Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторной работе №5

По теме “Интерпретация исходного кода”

Выполнил:
студент гр. 853504
Шевченя И.В.
Проверил:
Ст. преподаватель КИ Шиманский В. В.

Минск 2021

1 Постановка задачи:

Разработка интерпретатора подмножества языка программирования, определённого в лабораторной работе 1, с использованием результатов анализа предыдущих лабораторных работ, т. е. с использованием лексического, синтаксического, семантического анализаторов.

2 Теория:

Язык процессора (устройства, машины) называется машинным языком, машинным кодом. Код на машинном языке исполняется процессором. Обычно, машинный язык — язык низкого уровня, но существуют процессоры, использующие языки высокого уровня (например, iAPX-432). Однако, такие процессоры не получили распространения в силу своей сложности и дороговизны.

Компилятор – это специальная программа, которая переводит текст программы, написанный на языке программирования, в набор машинных кодов.

Компиляция — сборка программы, включающая трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования высокого уровня и/или языке ассемблера, в эквивалентные программные модули на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера) или непосредственно на машинном языке или ином двоичнокодовом низкоуровневом командном языке и последующую сборку исполняемой машинной программы. Если компилятор генерирует исполняемую машинную программу на машинном языке, то такая программа непосредственно исполняется физической программируемой машиной (например компьютером). В других случаях исполняемая машинная программа выполняется соответствующей виртуальной машиной. Входной информацией для компилятора (исходный код) является описание алгоритма или программы на предметно-ориентированном языке, а на выходе компилятора — эквивалентное описание алгоритма на машинноориентированном языке (объектный код, байт-код).

Интерпретатор (от лат. *interpretator* - толкователь) — программа (разновидность транслятора), выполняющая интерпретацию.

Интерпретация — построчный анализ, обработка и выполнение исходного кода программы или запроса (в отличие от компиляции, где весь текст программы, перед запуском, анализируется и транслируется в машинный или байт-код, без её выполнения).

Транслятор — программа или техническое средство, выполняющее трансляцию программы.

Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т. д.

Язык, на котором представлена входная программа, называется **исходным языком**, а сама программа — **исходным кодом**. Выходной язык называется **целевым языком**, а выходная (результирующая) программа — **объектным кодом**.

Термин «**программа**» употребляется в вышеприведённых определениях в трёх значениях:

- Программа, которой является сам транслятор.
- Программа, текст которой подаётся на вход транслятора.
- Программа, текст которой является выходом транслятора.

2.а **Теория трансляции и интерпретации**

Существует несколько видов трансляторов:

- *Диалоговый транслятор* — транслятор, обеспечивающий использование языка программирования в режиме разделения времени.
- *Синтаксически-ориентированный (синтаксически-управляемый) транслятор* — транслятор, получающий на вход описание синтаксиса и семантики языка, текст на описанном языке и выполняющий трансляцию в соответствии с заданным описанием.
- *Однопроходной транслятор* — транслятор, создающий объектный модуль при однократном последовательном чтении исходного кода (за один проход).
- *Многопроходной транслятор* — транслятор, создающий объектный модуль после нескольких чтений исходного кода (за несколько проходов).
- *Оптимизирующий транслятор* — транслятор, выполняющий оптимизацию создаваемого кода перед записью в объектный файл.
- *Тестовый транслятор* — транслятор, получающий на вход исходный код и выдающий на выходе изменённый исходный код. Запускается перед основным транслятором для добавления в исходный код отладочных процедур. Например, транслятор с языка ассемблера может выполнять замену макрокоманд на код.

- *Обратный транслятор* — транслятор, выполняющий преобразование машинного кода в текст на каком-либо языке программирования.

Цель трансляции — преобразование текста с одного языка на язык, понятный адресату. При трансляции компьютерной программы адресатом может быть:

- устройство — процессор (трансляция называется компиляцией);
- программа — интерпретатор (трансляция называется интерпретацией).

Виды трансляции:

- компиляция;
- в исполняемый код;
- в машинный код;
- в байт-код;
- транспиляция;
- интерпретация;
- динамическая компиляция.

Понятия «трансляция» и «интерпретация» отличаются. Во время трансляции выполняется преобразование кода программы с одного языка на другой. Во время интерпретации программа исполняется.

Так как целью трансляции является, обычно, подготовка к интерпретации, эти процессы рассматриваются вместе. Например, языки программирования часто характеризуются как «компилируемые» или «интерпретируемые», в зависимости от того, что преобладает при использовании языка: компиляция или интерпретация. Причём, практически все языки низкого уровня и третьего поколения, вроде ассемблера, Си, являются компилируемыми, а более высокоуровневые языки, вроде Python или SQL — интерпретируемыми.

С другой стороны, существует взаимопроникновение процессов трансляции и интерпретации: интерпретаторы могут быть компилирующими (в том числе с динамической компиляцией), а в трансляторах может требоваться интерпретация для реализации метапрограммирования (например, для макросов в языке ассемблера, условной компиляции в Си или шаблонов в C++).

Более того, один и тот же язык программирования может и транслироваться, и интерпретироваться, и в обоих случаях должны

присутствовать общие этапы анализа и распознавания конструкций и директив исходного языка. Это относится и к программным реализациям, и к аппаратным — так, процессоры семейства x86 перед исполнением инструкций машинного языка выполняют их декодирование, выделяя в опкодах поля операндов (указание регистров, адресов в памяти, констант), разрядности и т. п., а в процессорах Pentium с архитектурой NetBurst тот же самый машинный код перед сохранением во внутреннем кэше дополнительно транслируется в последовательность микроопераций.

Альтернативой компиляции является интерпретация. Основная разница между компилятором и интерпретатором заключается в том, как они работают. Компилятор берет всю программу и преобразует ее в машинный код, который понимает процессор.

Интерпретатор выполняет программу поэтапно как часть собственного исполняемого файла. Объектный код не передается процессору, интерпретатор сам является объектным кодом, построенным таким образом, чтобы его можно было вызвать в определенное время.

Главные отличия между компилятором и интерпретатором:

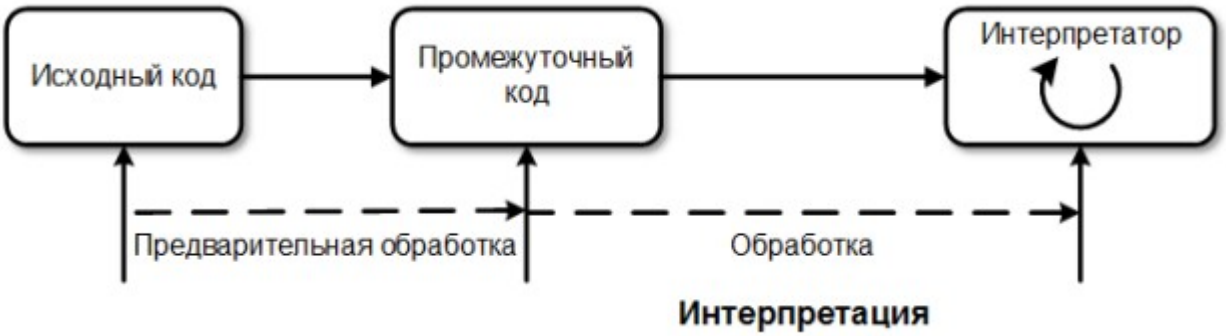
- Интерпретатор берет одну инструкцию, транслирует и выполняет ее, а затем берет следующую инструкцию. Компилятор же транслирует всю программу сразу, а потом выполняет ее.

