

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)

Факультет информационных технологий
Кафедра параллельных вычислений

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Курбатова Максима Андреевича

Тема работы:

**СТАТИЧЕСКИЙ АНАЛИЗ ФРАГМЕНТИРОВАННЫХ ПРОГРАММ НА БАЗЕ
АБСТРАКТНОГО СИНТАКСИЧЕСКОГО ДЕРЕВА**

«К защите допущена»
Заведующий кафедрой,
д.т.н., профессор
Мальшкин В.Э. /.....
(ФИО) / (подпись)
«31» мая 2024 г.

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю. /.....
(ФИО) / (подпись)
«31» мая 2024 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ
Киреев С. Е. /.....
(ФИО) / (подпись)
«31» мая 2024 г.

Новосибирск, 2024

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
(НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ, НГУ)
Факультет информационных технологий
Кафедра параллельных вычислений
(название кафедры)

Направление подготовки 09.03.01 Информатика и вычислительная техника
Направленность (профиль): Программная инженерия и компьютерные науки

УТВЕРЖДАЮ
Зав. кафедрой Малышкин В.Э.
(фамилия, И., О.)

.....
(подпись)
«03» ноября 2023 г.

**ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ БАКАЛАВРА**

Студенту Курбатову Максиму Андреевичу, группы 20209

(фамилия, имя, отчество, номер группы)

Тема: Статический анализ фрагментированных программ на базе
абстрактного синтаксического дерева.

(полное название темы выпускной квалификационной работы)

утверждена распоряжением проректора по учебной работе от 03 ноября 2023 №
0413

Срок сдачи студентом готовой работы 20 мая 2024 г.

Исходные данные (или цель работы): проектирование и разработка статического
анализатора LuNA-программ на базе абстрактного синтаксического дерева.

Структурные части работы: ошибки в LuNA-программах, статический анализ,
анализатор AST языка LuNA, тестирование.

Руководитель ВКР
к.т.н.,
доц. каф. ПВ ФИТ НГУ
Власенко А.Ю./.....
«03» ноября 2023 г.

Задание принял к исполнению
Курбатов М.А. /.....
(ФИО) / (подпись)
«03» ноября 2023 г.

Соруководитель ВКР
ст. преп. каф. ПВ ФИТ
Киреев С. Е./.....
«03» ноября 2023 г.

СОДЕРЖАНИЕ

Введение.....	4
Глава 1. Ошибки в LuNA-программах.....	7
1.1 Система LuNA.....	7
1.2 Синтаксические ошибки.....	7
1.3 Семантические ошибки.....	9
1.4 Методы отладки.....	18
Глава 2. Статический анализ.....	23
2.1 Промежуточные представления исходного кода.....	23
2.2 Способы хранения абстрактного синтаксического дерева.....	24
2.3 Способы обхода абстрактного синтаксического дерева.....	26
2.4 Проблемы синтаксических ошибок.....	27
2.5 Методы статического анализа.....	27
2.6 Опробованные статические анализаторы.....	29
Глава 3. Анализатор AST языка LuNA.....	32
3.1 Лексический анализ.....	32
3.2 Синтаксический анализ.....	33
3.3 Восстановление после грамматических ошибок.....	34
3.4 Анализ AST.....	35
3.5 Параллельный анализ ошибок.....	35
3.6 Формат вывода ошибок.....	36
3.7 Визуализатор AST.....	38
3.8 Обнаруживаемые ошибки.....	41
3.9 Комплекс автоматизированной отладки ADAPT.....	47
Глава 4. Тестирование.....	49
4.1 Исходные данные тестирования.....	49
4.2 Анализ результатов тестирования.....	51
Заключение.....	52
Список использованных источников и литературы.....	53
Приложение А.....	56
Приложение Б.....	61
Приложение В.....	66

ВВЕДЕНИЕ

Последовательные программы представляют собой наиболее базовый тип компьютерных программ, в которых команды выполняются одна за другой, в том порядке, в котором они записаны. Параллельное программирование [1] позволяет ускорить решение задач численного моделирования по сравнению с последовательными программами. Однако разработка параллельных программ требует дополнительных навыков, выходящих за рамки обычного программирования. Для этого необходимы специальные знания и опыт в области параллельных вычислений. Основные сложности написания параллельных программ.

