкций языка C++ и английский алфавит. Стартовая аксиома данной грамматики — нетерминал **Programm**. Перечисление всего множества нетерминальных символов грамматики столь же бессмысленно, сколько и долго, однако некоторые из них были представлены выше, в целях упрощения понимания грамматики.

Правила вывода синтаксиса языка программирования не могут быть короткими и простыми, поэтому в данной главе будет произведён их подробный анализ и разработка

Общие правила

Программа рассматривается как совокупность функций , в том числе и главной (MainFunction)

Program \rightarrow **Functions** **MainFunction** **Functions**

Functions \rightarrow **Function** **Functions** $\mid \varepsilon$

Каждая функция состоит из заголовка, включающего в себя тип возвращаемого значения, названия функции и её аргументов(или их отсутствие) и основной части (тела функции).

Function \rightarrow

FunctionHeader

tOpenBrace **StatementsOrNot** tCloseBrace

MainFunction \rightarrow

MainFunctionHeader

tOpenBrace **StatementsOrNot** tCloseBrace

MainFunctionHeader \rightarrow

tInt \sqcup tMain tOpenParenthesis

tArgcArgvOrNot tCloseParenthesis

FunctionHeader \rightarrow **FunctionType** **FunctionSignature**

FunctionSignature \rightarrow

FunctionIdentifier tOpenParenthesis

ParametersOrNot tCloseParenthesis

Нетерминал **FunctionIdentifier** гарантирует уникальность имени функции, его описание будет предоставлено чуть позже

ParametersOrNot \rightarrow **Parameters** $\mid \varepsilon$

Parameters \rightarrow **Parameter** tComma \sqcup **Parameters** \mid **Parameter**

Parameter \rightarrow **VariableType** **Identifier**

Нетерминал **Identifier** определяет первое объявлений переменной в некоторой зоне видимости, будет описан позже.

Рассмотрим описание типов функций и переменных, предварительно заметив, что тип может быть как для обычной переменной, так и для указателя. В данной статье во избежание громоздкости грамматики будут рассмотрены лишь 4 типа: void, bool, int, float. Тип переменной в отличие от типа функции не может быть void, а

лишь `bool`, `int` и `float`.

FunctionType \rightarrow **VariableType** | **VoidType**

VariableType \rightarrow **Type** $_$ **Stars**

Type \rightarrow `tInt` | `tBool` | `tFloat`

Stars \rightarrow `tStar Stars` | ε

VoidType \rightarrow `tVoid` $_$ **Stars**

Identifier, как было объявлено ранее, определяет название переменной в аргументах функции. А значит, грамматика должна отражать требования к её названию: не начинаться с цифры, состоять из допустимых символов, не являться словом языка программирования (например, `int`) и т. д. Такие требования легко задаются с помощью конъюнктивных грамматик.

Identifier \rightarrow **Letter** **LettersOrDigits** & **NotKeyword**

NotKeyword \rightarrow **NotInt** & ... & **NotFalse**

Здесь подробное описание нетерминала **NotKeyword** опущено в силу его громоздкости. Стоит заметить, что язык, состоящий из одного слова (например, `int`) является регулярным, а значит и его дополнение тоже. Поэтому данная конструкция реализуема и принадлежит классу конъюнктивных грамматик.

Описание нетерминалов **Letter**, **LettersOrDigits** было предоставлена выше.

Теперь обратимся к нетерминалу **StatementsOrNot**. Тело каждой функции должно либо быть пустым либо состоять из набора выражений, последнее из которых должно грамотно осуществлять вызов `return`.

StatementsOrNot \rightarrow **Statements** | ε

Statements \rightarrow

Statement **Statements** |

ProperlyReturningStatement

Следует подчеркнуть зачимость нижепредставленной конструкции, поскольку она является центральной частью всей грамматики и утверждает, что тело функции может состоять только из набора следующих конструкций.

Statement \rightarrow

VariableDeclaration |

ExpressionStatement |

VoidFunctionCall |

AssignmentStatement |
IfStatement |
WhileStatement |
tOpenBrace **Statements** tCloseBrace

ExpressionStatement всего лишь содержит выражение, а вот **AssignmentStatement** уже содержит оператор присваивания.