- Компилятор генерирует отчет об ошибках после трансляции всего, в то время как интерпретатор прекратит трансляцию после первой найденной ошибки.

- Компилятор по сравнению с интерпретатором требует больше времени для анализа и обработки языка высокого уровня.

- Помимо времени на обработку и анализ, общее время выполнения кода компилятора быстрее в сравнении с интерпретатором.

Интерпретаторы могут создаваться по-разному. Существуют интерпретаторы, которые читают исходную программу и не выполняют



Интерпретация

дополнительной обработки. Они просто берут определенное количество строк кода за раз и выполняют его.

Некоторые интерпретаторы выполняют собственную компиляцию, но обычно преобразуют программу байтовый код, который имеет смысл только для интерпретатора. Это своего рода псевдомашинный язык, который понимает только интерпретатор.

Такой код быстрее обрабатывается, и его проще написать для исполнителя (части интерпретатора, которая исполняет), который считывает байтовый код, а не код источника.

Есть интерпретаторы, для которых этот вид байтового кода имеет более важное значение. Например, язык программирования Java «запускается» на так называемой виртуальной машине. Она является исполняемым кодом или частью программы, которая считывает конкретный байтовый код и эмулирует работу процессора. Обрабатывая байтовый код так, как если бы процессор компьютера был виртуальным процессором.

Простые интерпретаторы анализируют и выполняют (интерпретируют) программу последовательно (покомандно или построчно). Синтаксические ошибки обнаруживаются, когда интерпретатор приступает к выполнению команды (строки) содержащей ошибку. Сложные интерпретаторы компилирующего типа перед выполнением производят компиляцию исходного кода программы в машинный или «промежуточный код». Они быстрее выполняют большие и циклические программы, не занимаются анализом исходного кода в реальном времени. Некоторые интерпретаторы для начинающих программистов (преимущественно, для языка Бейсик) могут работать в режиме диалога, добавляя вводимую строку команд в программу (в памяти) или выполняя команды непосредственно.

Основным аргументом за использование процесса компиляции является скорость. Возможность компилировать любой программный код в машинный, который может понять процессор ПК, исключает использование промежуточного кода. Можно запускать программы без дополнительных шагов, тем самым увеличивая скорость обработки кода.

Но наибольшим недостатком компиляции является специфичность. Когда компилируете программу для работы на конкретном процессоре, вы создаете объектный код, который будет работать только на этом процессоре. Если хотите, чтобы программа запускалась на другой машине, вам придется перекомпилировать программу под этот процессор. А перекомпиляция может быть довольно сложной, если процессор имеет ограничения или особенности, не присущие первому. А также может вызывать ошибки компиляции.

Основное преимущество интерпретации — гибкость. Можно не только запускать интерпретируемую программу на любом процессоре или платформе, для которых интерпретатор был скомпилирован. Написанный интерпретатор может предложить дополнительную гибкость. В определенном смысле интерпретаторы проще понять и написать, чем компиляторы.

С помощью интерпретатора проще добавить дополнительные функции, реализовать такие элементы, как сборщики мусора, а не расширять язык.

Другим преимуществом интерпретаторов является то, что их проще переписать или перекомпилировать для новых платформ.

Написание компилятора для процессора требует добавления множества функций, или полной переработки. Но как только компилятор написан, можно скомпилировать кучу интерпретаторов и на выходе мы имеем перспективный язык. Не нужно повторно внедрять интерпретатор на базовом уровне для другого процессора.

Самым большим недостатком интерпретаторов является скорость. Для каждой программы выполняется так много переводов, фильтраций, что это приводит к замедлению работы и мешает выполнению программного кода.

Это проблема для конкретных real-time приложений, таких как игры с высоким разрешением и симуляцией. Некоторые интерпретаторы содержат компоненты, которые называются just-in-time компиляторами (JIT). Они

компилируют программу непосредственно перед ее исполнением. Это специальные программы, вынесенные за рамки интерпретатора. Но поскольку процессоры становятся все более мощными, данная проблема становится менее актуальной.

Во время цикла разработки программного обеспечения программисты вносят частые изменения в исходный код. При использовании компилятора, каждый раз, когда изменение было внесено в исходный код, они должны ожидать компилятор, чтобы перевести измененные исходные файлы и соединить все файлы двоичного кода, прежде чем программа может быть исполнена. Чем больше программа, тем дольше ожидание. В отличие от этого, программист, использующий интерпретатор, ждет намного меньше, поскольку интерпретатор обычно просто должен перевести код, работающий на промежуточном представлении (или не перевести его вообще), таким образом требуется намного меньшего количества времени, прежде чем изменения смогут быть протестированы. Эффекты заметны после сохранения исходного кода и перезагрузки программы. Скомпилированный код обычно с меньшей готовностью отлажен для редактирования, компиляции и соединения - последовательные процессы, которые должны быть проведены в надлежащей последовательности с надлежащим набором команд. Поэтому у многих компиляторов также есть исполнительное средство, известное как Make-файл и программа. Make-файл перечисляет командные строки компилятора и компоновщика и файлы исходного кода программы, но мог бы использовать простой ввод меню командной строки (например, "Make 3"), который выбирает третью группу (набор) инструкций, и тогда дает команды компилятору и компоновщику, которые подают указанные файлы исходного кода.

Каждая функция в реализованной программе собирается в отдельный скомпилированный объект со своим байт-кодом, представленный последовательностью различных чисел. Данная последовательность представляет собой код имени инструкции, операнды инструкции и потенциальные дополнительные байты идентификации для расширения количества возможных интерпретируемых переменных. Циклы и сравнения работают как переход на указанную инструкцию при отсутствии выполнения условия или продолжение в той же ленте байт кода, если условие выполняется. Разницы между ними нет, так как любой цикл можно записать с помощью операторов сравнения, возвращающихся к проверке условия до тех пор, пока то не будет выполнено.

В некоторые стандартные интерпретаторы были внесены следующие изменения:

1. Байткод переводчиков

Существует спектр возможностей интерпретации и компиляции, в зависимости от объема анализа, выполненного до выполнения программы. Этот" скомпилированный " код затем интерпретируется интерпретатором байт-кода (сам написан на языке C). Скомпилированный код в этом случае представляет собой машинный код для виртуальной машины, который реализуется не в аппаратном обеспечении, а в интерпретаторе байт-кода. Такие компилирующие переводчики иногда называют компиляторами. В интерпретаторе байт-кода каждая инструкция начинается с байта, и поэтому интерпретаторы байт-кода имеют до 256 инструкций, хотя не все могут быть использованы. Некоторые байт-коды могут принимать несколько байт и могут быть произвольно сложными.

Управляющие таблицы - это не обязательно когда-нибудь понадобится, чтобы пройти через составление поэтапного диктуют соответствующие алгоритмические поток управления через индивидуальные переводчики в аналогично байт-кода переводчиков.