1. Сложность отладки.

Из-за недетерминированности выполнения параллельной программы очень сложно повторно воспроизвести проблему. Характерным примером этого являются гонки данных (ситуации, когда несколько потоков/процессов одновременно пытаются получить доступ к общим ресурсам без должной синхронизации), взаимные блокировки (ситуации, когда несколько потоков/процессов ожидают освобождения ресурсов, занятых другими, создавая тупиковую ситуацию, что приводит к полной остановке работы программы).

2. Организация коммуникаций между узлами.

Потоки/процессы должны обмениваться данными между собой, что может приводить к ошибкам и проблемам с эффективностью.

Для упрощения процесса создания параллельных программ существуют системы автоматического конструирования. Такие инструменты позволяют разрабатывать параллельные программы быстрее и без необходимости обладать глубокими познаниями в параллельном программировании.

Система LuNA (Language for Numerical Algorithms) [2] является средством фрагментированного программирования [3], специально ориентированным на решение крупномасштабных задач численного

моделирования и вычислительной математики. Ключевым преимуществом системы LuNA является автоматическое построение параллельной программы, а также автоматическая балансировка вычислительной нагрузки [4]. Благодаря этому, программист может сосредоточиться исключительно на реализации численных алгоритмов, не беспокоясь о сложностях параллельного программирования.

Однако создание программ на языке LuNA сопровождается риском допустить ошибку в исходном коде. Это могут быть как синтаксические ошибки, так и ошибки, связанные с нарушением логики кода.

Статический анализ [5] используется в инструментальных средствах для многих языков программирования с целью уменьшения времени поиска ошибок. На текущий момент таких инструментов для LuNA нет, а компилятор во многих случаях не информирует пользователя об ошибке в доступной форме.

Статический анализ можно проводить как по чистому исходному коду, но более распространенным решением является использование промежуточных представлений кода. Базовым является AST (abstract syntax tree) – конечное ориентированное дерево, узлы которого хранят операции и операнды.

Цель на ВКР:

Проектирование и разработка статического анализатора LuNA-программ на базе абстрактного синтаксического дерева.

Задачи на ВКР:

1. Выявление и анализ ошибок, свойственных фрагментированным программам. Участие в формировании базы типов ошибок LuNA-программ.
2. Обзор методов и инструментальных средств статического анализа.
3. Реализация построения AST для произвольной LuNA-программы.
4. Разработка средства AST-analyzer, реализующего обнаружение ошибок из перечня.
5. Распараллеливание поиска ошибок в средстве AST-analyzer.

6. Оценка времени работы и потребляемых ресурсов статического анализатора на различных LuNA-программах.
7. Интеграция программного средства AST-analyzer в систему автоматизированной отладки ADAPT.

Данный инструмент поможет программистам писать корректный код на LuNA, не тратя на это большого количества времени. Разработчик сможет сосредоточиться над логикой кода, минуя рутинные действия отладки.

Работа содержит четыре главы. В первой главе представлено описание ошибок, которые могут встречаться во фрагментированных программах. Вторая глава посвящена статическому анализу, третья обзору реализованного анализатора, распараллеливанию поиска ошибок и интеграции в ADAPT. Четвертая содержит исходные данные и результаты тестирования.

Глава 1. Ошибки в LuNA-программах

1.1 Система LuNA

LuNA (Language for Numerical Algorithms) – система, разрабатываемая в лаборатории синтеза параллельных программ ИВМиМГ СО РАН. Ее основная цель – это упростить рабочий процесс программиста за счет сокращения трудозатрат на создание параллельных реализаций алгоритмов. Также LuNA имеет свой одноименный язык программирования, основными элементами которого являются:

1. **Фрагмент кода (ФК)** – переиспользуемые участки кода, написанные на LuNA или C/C++. ФК делятся на атомарные и структурированные. Атомарные фрагменты представляют собой подпрограмму (функцию) на языке C++. Структурированные фрагменты вычислений соответствуют управляющим конструкциям языка LuNA (циклы, условные операторы, вызовы подпрограмм)
2. **Фрагмент данных (ФД)** – аналог переменной в языках императивного программирования, но может быть также массивом, структурой и др.
3. **Фрагмент вычисления (ФВ)** – экземпляр ФК, запущенный во время работы программы.