VariableDeclaration →
VariableType **InvalidIdentifiers**
InvalidIdentifier tSemicolon

InvalidIdentifiers →
InvalidIdentifier tComma **InvalidIdentifiers** | ε

Здесь предоставлено описание переменной. Нетерминал **InvalidIdentifier** гарантирует уникальность названия. Его определение будет предоставлено позже.

ExpressionStatement → **Expression** tSemicolon
Expression → **FloatExpression** | **BooleanExpression** |
IntegerExpression
IntegerExpression →
IntegerConstant |
ValidIntegerIdentifier |
tOpenParenthesis **IntegerExpression**
tCloseParenthesis |
IntegerExpression **IntegerBinaryOperation**
IntegerExpression |
IntegerUnaryOperator **IntegerExpression** |
IntegerFunctionCall

BooleanExpression →
tTrue | tFalse |
ValidBooleanIdentifier |
tOpenParenthesis **BooleanExpression**
tCloseParenthesis |
BooleanExpression **BooleanBinaryOperation**
BooleanExpression |

IntegerExpression BooleanBinaryOperation
IntegerExpression |
 BooleanUnaryOperation **BooleanExpression** |
BooleanFunctionCall

FloatExpression \rightarrow
FloatConstant | **IntegerConstant**
ValidFloatIdentifier |
 tOpenParenthesis **FloatExpression**
 tCloseParenthesis |
FloatExpression FloatBinaryOperation
FloatExpression |
 IntegerUnaryOperator **FloatExpression** |
FloatFunctionCall

Переменная любого типа, используемая в выражениях, должна быть объявлена и более того, само выражение должно находиться в зоне видимости переменной, что и гарантируют нетерминалы, **ValidFloatIdentifier**, **ValidIntegerIdentifier** и **ValidBooleanIdentifier**. Их подробное описание будет предоставлено в следующем разделе

Определив нетерминалы **FloatExpression**, **BooleanExpression** и **IntegerExpression** появилась возможность определить выражения присваивания, условные конструкции и циклы.

AssignmentStatement \rightarrow
ValidIntegerIdentifier tEqual
IntegerExpression tSemicolon |
ValidFloatIdentifier tEqual
FloatExpression tSemicolon |
ValidBooleanIdentifier tEqual
BooleanExpression tSemicolon |

Определим условные конструкции и цикл while

IfStatement \rightarrow tIf tOpenParenthesis
BooleanExpression tCloseParenthesis
 tOpenBrace **Statements** tCloseBrace

ElseStatement

$$\text{ElseStatement} \rightarrow \text{tElse tOpenBrace}$$

$$\text{Statements tCloseBrace} \mid \varepsilon$$

$$\text{WhileStatement} \rightarrow \text{tWhile tOpenParenthesis}$$

$$\text{BooleanExpression tCloseParenthesis}$$

$$\text{tOpenBrace Statements tCloseBrace}$$

Теперь, когда определены базовые правила вывода программ в целом, можно приступить к рассмотрению ошибок семантического характера.

Обращение к переменной

При обращении к переменной программы следует помнить о корректности этой операции, например, стоит проверить, была ли она объявлена в данной области видимости. Также следует помнить, что переменная может быть объявлена, как в параметрах функции, так и в её теле.

```
int foo(int x)
{
  int y;
  ...
  y = x + y; \ exists declaration of x?, exists declaration of y?
  ...
}
```

Это требование можно частично формализовать как строку языка вида www , где w — имя переменной v — весь код между ними. В предыдущей главе было рассмотрено, что такой язык не является контекстно-свободным, а также была приведена грамматика для него. Поэтому, начиная с этого раздела, мы обратимся к конъюнктивным грамматикам и грамматикам с контекстами.