2. Переводчики с резьбовым кодом

Потоковые интерпретаторы кода похожи на интерпретаторы байт-кода, но вместо байтов используются указатели. Каждая" инструкция " - это слово, указывающее на функцию или последовательность инструкций, за которой, возможно, следует параметр. Интерпретатор продетого нитку кода либо выполняет циклическую выборку инструкций и вызывает функции, на которые они указывают, либо выбирает первую инструкцию и прыгает к ней, и каждая последовательность инструкций заканчивается выборкой и переходом к следующей инструкции. В отличие от байткода нет эффективного ограничения на количество различных инструкций, кроме доступной памяти и адресного пространства. Классический пример многопоточности-четвертый код, используемый в открытых системах Прошивка: язык исходный код компилируется в код "Ф" (байт-код), который затем интерпретируется виртуальной машиной.

3. Абстрактные синтаксические интерпретаторы

В спектре между интерпретацией и компиляцией другой подход заключается в преобразовании исходного кода в оптимизированное абстрактное дерево синтаксиса (AST), а затем выполнении программы, следующей за этой древовидной структурой, или использовании его для генерации собственного кода просто в срок. При таком подходе каждое предложение должно быть проанализировано только один раз. Как преимущество над байт-код, АСТ сохраняет глобальную структуру программы и отношения между утверждениями (которые теряются в представлении байт-код), а при сжатии обеспечивает более компактное представление. Таким образом, использование AST было предложено в качестве лучшего промежуточного формата для компиляторов just-in-time, чем байткод. Кроме того, это позволяет системе выполнять лучший анализ во время выполнения.

Однако, для устных переводчиков, АСТ вызывает больше накладных расходов, чем байт-код интерпретатора, из-за узлов, связанных с синтаксисом, выполняющих никакой полезной работы, менее последовательное представление (требуется прохождение нескольких указателей) и накладных осмотреть дерево.

4. Сборник «точно в срок»

Дальнейшим размыванием различия между интерпретаторами, интерпретаторами байт-кода и компиляцией является компиляция just-in-time (JIT), метод, в котором промежуточное представление компилируется в машинный код машинного кода во время выполнения. Это повышает эффективность выполнения собственного кода за счет времени запуска и увеличения использования памяти при первой компиляции байт-кода или AST. Адаптивная оптимизация-это дополнительный метод, при котором интерпретатор профилирует запущенную программу и компилирует ее наиболее часто выполняемые части в машинный код. Оба метода несколько десятилетий, появляются в языках, таких как Smalltalk в 1980-х.

Просто в момент компиляции завоевала всеобщее внимание среди разработчиков языков в последние годы, с Java, на Framework, у большинства современных реализациях JavaScript, MATLAB, которые сейчас в том числе JITs.

5. Микрокод

Микрокод - это очень часто используемый метод, который накладывает интерпретатор между аппаратным и архитектурным уровнем компьютера.

Таким образом, микрокод является слоем аппаратных инструкций, которые реализуют инструкции машинного кода более высокого уровня или секвенсирование внутреннего состояния во многих элементах цифровой обработки. Микрокод используется в центральных процессорах общего назначения, а также в более специализированных процессорах, таких как микроконтроллеры, цифровые сигнальные процессоры, контроллеры каналов, контроллеры дисков, контроллеры сетевых интерфейсов, сетевые процессоры, графические процессоры и другие аппаратные средства.

Микрокод обычно находится в специальной высокоскоростной памяти и преобразует машинные инструкции, данные конечных автоматов или другой вход в последовательности подробных операций на уровне цепей. Он отделяет машинные инструкции от базовой электроники, так что инструкции могут быть спроектированы и изменены более свободно. Это также облегчает создание сложных многоступенчатых инструкций, одновременно уменьшая сложность компьютерных схем. Написание микрокода часто называют микропрограммированием, а микрокод в конкретной реализации процессора иногда называют микропрограммой.

Более обширное микрокодирование позволяет малым и простым микроархитексам эмулировать более мощные архитектуры с более широкой длиной слова, большим количеством исполнительных блоков и т. д., что является относительно простым способом обеспечения совместимости программного обеспечения между различными продуктами семейства процессоров.

3. Результат работы:

Проверим работу программы на нескольких тестовых примерах, кроме того в код будут добавлены блоки не имеющие смысла и привязанности к выполняемому алгоритму, чтобы проверить всё выбранное подмножество языка. Данная программа производит переворачивание массива и вывод его в обратном порядке, код самой программы указан в приложении снизу.

Исходный массив:

```
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr, n);
```

Примечание: исходные данные для работы программы указываются в статическом массиве в блоке/функции main.

Результат работы программы:

```
1 2 3 4 5 6
Reversed array is
6 5 4 3 2 1
10
```

Первая строка — исходный массив, третья строка — результат работы программы, кроме того в последней строке указано число, которое было присвоено переменной в условной конструкции if, сделано это для проверки работоспособности.

Возьмем входной массив значительно больше:

```
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25};

    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr, n);

    reverseArray(arr, 0, n - 1);

    cout << "Reversed array is";
    cout << endl;
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Reversed array is
25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
10
```

Попробуем внедрить ошибку и посмотреть работоспособность и

отображение ошибки:

```
    int a = 10;
    int b = 15;
    int t;
    if (a < b)
    {
        t = 10;
    }
    else {
        t = 15;
    }

    cout << ilya;

    return 0;
}
```

Использование не инициализированной переменной.

```
Traceback (most recent call last):
  File "<input>", line 1, in <module>
  File "/home/shevchenya/Загрузки/pycharm-professional-2020.1.2/pycharm-2020.1.2/plugins/python/helpers/pydev/_pydev_bundle/pydev_umd.py", lin
    pydev_imports.execfile(filename, global_vars, local_vars)  # execute the script
  File "/home/shevchenya/Загрузки/pycharm-professional-2020.1.2/pycharm-2020.1.2/plugins/python/helpers/pydev/_pydev_imps/_pydev_execfile.py",
    exec(compile(contents+"\n", file, 'exec'), glob, loc)
  File "/home/shevchenya/PycharmProjects/mtran/compiler.py", line 156, in <module>
    testing_work(tree)
  File "/home/shevchenya/PycharmProjects/mtran/compiler.py", line 73, in testing_work
    raise Exception(result.stderr.decode("utf-8"))
Exception: main_1.cpp: In function ‘int main()’:
main_1.cpp:55:13: error: ‘ilya’ was not declared in this scope
```

Вывод

В ходе работы были получены знания о видах трансляторов и интерпретаторов. Были изучены отличия между компилятором и интерпретатором. Были получены практические навыки разработки данного программного обеспечения. Язык программирования Python оказался очень достойным языком для изучения данного раздела и позволил просто и красиво разрабатывать необходимый функционал для приложения.