Фрагментированный алгоритм (ФА) – это алгоритм, который разбивает задачу на несколько более мелких частей, которые затем решаются независимо друг от друга.

Язык LuNA позволяет описывать ФА как множество ФВ и ФД, а также использовать рекомендации для контроля над нефункциональными свойствами программы, такими как:

1. сборка мусора;
2. миграция ФВ и ФД на другие вычислительные узлы;
3. порядок выполнения ФВ.

1.2 Синтаксические ошибки

Каждый язык программирования имеет свои правила написания кода, которые задаются грамматикой языка. Важным подмножеством грамматик

являются так называемые контекстно-свободные грамматики (КСГ) [6] - это один из основных классов формальных грамматик, используемых в теории формальных языков. Они обладают следующими ключевыми характеристиками:

Синтаксис:

- Состоят из терминальных и нетерминальных символов.
- Имеют правила вывода в виде $A \rightarrow \alpha$, где A - нетерминальный символ, а α - последовательность терминальных и нетерминальных символов.
- Правила вывода применяются независимо от контекста, то есть замена нетерминального символа A на α может происходить в любом месте строки.

Свойства:

- Распознавание контекстно-свободных языков может быть выполнено за полиномиальное время.
- Контекстно-свободные грамматики обладают хорошими математическими свойствами и широко используются в теории формальных языков, компиляторах, анализе естественных языков и других областях информатики.

Примеры контекстно-свободных языков: арифметические выражения, языки программирования, естественные языки. И LuNA тоже имеет язык со своей контекстно свободной грамматикой (см. приложение А) и накладывает очень жесткие ограничения на написание кода.

Есть множество вариантов написания парсера контекстно-свободного языка. Известные генераторы парсеров:

- flex, bison для C++;
- lingo для flow9;
- peg.js для JavaScript.

Использование уже готового генератора парсера – очень быстрый способ его парсера, но имеются ограничения по гибкости. Например, трудно

продолжать парсинг после грамматической ошибки, а также освобождать память после таковой.

Также можно написать свой парсер на чистом языке программирования. Для этого реализуется рекурсивный алгоритм для парсинга каждого нетерминального символа по правилам вывода. Будет сложное и долгое написание кода, но хорошо настраиваемое поведение парсера в случае грамматической ошибки.

На данный момент компилятор LuNA способен обнаруживать синтаксические ошибки (листинги 1, 2), однако компиляция останавливается после нахождения первой из них. Информационное сообщение содержит название файла, строку в исходном коде и номер символа в ней.

Листинг 1 – Пример синтаксически неправильного объявления многих ФД

```
1 import c_foo() as foo;
2
3 sub main() {
4     df x;
5     df y; // ошибка
6     foo();
7 }
8
```

Листинг 2 – Пример синтаксически неправильной инициализации ФД

```
1
2 sub main() {
3     df x;
4     x = 1; // ошибка
5 }
6
```

1.3 Семантические ошибки

Семантические ошибки есть в каждом языке программирования, они связаны с неправильной логикой программы. Такие ошибки могут допускаться

языком или приводить к аварийному завершению программы во время исполнения или компиляции.

В рамках работы совместно с другими студентами была собрана база типов ошибок LuNA программ. Каждому типу присвоен уникальный идентификатор LuNAxx.

Можно выделить 4 класса семантических ошибок:

Ошибки, связанные с неверным использованием ФД:

1. повторная инициализация ФД (LuNA03);
2. попытка использования неинициализированного ФД (LuNA05);
3. два или более объявлений ФД в подпрограмме (LuNA07);
4. использование ФД после его удаления (LuNA09);
5. неиспользуемое имя (LuNA10);
6. ФД с одинаковыми названиями в одной области видимости (LuNA13);
7. попытка использования необъявленного ФД (LuNA14);
8. циклическая зависимость по данным (LuNA15);
9. использование ФД после его удаления при помощи соответствующего оператора (LuNA29);
10. попытка запросить неинициализированный ФД при помощи request (LuNA27);
11. попытка использования ФД после превышения допустимого числа запросов (LuNA28);
12. попытка индексации объекта, не являющегося ФД (LuNA36);
13. попытка запросить ФД из узла, где его нет (LuNA32);
14. Ошибки, связанные с циклами инициализаций и использований (LuNA18 – LuNA21).