Однако разобранный пример никак не учитывает область видимости конструкции, где было произведено обращение к переменной. Нетерминалы **ValidFloatIdentifier**, **ValidIntegerIdentifier** и **ValidBooleanIdentifier**, гарантирующие выполнение двух этих условий и будут описаны в данном разделе.

$$\text{ValidIntegerIdentifier} \rightarrow \text{Identifier}$$

$\& \leq \mathbf{Functions} \mathbf{FunctionType} \text{ Anything-except-function-header}$

$\mathbf{tInt} _ \mathbf{Identifiers} \mathbf{V}$

$\mathbf{ValidFloatIdentifier} \rightarrow \mathbf{Identifier}$

$\& \leq \mathbf{Functions} \mathbf{FunctionType} \text{ Anything-except-function-header}$

$\mathbf{tFloat} _ \mathbf{Identifiers} \mathbf{V}$

$\mathbf{ValidBooleanIdentifier} \rightarrow \mathbf{Identifier}$

$\& \leq \mathbf{Functions} \mathbf{FunctionType} \text{ Anything-except-function-header}$

$\mathbf{tBool} _ \mathbf{Identifiers} \mathbf{V}$

$\mathbf{Identifiers} \rightarrow \mathbf{Identifier} \mathbf{tComma} _ \mathbf{Identifiers} \mid \varepsilon$

Нетерминал `Anything-except-function-header` задаёт регулярный язык, в котором отсутствуют заголовки функций. В целях наглядности работы и экономии пространства опущена.

Функционал конструкции, приведённой после оператора \leq можно описать как поиск слова *www* за вывод которого будет отвечать нетерминал **V** (variable). Для этого она сначала "пропускает" описание сторонних функций, после этого "ищет" ключевое слово нужного типа либо в аргументах функции, либо в её теле, затем "находит" объявление переменной. И в конце концов определяет, соответствие между объявленной и используемой переменной.

$\mathbf{V} \rightarrow \mathbf{Vlen} \& \mathbf{Viterate}$

$\mathbf{Vlen} \rightarrow$

$\mathbf{LetterOrDigit} \mathbf{Vlen} \mathbf{LetterOrDigit} \mid$

$\mathbf{LetterOrDigit} \mathbf{Vmid} \mathbf{LetterOrDigit}$

$\mathbf{Va} \rightarrow$

$\mathbf{LetterOrDigit} \mathbf{Va} \mathbf{LetterOrDigit} \mid$

$\text{"a"} \mathbf{LettersOrDigits} \mathbf{Vmid}$

\vdots

$\mathbf{V9} \rightarrow$

$\mathbf{LetterOrDigit} \mathbf{V9} \mathbf{LetterOrDigit} \mid$

$\text{"9"} \mathbf{LettersOrDigits} \mathbf{Vmid}$

$\mathbf{Viterate} \rightarrow$

$\mathbf{Va} \text{"a"} \& \mathbf{Viterate} \text{"a"} \mid \dots \mid \mathbf{Vz} \text{"z"} \& \mathbf{Viterate} \text{"z"} \mid$

$\mathbf{V0} \text{"0"} \& \mathbf{Viterate} \text{"0"} \mid \dots \mid \mathbf{V9} \text{"9"} \& \mathbf{Viterate} \text{"9"} \mid$

LettersOrDigits Vmid

Vmid \rightarrow **Vmiddle** & **VmidDyck**

Vmiddle \rightarrow

Any-punctuator Any-string Any-punctuator |
 _ Any-string Any-punctuator | Any-punctuator Any-string _ |
 _ Any-string _ | Any-punctuator | _

Нетерминал **Vmiddle** определяет границы переменных: перед ними должна быть либо пунктуация, либо пробел.

Проверку на зону видимости осуществляет **VmidDyck** — задаёт язык таких слов, скобочная последовательность в которых является правильной, за исключением некоторых открывающихся скобок.

VmidDyck \rightarrow

Anything-except-braces |
VmidDyck Anything-except-braces **VmidDyck** |
 tOpenBrace Anything-except-braces tCloseBrace |
 tOpenBrace Anything-except-braces **VmidDyck**
 Anything-except-braces tCloseBrace |
 tOpenBrace

Подобная конструкция гарантирует область видимости.

Тонкости объявление переменной и функции

Не только обращение к переменной требует определённых проверок, но и её объявление может содержать ряд семантических ошибок. Например, во время объявления имени непервой переменной в текущей области видимости мы должны гарантировать его уникальность. Также при отсутствии дальнейшего обращения к переменной, следует указать на её бессмысленность. Данный недочёт будет рассмотрен, как ошибка в программе.

```
int main()
{
  int num;
  ...
  int num2; \\is name of variable unique in scope of visiability?
```

```
...    \\ will be num2 used later in it's scope?
num2 += num;
}
```

Нетерминал **InvalidIdentifier** отвечает за проверку выполнения данных требований.