Приложение 1. Полный текст программы(включая код функций используемых в прошлых работах)

Код лексического анализатора:

```
import re

import texttable as tt
from ply import lex
from ply.lex import TOKEN

class Token:
    """
    docstring for Token

    """

    def __init__(self, value, tag, row, col):
        self.value = value
        self.tag = tag
        self.row = row
        self.col = col

    def __str__(self):
        return "<{}, {}, {}, {}>".format(self.value, self.tag,
self.row, self.col)

    def __repr__(self):
        return self.__str__()


class Lexer(dict):
    """
    docstring for Lexer

    """

    def __init__(self, file, *args):
        super().__init__(*args)
        self.pos, self.row, self.col = 0, 1, 1
        self.skip_end = False
        self.variable_type_defined = False
        self.char = ""
        self.file = open(file, "r")
        self.string = self.file.readline()
        self.errors_list = list()

    def errors(self):
        """
        print all errors

        """
        import sys
```

```
        self.file.close()
        sys.stderr.write("Lexer errors:\n")

        for i in self.errors_list:
            sys.stderr.write("\t%s\n" % i)

        sys.stderr.flush()
        exit(1)

    def error(self, text):
        """
        print error

        """
        self.errors_list.append(
            "{} in line {}, column {}".format(text, self.row,
self.col)
        )

    def check_end_of_line(self, pos):
        result = True
        while pos > 0 and self.string[pos] == " ":
            pos -= 1

        if self.string[pos] == ";":
            result = False

        return result

    def empty_line(self):
        line = self.string

        for char in line[:-1]:
            if char != " ":
                return False

        return True

    def skip_line(self):
        self.string = self.file.readline()
        self.skip_end = False
        self.col = 1
        self.row += 1
        self.pos = 0

    def next_char(self):
        """
        set next char

        """
        if self.pos < len(self.string):
            self.char = self.string[self.pos]
```

```
            if self.char != "\n":
                self.col += 1
                self.pos += 1
            else:
                if self.check_end_of_line(self.pos - 1):
                    if not self.skip_end and not
self.empty_line():
                        self.error("Missing end of line: ")

                self.skip_line()
        else:
            self.char = "#0"

    def skip_space(self):
        """
        skip spaces

        """
        while self.char.isspace():
            self.next_char()

    @staticmethod
    def compare_signs(lexeme):
        possible_signs = {
            "=": EQUAL_SIGN,
            "==": EQUAL,
            "!=": NOT_EQUAL,
            "<": LT,
            ">": GT,
            "<=": LE,
            ">=": GE,
        }

        return possible_signs.get(lexeme, None)

    @staticmethod
    def arithmetics_function(lexeme):
        possible_signs = {
            "=": EQUAL_SIGN,
            "==": EQUAL,
            "!=": NOT_EQUAL,
            "<": LT,
            ">": GT,
            "<=": LE,
            ">=": GE,
        }

        return possible_signs.get(lexeme, None)

    @staticmethod
    def logical_operation(lexeme):
        possible_operation = {
            "&&": AND,
```

```
            "||": OR,
        }

        return possible_operation.get(lexeme, None)

    def is_build_in_function(self, lexeme):
        possible_func_names = {
            "if": IF,
            "else": ELSE,
            "while": WHILE,
            "for": FOR,
            "break": BREAK,
            "continue": CONTINUE,
            "return": RETURN,
            "printf": FUNC,
            "getchar": FUNC,
            "endl": FUNC,
            "cout": FUNC,
            "sizeof": FUNC,
        }

        if not self.variable_type_defined:
            return possible_func_names.get(lexeme, None)
        else:
            self.error(f"Undefined function type: {lexeme}")

    def is_function(self):
        if self.string[self.pos - 1] == "(":
            self.variable_type_defined = False
            return True

        return False

    def number_conversion(self, lexeme=""):
        """
        Parsing numbers: float or integer, catch incorrect
number input
        :param lexeme: str
        :return: Token
        """
        if self.char.isdigit():
            count = 0
            sign = 1 if lexeme == "+" or lexeme == "" else -1

            while self.char.isdigit() or self.char == ".":
                if self.char == ".":
                    count += 1

                lexeme += self.char
                self.next_char()

            if count > 1:
                self.error('Incorrect format of number: "%s"' %
```

```
lexeme)
                return None
            else:
                return Token(
                    sign * (int(lexeme)) if count == 0 else sign
* (float(lexeme)),
                    NUMBER,
                    self.row,
                    self.col,
                )

    def check_names(self, lexeme):
        """
        Parsing name. Defining functions, variables, build-in
functions
        :param lexeme: str
        :return: Token
        """

        token = None

        if self.variable_type_defined:
            if self.is_function():
                self.variable_type_defined = False
                token = Token(lexeme, FUNC_DECLARATION,
self.row, self.col)
        else:
            if (func_type := self.is_build_in_function(lexeme))
is not None:
                token = Token(lexeme, func_type, self.row,
self.col)
            elif lexeme in VARIABLE_TYPES:
                self.variable_type_defined = True
                token = Token(lexeme, TYPE, self.row, self.col)

        return token

    def check_operation(self, lexeme):
        """
        Parsing different operations
        :param lexeme: str
        :return: Token
        """

        token = None

        if lexeme in ARITHMETIC_OPERATIONS:
            token = Token(lexeme, "ARITHMETIC_OPERATIONS",
self.row, self.col)
        elif lexeme in OVERRIDE_OPERATION:
            token = Token(lexeme, "OVERRIDE_OPERATION",
self.row, self.col)
        elif (logical_operation :=
```

```
self.logical_operation(lexeme)) is not None:
            token = Token(lexeme, logical_operation, self.row,
self.col)
        elif (sign_type := self.compare_signs(lexeme)) is not
None:
            token = Token(lexeme, sign_type, self.row, self.col)

        return token

    def processing_bracket(self, bracket, *, skip_end=False):
        """
        Return Token for different types of brackets
        :param bracket: str
        :param skip_end: bool
        :return: Token
        """

        possible_brackets = {
            "(": L_PAR,
            ")": R_PAR,
            "[": L_SQUARE,
            "]": R_SQUARE,
            "{": L_CURL,
            "}": R_CURL,
        }

        if skip_end:
            self.skip_end = True

        return Token(bracket, possible_brackets[bracket],
self.row, self.col)

    def check_brackets(self):
        """
        Check which type of brackets is used
        :return: Token
        """

        lexeme = self.char
        token = None

        if lexeme in ("(", ")"):
            token = self.processing_bracket(lexeme,
skip_end=True)
        elif lexeme in ("[", "]"):
            token = self.processing_bracket(lexeme)
        elif lexeme in ("{", "}"):
            token = self.processing_bracket(lexeme,
skip_end=True)

        self.next_char()

        return token
```

```
    def next_token(self):
        """
        Parsing code file and getting tokens
        :return: Token
        """

        self.skip_space()
        lexeme = ""

        if self.char.isalpha() or self.char == "_":
            lexeme = self.char
            self.next_char()

            while self.char.isalpha() or self.char.isdigit():
                lexeme += self.char
                self.next_char()

            if (token := self.check_names(lexeme)) is not None:
                return token

            if not self.variable_type_defined:
                pos = self.pos

                while self.string[pos] == " ":
                    pos += 1

                if self.string[pos] == "(":
                    self.error(f"Undefined function type
'{lexeme}'")
                    return None

            self.variable_type_defined = False
            return Token(lexeme, ID, self.row, self.col)

        elif self.char in "+-*%><=^!?&|":
            lexeme, count = self.char, 1
            self.next_char()

            while self.char in "+-*%><=^!?&|":
                lexeme += self.char
                count += 1
                self.next_char()

            if count > 2:
                self.error('Incorrect format of operation: "%s"'
% lexeme)
                return None
            else:
                if lexeme in ("-", "+"):
                    sign = lexeme
                    self.next_char()
```

```
                    return self.number_conversion(sign)

                elif (token := self.check_operation(lexeme)) is
not None:
                    return token

            self.error('Undefined operation: "%s"' % lexeme)

        elif self.char.isdigit():
            return self.number_conversion()

        elif self.char in ("(", ")", "{", "}", "[", "]"):
            return self.check_brackets()

        elif self.char == "#0":
            return Token("EOF", None, self.row, self.col)

        elif self.char == "/":
            lexeme = self.char
            self.next_char()
            if self.char in ("/", "*"):
                return self.skip_comments("\n" if self.char ==
"/" else "/")

            return Token(lexeme, "ARITHMETIC_OPERATIONS",
self.row, self.col)

        elif self.char in ('"', "'"):
            character, count = self.char, 0
            self.next_char()

            while self.char != character:
                count += 1
                condition, lexeme = self.parse_line_end(lexeme)
                if condition:
                    continue

                lexeme += self.char
                self.next_char()

            self.next_char()

            if character == "'":
                if count == 1:
                    return Token(lexeme, CHAR, self.row,
self.col)
            elif character == '"':
                return Token(lexeme, STRING, self.row, self.col)

            self.error("Incorrect quotes: '%s'" % lexeme)

        elif self.char in (";", ","):
            lexeme = self.char
```

```
            self.next_char()
            return Token(
                lexeme, SEMICOLON if lexeme == ";" else COMMA,
self.row, self.col
            )

        elif self.char == "\n":
            self.pos -= 1
            self.col -= 1
            self.next_char()
            return None

        elif self.char in self:
            lexeme = self.char
            self.next_char()
            return Token(lexeme, self[lexeme], self.row,
self.col)

        else:
            lexeme = self.char
            self.error('Unknown character: "%s"' % self.char)
            self.next_char()
            return Token(lexeme, UNKNOWN, self.row, self.col)

        return None

    def parse_line_end(self, lexeme):
        """
        Parsing symbol of line end inside C++ char or string
types
        :param lexeme: str
        :return: (bool, str)
        """

        if self.char == "\\":
            lexeme += self.char
            self.next_char()
            lexeme += self.char
            self.next_char()

            return True, lexeme

        return False, lexeme

    def skip_comments(self, char):
        """
        Base of condition skipping line content in comment
        :param char: str - comment end character
        :return: Token()
        """

        self.skip_end = True
```

```
            while self.char != char:
                self.next_char()

            self.next_char()

            return self.next_token()

        def get_token(self):
            """
            Returning token
            :return: Token
            """

            self.next_char()
            while True:
                result = self.next_token()

                if not result:
                    continue

                if result.value == "EOF":
                    break

                yield result

        def tokens(self):
            """
            Returning list of parsing tokens
            :return: list
            """

            result = [i for i in self.get_token()]
            return result

        def raw_input(self, user_string):
            """
            Return raw user input
            :param user_string: str
            :return: list
            """

            self.string = user_string
            return self.tokens()


    def draw_tags_groups(tokens):
        tokens_copy = copy.deepcopy(tokens)
        tokens_copy.sort(key=lambda x: x.tag)
        tag_names = {*[token.tag for token in tokens_copy]}
        tables = []
        for name in tag_names:
            table = tt.Texttable()
            table.header(["Value", "Row", "Column"])
```

```
        for token in filter(lambda x: x.tag == name,
tokens_copy):
            table.add_row((token.value, token.row, token.col))

        tables.append((name, table))

    for name, table in tables:
        print("Tag:", name)
        print(table.draw())
        print()


def draw_result_table(tokens):
    tab = tt.Texttable()
    headings = ["Value (token)", "Tag", "Row", "Column"]
    tab.header(headings)

    values = list()
    tags = list()
    rows = list()
    columns = list()

    for token in tokens:
        values.append(token.value)
        tags.append(token.tag)
        rows.append(token.row)
        columns.append(token.col)

    for row in zip(values, tags, rows, columns):
        tab.add_row(row)
    s = tab.draw()
    print(s)


def check_if_main_exist(tokens):
    return list(
        filter(lambda x: x.tag == FUNC_DECLARATION and x.value
== "main", tokens)
    )


def syntax_analyzer(ast, tabs):
    for i in ast:
        if isinstance(i, list):
            syntax_analyzer(i, tabs + 1)
        else:
            result, value = tabs * "  |", i.value if not
isinstance(i, str) else i
            print("{}{}".format(result, value))
```

```
tokens = (
    "FUNCDECL",
    "LPAR",
    "RPAR",
    "COMMA",
    "LCURL",
    "RCURL",
    "LCUADR",
    "RCUADR",
    "CUSTOM_FUNC",
    "EQUAL",
    "SEMICOLON",
    "NUMBER",
    "VARIABLE_TYPE",
    "ID",
    "BUILD_IN",
    "PLUSMINUS",
    "DIVMUL",
    "STRING",
    "IF",
    "ELSE",
    "DEQUAL",
    "RETURN",
    "GT",
    "LT",
    "GE",
    "LE",
    "MOD",
    "NOTEQUAL",
    "WHILE",
    "FOR",
    "CONTINUE",
    "BREAK",
)


types = {
    "int": "VARIABLE_TYPE",
    "float": "VARIABLE_TYPE",
    "double": "VARIABLE_TYPE",
    "char": "VARIABLE_TYPE",
    "void": "VARIABLE_TYPE",
}


reserved = {
    "if": "IF",
    "else": "ELSE",
    "auto": "VARIABLE_TYPE",
    "while": "WHILE",
    "for": "FOR",
    "break": "BREAK",
```

```
    "continue": "CONTINUE",
    "return": "RETURN",
    "sizeof": "BUILD_IN",
    "cout": "BUILD_IN",
    "endl": "BUILD_IN",
}

identifier = r"[a-zA-Z]\w*"

t_LCUADR = r"\["
t_RCUADR = r"\]"
t_LPAR = r"\("
t_RPAR = r"\)"
t_COMMA = r","
t_LCURL = r"\{"
t_RCURL = r"\}"
t_DEQUAL = r"\=\="
t_GE = r"\>\="
t_LE = r"\<\="
t_GT = r"\>"
t_LT = r"\<"
t_MOD = r"\%"
t_NOTEQUAL = r"!\="
t_EQUAL = r"\="
t_SEMICOLON = r";"
t_PLUSMINUS = r"\+|\-"
t_DIVMUL = r"/|\*"
t_STRING = r'("(\\.|[^"])*")|(\'(\\.|[^\'])*\')'

t_ignore = " \r\t\f"


def t_comment_ignore(t):
    r'/[*][^*]*[*]+([^/*][^*]*[*]+)*/|//[^\n]*'
    pass


def t_ignore_imports(t):
    r'#include <.*>'
    pass


def t_ignore_namespace(t):
    r'using namespace std;'
    pass


def t_newline(t):
    r"\n+"
    t.lexer.lineno += len(t.value)


def t_NUMBER(t):
```

```
    r"[0-9.]+"
    try:
        t.value = int(t.value)
    except BaseException:
        try:
            t.value = float(t.value)
        except BaseException:
            t.value = None
    return t


class TypeDefine:
    type_define = False


@TOKEN(identifier)
def t_ID(t):
    if TypeDefine.type_define:
        TypeDefine.type_define = False
        if t.lexer.lexdata[t.lexpos + len(t.value)] == "(":
            reserved[t.value] = "CUSTOM_FUNC"
            t.type = "FUNCDECL"
        else:
            t.type = "ID"
    else:
        if t.lexer.lexdata[t.lexpos + len(t.value)] == "(":
            if (value := reserved.get(t.value, None)) is None:
                print("error")
            else:
                t.type = value
        else:
            if (res := types.get(t.value, "ID")) ==
"VARIABLE_TYPE":
                TypeDefine.type_define = True

            t.type = res if t.value not in reserved else
reserved[t.value]

    return t


def t_error(t):
    print("Illegal character '%s' at line %d" % (t.value[0],
t.lineno))
    t.lexer.skip(1)


data = """
#include <iostream>
using namespace std;
void reverseArray(int arr[], int start, int end)
{
    while (start < end)
```

```
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}

void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
       cout << arr[i];
       cout << " ";
   }
   cout << endl;
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr, n);

    reverseArray(arr, 0, n - 1);

    cout << "Reversed array is";
    cout << endl;

    printArray(arr, n);

    if (a < b)
    {
        int t = 10;
    }
    else {
        int t = 15;
    }

    return 0;
}"""


lexer = lex.lex(reflags=re.UNICODE | re.DOTALL)


def display_tokens():
    tab = tt.Texttable()
    headings = ["Value (token)", "Tag", "Row", "Column"]
    tab.header(headings)
    lexer.input(data)
```

```
    character_number, line_number = 0, 0
    while True:
        tok = lexer.token()

        if not tok:
            break

        if line_number < tok.lineno:
            character_number = tok.lexpos
            line_number = tok.lineno

        tab.add_row(
            (
                tok.value,
                tok.type,
                tok.lineno,
                tok.lexpos - character_number +
len(str(tok.value)),
            )
        )
    s = tab.draw()
    print(s)


if __name__ == "__main__":
    path = "main_1.cpp"
    lexer = Lexer(path)
    parser = Parser()
    tokens = lexer.tokens()

    if lexer.errors_list:
        lexer.errors()

    draw_result_table(tokens)
    draw_tags_groups(tokens)

    ast = parser.build(tokens)
    syntax_analyzer(ast, 2)
    display_tokens()
```