InvalidIdentifier \rightarrow **Identifier**

$\& \leq$ **Functions** **FunctionType** Anything-except-function-header

VariableType **Identifiers** **D**

$\& \geq$ **V** tEqual Anything

Нетерминал **V** как раз таки осуществляет проверку дальнейшего обращения к переменной, а именно поиск синтаксической конструкции wvw в последующем участке кода, безусловно беря во внимание зону видимости.

Нетерминал **D** здесь определяет конъюнктивный язык, строк вида $w_1x_1 \dots w_lx_lw_{l+1}$, в которых каждый x_i не начинается и не кончается ни буквой, ни цифрой и

- либо все w_i — различны
- либо некоторые из них одинаковые, но тогда по крайней мере один x_i содержит больше «{»-скобок, чем «}»-скобок

Несмотря на то, что оператор отрицания не определён для грамматик с контекстами, он может помочь в интуитивном понимании вышеописанной конструкции. Данная конструкция не является правилами вывода и лишь упрощает понимание предыдущих правил.

InvalidIdentifier \rightarrow **Identifier** $\& \neg$ **ValidIdentifier**

ValidIdentifier \rightarrow **ValidBooleanIdentifier** |

ValidFloatIdentifier | **ValidIntegerIdentifier**

Теперь, когда введён нетерминал для корректного объявления переменной, аналогичным способом определяется нетерминал **FunctionIdentifier**, который отвечает за корректное объявление имени функции и наличие дальнейшего обращения к ней.

```
int foo(...)
{...}
```

```
int foo2(...) \\is name of the function unique?
{...}    \\ will be foo2 used later in programm?
```

```
int main()
{
...
x = foo2(...);
...
}
```

FunctionIdentifier \rightarrow

Identifier $\& \leq$ **Functions** **FunctionType** **D**

$\& \geq$ **F** Anything

Здесь осуществляется проверка наличия функции с помощью нетерминала **F**, ровно таким же способом, как и для нетерминалов **ValidIntegerIdentifier** и прочих. Поэтому описание нетерминала **F** полностью аналогично нетерминалу **V**, за исключением того факта, что область видимости для вызываемой функции — вся программа. Поэтому вместо правила **Vmid** \rightarrow **Vmiddle** $\&$ **VmidDyck** будет правило **Fmid** \rightarrow **Vmiddle**.

Вызов функции

Однако, в отличие от обращения к переменной, вызов функции требует множество дополнительных проверок. Так помимо проверки об объявлении, данные правила также должны осуществлять проверку, согласования между количеством переданных актуальных аргументов и количеством параметров в её описании. А также соответствие их типизаций.

```
void foo(int x, bool y)
{...}
```

```
int main()
{
int x = 5;
```



```

bool y = true;
...
foo(x, y); \\ was declared earlier in this programm
... \\is amount of actual arguments equal to
...      amount of function parameters?
...      \\ is agreement of their types correct?
}

```

ValidFunctionNameAndArguments \rightarrow

ValidFunctionName ValidFunctionArguments

ValidFunctionName \rightarrow **Identifier**

& \leq Functions FunctionType F

Здесь осуществляется проверка наличия функции, ровно таким же способом, как и для нетерминалов **ValidIntegerIdentifier** и прочих. Поэтому описание нетерминала **F** полностью аналогично нетерминалу **V**, за исключением того факта, что область видимости для вызываемой функции — вся программа. Поэтому вместо правила **Vmid** \rightarrow **Vmiddle** **& VmidDyck** будет правило **Fmid** \rightarrow **Vmiddle**.

ValidFunctionArguments \rightarrow

tOpenParenthesis ExpressionsOrNot tCloseParenthesis

& tOpenParenthesis A tCloseParenthesis

ExpressionsOrNot \rightarrow **Expressions** | ε

Expressions \rightarrow **Expressions tComma Expression** | **Expression**

Нетерминал **A** гарантирует корректность количества аргументов функции и их типов. Его описание будет предоставлено ниже.