Код синтаксического анализатора:

```
import re

from ply import yacc

import lexical_analysis
from lexical_analysis import tokens


class Node(object):
    def parts_str(self):
        return "\n".join(map(str, [x for x in self.parts]))
```

```
    def __init__(self, type, parts):
        self.type = type
        self.parts = parts

    def __str__(self):
        return self.type + ":\n____" +
self.parts_str().replace("\n", "\n____")

    def add_parts(self, parts):
        self.parts += parts
        return self


def p_program(p):
    """program :
    | function
    | program function
    """
    if len(p) <= 2:
        p[0] = Node("program", p[1:])
    else:
        p[0] = p[1].add_parts([p[2]])


def p_function(p):
    """function : func_header func_body"""

    p[0] = Node("function", p[1:])


def p_func_header(p):
    """func_header : VARIABLE_TYPE FUNCDECL LPAR args RPAR"""
    p[0] = Node("func_declaration", [p[1], p[2], p[4]])


def p_args(p):
    """args :
    | expr
    | args COMMA expr"""
    if len(p) <= 2:
        p[0] = Node("args", p[1:] if p[1:] else ["EMPTY"])
    else:
        p[0] = p[1].add_parts([p[3]])


def p_func_body(p):
    """func_body : block"""
    p[0] = p[1]


def p_block(p):
    """block : LCURL body RCURL"""
```

```
    p[0] = Node("block", [p[2]])


def p_body(p):
    """body :
    | body line semicolons
    | body multiline"""
    if len(p) > 1:
        if p[1] is None:
            p[1] = Node("body", [])
        p[0] = p[1].add_parts([p[2]])
    else:
        p[0] = Node("body", [])


def p_semicolons(p):
    """semicolons : SEMICOLON
    | semicolons SEMICOLON"""


def p_multiline(p):
    """multiline : if_statement
    | while_statement
    | for_statement"""
    p[0] = p[1]


def p_line(p):
    """line : modal_function
    | init
    | func
    | assign"""
    p[0] = p[1]


def p_modal_function(p):
    """modal_function : RETURN arg
    | BREAK
    | CONTINUE"""
    if len(p) == 3:
        p[0] = Node("modal_function", p[1:])
    else:
        p[0] = Node("modal_function", [p[1]])


def p_var_call(p):
    """var_cal : ID LCUADR expr RCUADR"""
    p[0] = Node("var_call", [p[1], p[3]])


def p_if_statement(p):
    """if_statement : IF LPAR condition RPAR block
    | if_statement ELSE block"""
```

```
    if len(p) == 4:
        p[0] = p[1].add_parts(["else", p[3]])
    else:
        p[0] = Node("if", [p[3], p[5]])


def p_while_statement(p):
    """while_statement : WHILE LPAR condition RPAR block"""
    p[0] = Node("while", [p[3], p[5]])


def p_for_statement(p):
    """for_statement : FOR LPAR init SEMICOLON condition
SEMICOLON change_val RPAR block"""
    p[0] = Node("for", [p[3], p[5], p[7], p[9]])


def p_change_value(p):
    """change_val : ID expr"""
    p[0] = Node("change_val", p[1:])


def p_condition(p):
    """condition : expr cond_sign expr"""
    p[0] = Node("condition", [p[1], p[2], p[3]])


def p_cond_sign(p):
    """cond_sign : DEQUAL
    | GT
    | LT
    | GE
    | LE
    | NOTEQUAL"""
    p[0] = p[1]


def p_init(p):
    """init :
    | VARIABLE_TYPE ID
    | VARIABLE_TYPE ID EQUAL ID DIVMUL NUMBER
    | VARIABLE_TYPE ID EQUAL expr
    | VARIABLE_TYPE ID EQUAL var_cal
    | VARIABLE_TYPE ID LCUADR RCUADR EQUAL array_init"""
    if len(p) > 5:
        p[0] = Node("init", [p[1], p[2], "[]", p[5], p[6]])
    else:
        p[0] = Node("init", p[1:])


def p_array_init(p):
    """array_init : LCURL init_block RCURL"""
    p[0] = Node("array_init", [p[2]])
```

```
def p_init_block(p):
    """init_block : arg
    | arg COMMA
    | init_block arg
    | init_block arg COMMA"""
    if len(p) == 2:
        p[0] = Node("init_block", p[1:])
    else:
        if p[2] != ",":
            p[0] = p[1].add_parts(p[2:])
        else:
            p[0] = Node("init_block", p[1:])


def p_assign(p):
    """assign : ID EQUAL expr
    | ID EQUAL var_cal
    | var_cal EQUAL expr
    | var_cal EQUAL var_cal
    | ID expr"""
    if len(p) == 5:
        p[0] = Node("assign", [p[2], p[4]])
    elif len(p) == 4 or len(p) == 3:
        p[0] = Node("assign", p[1:])
    else:
        p[0] = Node("assign", [p[1], p[3]])


def p_func(p):
    """func : CUSTOM_FUNC LPAR args RPAR
    | ID LPAR args RPAR
    | BUILD_IN LPAR args RPAR
    | BUILD_IN output_operator"""
    if len(p) == 3:
        p[0] = Node("func_call", [p[1], p[2]])
    else:
        p[0] = Node("func_call", [p[1], p[3]])


def p_output_operator(p):
    """output_operator : LT LT arg
    | LT LT BUILD_IN
    | LT LT ID"""

    p[0] = Node("output_operator", ["<<", p[3]])


def p_expr(p):
    """expr : fact
    | PLUSMINUS PLUSMINUS
    | expr PLUSMINUS fact
```

```
    | expr MOD fact
    | ID"""
    if len(p) == 2:
        p[0] = p[1]
    elif len(p) == 3:
        if p[2] == "+":
            p[0] = Node("increment", ["++"])
        elif p[2] == "-":
            p[0] = Node("decrement", ["--"])
    else:
        p[0] = Node(p[2], [p[1], p[3]])


def p_fact(p):
    """fact : term
    | fact DIVMUL term"""
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = Node(p[2], [p[1], p[3]])


def p_term(p):
    """term : arg
    | LPAR expr RPAR"""
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = p[2]


def p_arg(p):
    """arg : NUMBER
    | STRING
    | VARIABLE_TYPE ID
    | VARIABLE_TYPE ID LCUADR RCUADR
    | ID LCUADR RCUADR
    | var_cal
    | NUMBER ID
    | func"""
    if len(p) == 2:
        p[0] = Node("arg", [p[1]])
    else:
        p[0] = Node("arg", p[1:])


def p_error(p):
    raise Exception("Unexpected token in line %d: %s" %
(p.lineno, p))


def build_tree(code):
    parser = yacc.yacc()
```

```
    return parser.parse(code)


if __name__ == "__main__":
    tree = build_tree(data)
    print(tree)
```