IntegerFunctionCall \rightarrow

Identifier tOpenParenthesis

Expressions tCloseParenthesis

& \leq Functions tInt $_$ ValidFunctionNameAndArguments

FloatFunctionCall \rightarrow

Identifier tOpenParenthesis

Expressions tCloseParenthesis

& \leq Functions tFloat $_$ ValidFunctionNameAndArguments

BooleanFunctionCall \rightarrow

Identifier **tOpenParenthesis**

Expressions **tCloseParenthesis**

$\& \leq \mathbf{Functions}$ **tBool** $_ \mathbf{ValidFunctionNameAndArguments}$

VoidFunctionCall \rightarrow

Identifier **tOpenParenthesis**

Expressions **tCloseParenthesis**

$\& \leq \mathbf{Functions}$ **tVoid** $_ \mathbf{ValidFunctionNameAndArguments}$

Нетерминал **A** (Accordance) отвечает за проверку корректности типа переданных значений, а также их количественное соответствие между актуальными аргументами вызываемой функции и параметрами описанной.

A $\rightarrow \mathbf{A}len \ \& \ \mathbf{A}iterate$

A*len* \rightarrow

VariableType **Identifier** **A***len* **Expression** $|$

tComma $_ \mathbf{A}len$ **tComma** $_ | \mathbf{A}mid$

IntVarOrExpr $\rightarrow \mathbf{tInt} _ \mathbf{Identifier} | \mathbf{IntegerExpression}$

FloatVarOrExpr $\rightarrow \mathbf{tFloat} _ \mathbf{Identifier} | \mathbf{FloatExpression}$

BoolVarOrExpr $\rightarrow \mathbf{tBool} _ \mathbf{Identifier} | \mathbf{BooleanExpression}$

A*x* $\rightarrow \mathbf{IntVarOrExpr} | \mathbf{BoolVarOrExpr} |$

FloatVarOrExpr $| \mathbf{tComma} _$

A*y* $\rightarrow \mathbf{A}x \ \mathbf{A}y | \varepsilon$

A*int* $\rightarrow \mathbf{A}x \ \mathbf{A}int \ \mathbf{A}x | \mathbf{IntVarOrExpr} \ \mathbf{A}y \ \mathbf{A}mid$

A*float* $\rightarrow \mathbf{A}x \ \mathbf{A}float \ \mathbf{A}x | \mathbf{FloatVarOrExpr} \ \mathbf{A}y \ \mathbf{A}mid$

A*bool* $\rightarrow \mathbf{A}x \ \mathbf{A}bool \ \mathbf{A}x | \mathbf{BoolVarOrExpr} \ \mathbf{A}y \ \mathbf{A}mid$

A*comma* $\rightarrow \mathbf{A}x \ \mathbf{A}comma \ \mathbf{A}x | \mathbf{tComma} _ \mathbf{A}y \ \mathbf{A}mid$

A*iterate* \rightarrow

A*int* **IntVarOrExpr** $\&$

A*iterate* **IntVarOrExpr** $|$

A*bool* **BoolVarOrExpr** $\&$

A*iterate* **BoolVarOrExpr** $|$

A*comma* **tComma** $_ \&$

A*iterate* **tComma** $_ |$

Ау Amid

Amid \rightarrow tCloseParenthesis Anything tOpenParenthesis

Возвращаемое значение функции

При описании тела функции также должно выполняться условие, что тип значения, которое она возвращает, должен соответствовать заявленному. Для этого при выполнении операции *return* (или её отсутствии в void-функциях) с помощью оператора левого контекста будем осуществлять проверку типа данной функции, заявленного в заголовке.

```
int foo(...)
{
  int x;
  ...
  return x; \\is x an integer expression?
}
```

За проверку этого требования отвечает нетерминал **ProperlyReturningStatement**

ProperlyReturningStatement \rightarrow

```
tReturn IntegerExpression tSemicolon
& < Functions tInt
    Anything-except-function-header |
tReturn FloatExpression tSemicolon
& < Functions tFloat
    Anything-except-function-header
tReturn BooleanExpression tSemicolon
& < Functions tBool Anything-except-function-header
```

Использование динамической памяти

При написании программ на языке C++ часто возникает задача обращения к динамической памяти через указатели. В данной работе была рассмотрена грамматика для объявления указателей с помощью нетерминала **VariableDeclaration**, но

его синтаксис пока что не гарантирует корректной работы с динамическими переменными. Исправим это недоразумение в этом разделе.

Statement → **PointerDeclaration** | **PointerDeletion**

Объявление динамической переменной требует объявление указателя с помощью вызова *new*, а также его удаление с помощью вызова *delete()*. Эти вызовы являются равноправными выражениями поэтому их задаёт нетерминал **Statement**, использованный прежде **Вставить ссылку!**. Всевозможные проверки объявлений и удалений будут учтены позже

PointerDeclaration →

IntegerPointerConst | **IntegerPointer** |
FloatPointerConst | **FloatPointer** |
BooleanPointerConst | **BooleanPointer** |
NullPointer

Такое разделение по наличию константы связано с мгновенной инициализацией переменной или массива и последующей во избежание синтаксически ошибок. **...PointerConst** отвечает за мгновенную инициализацию константой корректного типа, **...Pointer** — за дальнейшую

IntegerPointerConst →

tInt ⊆ tStar **IntCheckDeletion**
ParenthesisIntegerConstant

IntegerPointer →

tInt ⊆ tStar **IntCheckDeletionAndStars**
ArrayOrSemicolon

FloatPointerConst →

tFloat ⊆ tStar **FloatCheckDeletion**
ParenthesisFloatConstant

FloatPointer →

tFloat ⊆ tStar **FloatCheckDeletionAndStars**
ArrayOrSemicolon

BooleanPointerConst →

tBool ⊆ tStar **BoolCheckDeletion**
ParenthesisBooleanConstant

BooleanPointer \rightarrow

$t\text{Bool} \sqcup t\text{Star}$ **BoolCheckDeletionAndStars**

ArrayOfSemicolon

ArrayOfSemicolon осуществляет что-то, **Parenthesis ... Constant** -тоже. Далее среди нетерминалов **... CheckDeletion** и **... CheckDeletionAndStars** в целях упрощения понимания конструкции будут рассмотрены только те, которые отвечают за тип *int*. Остальные восстанавливаются аналогичным образом.

ParenthesisIntegerConstant \rightarrow

$t\text{OpenParenthesis}$ **IntegerExpression** $t\text{CloseParenthesis}$

ParenthesisFloatConstant \rightarrow

$t\text{OpenParenthesis}$ **FloatExpression** $t\text{CloseParenthesis}$

ParenthesisIntegerConstant \rightarrow

$t\text{OpenParenthesis}$ **BooleanExpression** $t\text{CloseParenthesis}$

ArrayOfSemicolon $\rightarrow t\text{Semicolon} \mid$

$t\text{OpenBracket}$ **IntegerExpression** $t\text{CloseBracket}$

Стоит заметить, что здесь нет текста!

```
int main()
{
int ***x = new int **[10]; \\ will be the dynamic memory cleared?
...           \\ will be an agreement of stars correct
delete **x;    \\ between declaration, inicialization and deletion
}
```

IntCheckDeletion $\rightarrow \text{IntVariableName}$

$\& \geq \text{VDel}$ Anything

IntCheckDeletionAndStars $\rightarrow \text{IntPointerStars}$

$\& \geq \text{VDel}$ Anything

IntPointerStars $\rightarrow t\text{Star}$ **IntPointerStars** $t\text{Star} \mid$

$t\text{Star}$ **IntVariableName** $t\text{Star}$

Нетерминал **VDel** задаёт слова вида $w \ v \ t\text{Deletion} \ w$ с учётом зоны видимости. Что делает его очень схожим с нетерминалом **V** за исключением нескольких правил. Этот нетерминал осуществляет проверку очистки динамической памяти при её выделении.

```

int main()
{
int *x = new int; \\ will be the dynamic memory used?
...
*x = 10;
...
}

```

IntVariableName \rightarrow **InvalidIdentifier** tEqual tNew \sqcup tInt
 $\& \geq V$ Anything-except-deletion **PointerDeletion** Anything
PointerDeletion \rightarrow tDelete
tOpenParenthesis **ValidPointer** tCloseParenthesis
ValidPointer \rightarrow Stars **Identifier**
NullPointer \rightarrow
VariableType tStar **NullptrInicialization**
NullptrInicialization \rightarrow
InvalidIdentifier tEqual tNullptr
 $\& \geq VDel$ Anything

6. Численный эксперимент

Как известно, истинность любой теоретической гипотезы должна подкрепляться практическим подтверждением. Поэтому после формирования правила вывода синтаксиса программ, было проведёно их тестирование на реальных примерах.