Код семантического анализатора:

```
import re
from operator import add, mul, sub, truediv

from lexical_analysis import types
from syntax_analysis import build_tree


operations = "+-*/"
operation_resolver = {"+": add, "-": sub, "*": mul, "/":
truediv}
condition_resolver = ["<", ">", "<=", ">=", "==", "!="]
variables = {}
functions = {}


def parse_function_args(node):
    function_arguments = {}

    for part in node.parts:
        is_array = False

        if isinstance(part, str):
            break

        if len(part.parts) > 2:
            is_array = True

        function_arguments[part.parts[1]] = {
            "type": part.parts[0],
            "is_array": is_array,
        }

    return function_arguments


def parse_tree(tree):
    try:
        tree_type = tree.type
        parts = tree.parts
    except AttributeError:
        return tree

    if tree_type == "func_declaration":
        functions[parts[1]] = {
```

```
            "return": parts[0],
            "args": parse_function_args(parts[2]),
            "additional_args": {},
        }
        return

    if tree_type == "init":
        if len(parts) > 3:
            value = parts[3:]
        else:
            value = None

        functions[list(functions.keys())[-1]]["additional_args"]
[parts[1]] = {
            "type": parts[0],
            "value": value,
        }
        return

    if tree_type == "condition":
        cond = [
            *functions[list(functions.keys())[-1]]["args"],
            *list(functions[list(functions.keys())[-1]]
["additional_args"].keys()),
        ]

        first = parse_tree(parts[0])
        second = parse_tree(parts[2])
        possible_types = list(types.keys())
        if (
            parts[1] in condition_resolver
            and (first in cond or type(first).__name__ in
possible_types)
            and (second in cond or type(second).__name__ in
possible_types)
        ):
            first_temp = functions[list(functions.keys())[-1]]
["args"].get(
                parts[0], False
            ) or functions[list(functions.keys())[-1]]
["additional_args"].get(
                parts[0], False
            )
            second_temp = functions[list(functions.keys())[-1]]
["args"].get(
                parts[2], False
            ) or functions[list(functions.keys())[-1]]
["additional_args"].get(
                parts[2], False
            )

            if not first_temp:
                first_temp = type(first).__name__
```

```
            else:
                first_temp = first_temp["type"]

            if not second_temp:
                second_temp = type(second).__name__
            else:
                second_temp = second_temp["type"]

            if first_temp == second_temp:
                return

        raise TypeError("Wrong operand types in condition")

    if tree_type == "var_call":
        cond = [
            *functions[list(functions.keys())[-1]]["args"],
            *list(functions[list(functions.keys())[-1]]
["additional_args"].keys()),
        ]
        if parts[0] in cond and (parts[1] in cond or
parts[1].type == "arg"):
            arg = parse_tree(parts[1])
            if arg in cond or isinstance(arg, int):
                return arg
            else:
                raise ValueError(f"Forbidden argument type in
call {arg}")
        else:
            raise NameError(f"Unknown variable name {parts[0]}
or {parts[1]}")

    if tree_type == "func_call":
        function = parts[0]
        if function == "cout":
            output_value = parse_tree(parts[1].parts[1])
            try:
                if output_value.type == "var_call":
                    output_type =
functions[list(functions.keys())[-1]]["args"].get(
                        output_value.parts[0], False
                    ) or functions[list(functions.keys())[-1]]
["additional_args"].get(
                        output_value.parts[0], False
                    )
                    if output_type["type"] in ["int", "char",
"string"]:
                        return
                    else:
                        raise ValueError(
                            f"Output operator can't display
value {output_type}"
                        )
            except AttributeError:
```

```
                if (
                    isinstance(output_value, (int, float, str))
                    or output_value == "endl"
                ):
                    return
        elif function in functions.keys():
            arguments = parts[1].parts

            if len(arguments) != len(functions[function]
["args"]):
                raise Exception(
                    f"Wrong count of arguments passing to
function {function}"
                )

            for index, arg in enumerate(arguments):
                try:
                    arg = parse_tree(arg.parts[1])
                except Exception:
                    arg = parse_tree(arg)

                argument = get_type(arg)

                if (
                    argument
                    != functions[function]["args"][
                        list(functions[function]["args"].keys())
[index]
                    ]["type"]
                ):
                    raise ValueError(
                        f"Wrong argument type passing to
function {function}"
                    )
            return

    if tree_type == "assign":
        assign_arguments = []
        for part in parts:
            if part == "=":
                continue

            try:
                if part.type == "var_call":
                    argument = get_type(part.parts[0])

                    assign_arguments.append(argument)
            except AttributeError:
                arg = parse_tree(part)
                argument = get_type(arg)

                assign_arguments.append(argument)
```

```
        if not (
            len(assign_arguments) == 2
            and assign_arguments[0] == assign_arguments[1]
            or len(assign_arguments) == 1
        ):
            raise ValueError(
                f"Can't convert {assign_arguments[1]} to
{assign_arguments[0]}"
            )

    if tree_type == "modal_function":
        if parts[0] == "return":
            argument = parse_tree(parts[1])
            try:
                if eval(functions[list(functions.keys())[-1]]
["return"]) == type(
                    argument
                ):
                    return
                else:
                    raise ValueError("Incorrect return value
from function")
            except Exception:
                raise ValueError(
                    "Using return statement in function than
return 'void'"
                )

        return

    if tree_type == "arg":
        arg = parts[0]
        try:
            if arg.type == "var_call":
                return arg
        except Exception:
            pass

        if isinstance(arg, int):
            return arg
        elif isinstance(arg, float):
            return arg
        elif len(parts) == 1 and re.match(r"(\".*\")|(\'.*\')",
arg):
            return arg
        return

    if tree_type in operations:
        first = parse_tree(parts[0])
        second = parse_tree(parts[1])
        if type(first) != type(second):
            raise TypeError(
                "Types mismatch: {0} and
```

```
{1}".format(type(first), type(second))
            )
        if tree_type == "/" and second == 0:
            raise ZeroDivisionError("Unacceptable operation:
division by zero")
        return operation_resolver[tree_type](first, second)

    for part in parts:
        if part != "=":
            parse_tree(part)


def get_type(value):
    argument = functions[list(functions.keys())[-1]]
["args"].get(
        value, False
    ) or functions[list(functions.keys())[-1]]
["additional_args"].get(value, False)
    if not argument:
        argument = type(value).__name__
    else:
        argument = argument["type"]

    return argument


def check_inits():
    for func in functions:
        additional_args = functions[func]["additional_args"]
        for key, value in additional_args.items():
            value_type = value["value"]
            try:
                if value_type[0].type == "var_call":
                    argument = get_type(value_type[0].parts[0])
                elif value_type[0].type == "arg":
                    arg = parse_tree(value_type[0])
                    argument = get_type(arg)
            except (AttributeError, TypeError):
                pass

            if value_type is not None and argument !=
value["type"]:
                raise ValueError(
                    f"Wrong initialization of variable: type
{argument} can't be equal {value['value']}"
                )


if __name__ == "__main__":
    tree = build_tree(data)
    parse_tree(tree)

    if not functions.get("main", False):
```

```
        raise Exception(
            "Program should have starting point as function with
name 'main'"
        )

    check_inits()
    print(tree)
```

Интерпретатор:

```
import subprocess
import lexical_analysis
from lexical_analysis import tokens, types, display_tokens
from syntax_analysis import build_tree
from semantic_analysis import semantic_analysis
import operator as op
from lexer import ID, Token


OPERATIONS = dict()
OPERATIONS['+'] = lambda env, *x: obs(env, op.add, *x)
OPERATIONS['-'] = lambda env, *x: obs(env, op.sub, *x)
OPERATIONS['*'] = lambda env, *x: obs(env, op.mul, *x)
OPERATIONS['/'] = lambda env, *x: obs(env, op.truediv, *x)
OPERATIONS['//'] = lambda env, *x: obs(env, op.floordiv, *x)
OPERATIONS['%'] = lambda env, *x: obs(env, op.mod, *x)
OPERATIONS['='] = lambda env, *x: obs(env, op.eq, *x)
OPERATIONS['/='] = lambda env, *x: obs(env, op.ne, *x)
OPERATIONS['>'] = lambda env, *x: obs(env, op.gt, *x)
OPERATIONS['<'] = lambda env, *x: obs(env, op.lt, *x)
OPERATIONS['>='] = lambda env, *x: obs(env, op.ge, *x)
OPERATIONS['<='] = lambda env, *x: obs(env, op.le, *x)
OPERATIONS['~'] = lambda env, *x: obs(env, op.ne, *x)
OPERATIONS['setq'] = lambda env, *x: define_new_variable(env,
*x)
OPERATIONS['defun'] = lambda env, *x: define_function(env, *x)
OPERATIONS['if'] = lambda env, *x: compare(env, *x)
OPERATIONS['write'] = lambda env, *x: write(env, *x)
OPERATIONS['print'] = lambda env, *x: write_line(env, *x)
OPERATIONS['readint'] = lambda env, *x: read_integer(env, *x)


class Procedure(object):
    def __init__(self, params, *body):
        self.params, self.body = params, body

    def __call__(self, env, *args):
        if len(args) != len(self.params):
            msg = "Too many args! Expected %s, given %s" %
(len(self.params), len(args))
            msg += ' in line {}, column {}'.format(args[0].col,
args[0].row)
            raise TypeError(msg)
```

```
        for i, par in enumerate(self.params):
            env[par.value] = execute(args[i], env)

        magic = False
        while True:
            if magic:
                for i, par in enumerate(self.params):
                    env[par.value] = args[i]

            length = len(self.body) - 1
            for i, expr in enumerate(self.body):
                if i < length:  # если это не последнее
выражение
                    result = execute(expr, env)
                    magic = True
                    if magic and result:
                        return result
                else:
                    if isinstance(env[expr[0].value],
Procedure):
                        proc = env[expr[0].value]
                        self.params = proc.params
                        self.body = proc.body
                        args = [execute(i, env) for i in
expr[1:]]
                        magic = True
                    else:
                        result = execute(expr, env)
                        return result


def testing_work(tree):
    result = subprocess.run(['g++', 'main_1.cpp', '-o',
'main_1'], stderr=subprocess.PIPE)
    if result.returncode == 1:
        raise Exception(result.stderr.decode("utf-8"))
    subprocess.run(['chmod', '+x', 'main_1'])
    result = subprocess.run(['./main_1'],
stdout=subprocess.PIPE)
    print(result.stdout.decode("utf-8"))


def obs(env, fun, *args):
    result = execute(args[0], env)
    for i in args[1:]:
        result = fun(result, execute(i, env))
    return result


def define_function(env, *args):
    name, params, *body = args
    proc = Procedure(params, *body)
```

```
    if not name.value in env:
        env[name.value] = proc
    else:
        msg = 'Function "%s" already exists!' % name.value
        msg += 'in line {}, column {}'.format(name.col,
name.row)
        raise Exception(msg)


def compare(env, *args):
    if execute(args[0], env):
        return execute(args[1], env)
    elif len(args) == 3:
        return execute(args[2], env)


def write(env, *args):
    from sys import stdout
    stdout.write(str(execute(args[0], env)))
    stdout.flush()


def write_line(env, *args):
    from sys import stdout
    stdout.write('%s\n' % str(execute(args[0], env)))
    stdout.flush()


def read_integer(env, *args):
    i = 0
    env[args[i].value] = int(input())

    from sys import stdout
    if isinstance(args[i].value, str):
        stdout.write(str(execute(args[0], env)))
        stdout.flush()


def define_new_variable(env, *args):
    i = 0
    while i < len(args):
        env[args[i].value] = execute(args[i + 1], env)
        i += 2


def execute(expr, env):
    if isinstance(expr, Token):
        if expr.tag == ID and expr.value in env:
            return env[expr.value]
        else:
            return expr.value
    else:
        first, *second = expr
```

```
        if first.value in env and callable(env[first.value]):
            return env[first.value](env, *second)
        else:
            msg = 'Function "%s" not exists!' % first.value
            msg += 'in line {}, column {}'.format(first.col,
first.row)
            raise Exception(msg)


if __name__ == '__main__':
    display_tokens()
    with open("main_1.cpp", "r") as file:
        tree = build_tree(file.read())

    print(tree)
    semantic_analysis(tree)
    testing_work(tree)
```

Приложение 2. Код анализируемой программы на языке C++

```
#include <stdio.h>
#include <iostream>
using namespace std;


void reverseArray(int arr[], int start, int end)
{
    while (start < end)
    {
        int temp = arr[start];
        arr[start] = arr[end];
        arr[end] = temp;
        start++;
        end--;
    }
}


void printArray(int arr[], int size)
{
   for (int i = 0; i < size; i++)
   {
       cout << arr[i];
       cout << " ";
   }
   cout << endl;
}

int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};

    int n = sizeof(arr) / sizeof(arr[0]);

    printArray(arr, n);

    reverseArray(arr, 0, n - 1);

    cout << "Reversed array is";
    cout << endl;

    printArray(arr, n);

    int a = 10;
    int b = 15;
    int t;
    if (a < b)
    {
        t = 10;
    }
    else {
```

```
        t = 15;
    }

    cout << t;

    return 0;
}
```