6.1. Методика проведения эксперимента

Эксперимент состоит из трёх стадий:

1. Перевод грамматики в бинарную нормальную форму

Данная задача требует упразднения всех ε -переходов (нормальная форма), что решается алгоритмически с помощью представления всевозможных сентенциальных форм на предыдущем шаге для каждого ε -перехода.

Также в каждом правиле вывода грамматики должно быть не более двух нетерминалов в правой части без учёта нетерминалов для проверки контекста (бинарная форма). Данная задача решается с помощью введения дополнительных нетерминалов.

Такое упрощение правил вывода приведёт к чрезмерному усложнению грамматики: увеличению нетерминальных символов, утрате их интуитивного понимания. Однако описанные преобразования необходимы для правильного представления грамматики для потока входа анализатора.

2. Токенизация программы

Требуется представить исходный код программы, как последовательность токенов — специальных терминальных символов, удобных для анализа. Такой вид необходим для обеспечения удобного формата входных данных анализатора. Задача достигается с помощью использования библиотек и написанных программ.

найденных написанных программ, их настройки и адаптации для грамматики и анализатора.

3. Тестирование

Требуется произвести проверку грамматики на корректность работы как на правильных, так и на ошибочных примерах. Токенизированное представление программ и бинарная нормальная форма грамматики подаются на вход двустороннему синтаксическому анализатору. В случае отсутствия ошибок в написанной программе и разработанной грамматики, анализатор представляет мульти-дерево вывода исходной программы. В случае неправильности программы или же ошибочных правил грамматики анализатор выводит, что слово — токенизированная программа — грамматике не принадлежит.

Главная особенность мульти-дерева заключается в том, что отдельные вершины "умеют ссылаться" на другие вершины — корни других деревьев. Вершина, на которую происходит ссылка, отвечает за вывод контекста (левого или правого), проверка которого и производится. Благодаря данной конструкции построение вывода ошибочного слова позволяет наглядно отследить неточность.

6.2. Примеры задач

В целях упрощения восприятия и компактности ниже проиллюстрирован пример тестирования грамматики задающей язык $a^n b^n c^n, n > 0$. Более сложные примеры будут разобраны позже.

Грамматика, разобранная ранее, была переведена в нормальную бинарную форму, описанным выше способом. Стоит заметить, что прежде эта грамматика состояла 3 строки **вставить ссылку**.

```

grammar(S);
S = A MR;
MR = M C;
M = B & <= L & >= R;
L = AA LL;
LL = L BB;
L = AA BB;
R = BB RR;
RR = R CC;
R = BB CC;
A = A A;
B = B B;
C = C C;
A = "a";
B = "b";
C = "c";
AA = "a";
BB = "b";
CC = "c";

```

После этого полученная форма и слово (в данном примере это aabbcc) подавались на вход анализатору, который в силу корректности как грамматики, так и слова, ей принадлежащего, вывел мульти-дерево рабора, изображённое ниже.

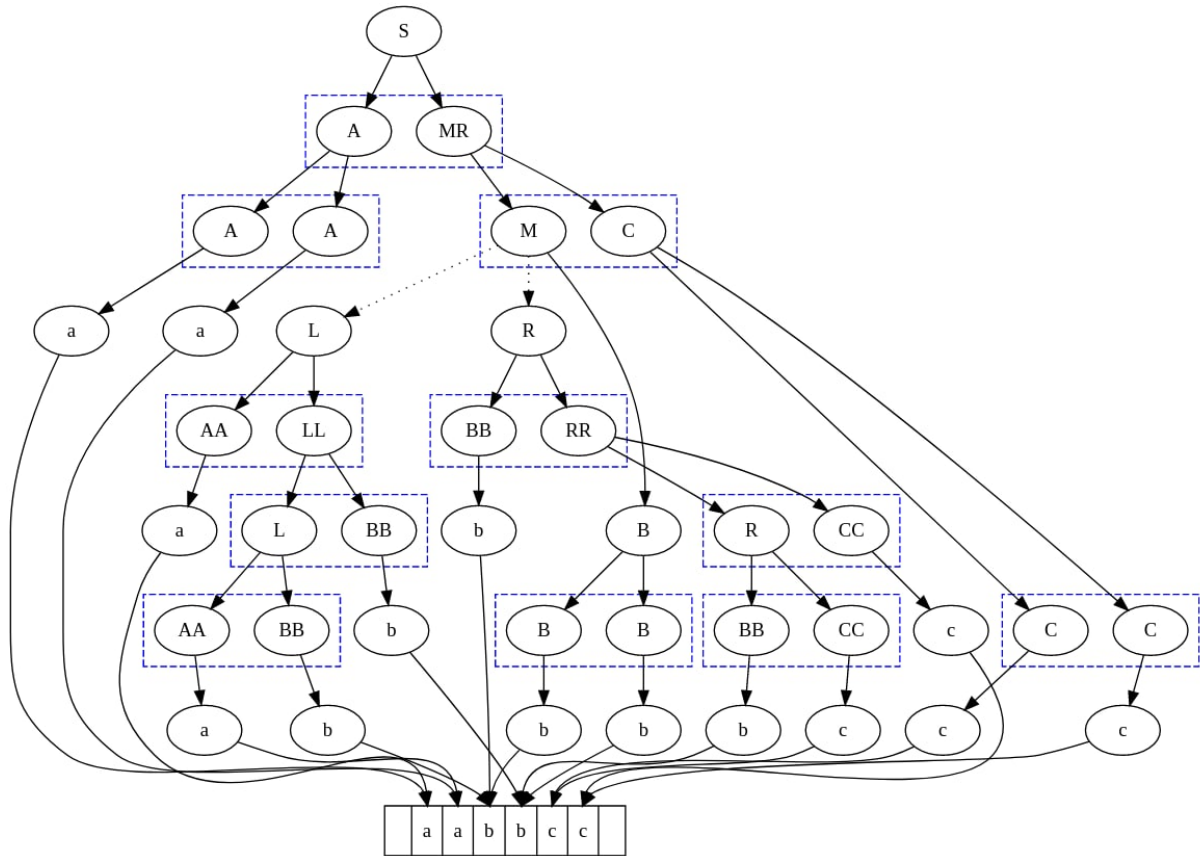


Рис. 1. Мульти-дерево вывода

6.3. Анализ результатов

Однако тестирование разработанной грамматики требует более сложных примеров с построением чрезмерно больших конструкций на промежуточных и выходных этапах. Все результаты как корректных, так и ошибочных тестов опубликованы в репозитории [10] на платформе <https://github.com>.

Тестирование грамматики было успешно проведено. Анализатор выводил ожидаемые конструкции на различных входных тестах. На основании чего можно сделать вывод, что данная грамматика позволяет идентифицировать вышеуказанные статические и динамические ошибки. Отсутствие ложных срабатываний в экспериментах обусловлено простотой входных тестов. Полнота входных тестов и анализ ложных срабатываний лежит за рамками данной бакалаврской работы.

7. Заключение

Выявлены достоинства и недостатки данного метода. А также рассмотрены перспективы применения данных конструкций в конгруэнтной области.

В заключении, стоит сказать, что цели работы были выполнены. Преимущества грамматики с контекстами, в том числе разработанной мной, были не только сформулированы теоретически, но и проверены эмпирически. На основании проведённой работы можно с уверенностью утверждать, что данный класс грамматик имеет достаточный ряд преимуществ в применении статического анализа. Однако имеет нелинейную асимптотику временем работы, что является его главным недостатком. В связи с чем напрашивается вывод, что использование данной конструкции может быть в разы эффективнее при смешанном использовании многих подходов анализа программ.

Список литературы

1. https://en.wikipedia.org/wiki/Rice%27s_theorem
2. <https://clang.llvm.org/>
3. <http://gcc.gnu.org/>
4. <https://ps-group.github.io/compiler/ast/>
5. https://en.wikipedia.org/wiki/Static_program_analysis
6. Barash M. Defining Contexts in Context-Free Grammars — 2015
7. https://en.wikipedia.org/wiki/Formal_grammar
8. <https://github.com/ilvivi/TwoSidedContextParser>
9. A. Okhotin, “Conjunctive grammars”, Journal of Automata, Languages and Combinatorics, 6:4 (2001), 519–535
10. <https://github.com/ilvivi/TwoSidedContextParser>