Ошибки, связанные с неверным использованием ФК:

1. несуществующая LuNA-подпрограмма (LuNA02);
2. несовпадение количества аргументов при объявлении ФК и его вызове (LuNA6);

3. импорт нескольких разных ФК под одним алиасом (LuNA11);
4. отсутствие ФК main (LuNA12);
5. повторное определение ФК (LuNA16);
6. импорт несуществующей C/C++ функции (LuNA17);
7. использование оператора информационной зависимости для структурированного ФК (LuNA30).

Ошибки, связанные с неверным использованием типизации:

1. несоответствие типов аргументов при вызове атомарного ФК (LuNA01);
2. несоответствие типов аргументов при вызове структурированного ФК (неверный порядок аргументов) (LuNA04);
3. несоответствие типов LuNA при присваивании (LuNA08);
4. использование нецелых переменных в границах циклов (LuNA39).

Логически некорректное использование конструкций языка LuNA:

1. шаг цикла использования не кратен шагу цикла инициализации (LuNA22);
2. формула в if тождественно истинна/ложна (LuNA23);
3. формула в if истинна/ложна во всех путях выполнения (LuNA24);
4. операторы, возвращающие булево значение, используется в целочисленном контексте $((x > y) - 1)$ (LuNA25);
5. попытка инициализации неподходящего выражения (LuNA26);
6. неверные границы циклов (for i=10..1) (LuNA38);
7. не передать значение для nfragam (данный параметр нужен для распределения вычислений по узлам);
8. неправильный параметр для nfragam;
9. безусловная рекурсия (LuNA34);
10. пересечение диапазонов инициализируемых индексов (LuNA35);
11. нет (гарантированного) цикла инициализации (LuNA37).

Рассмотрим некоторые типы ошибок подробнее.

Несоответствие типов аргументов при вызове атомарного ФК

Ошибка возникает при передаче аргумента ФК не того типа, какой объявлен в сигнатуре. Пример, когда возникает данный тип ошибки представлен в листингах 3-5.

Листинг 3 – Пример LuNA файла для ошибки LuNA01

```
1 import c_print(int) as print;
2
3 sub main(){
4   print("1");
5 }
```

Листинг 4 – Пример .cpp файла для ошибки LuNA01

```
1 #include <cstdio>
2 #include "ucenv.h"
3
4 extern "C" {
5   void c_print(int val) {
6     printf("%d\n", val);
7   }
8 }
```

Листинг 5 – Вывод компилятора для ошибки LuNA01

```
luna: fatal error: cpp-generation failed (see
below):
err> Traceback (most recent call last):
err>
File
"/home/maxwell/luna/scripts/../scripts/fcmp2",
. . .
err>
File
"/home/maxwell/luna/scripts/../scripts/fcmp2", line 328,
in value_int
err>     raise NotImplementedError("Need quotation",
x['value'], c)
```

```
err> NameError: name 'c' is not defined
err> }
```

Несуществующая LuNA-подпрограмма

Возникает при попытке вызвать ФК, который не был объявлен в программе как структурированный и не был импортирован как атомарный. Пример, когда возникает данный тип ошибки представлен в листингах 6-8.

Листинг 6 – Пример LuNA файла для ошибки LuNA02

```
1 import c_print(int) as prints;
2 sub main(){
3     print(1);
4 }
```

Листинг 7 – Пример .cpp файла для ошибки LuNA02

```
1 #include <cstdio>
2 #include "ucenv.h"
3
4 extern "C" {
5     void c_print(int val) {
6         printf("%d\n", val);
7     }
8 }
9
```

Листинг 8 – Вывод компилятора для ошибки LuNA02

```
luna: fatal error: recom-generation failed (see
below):
```

```
Traceback (most recent call last):
  File "/home/maxwell/luna/scripts/../scripts/fcmp",
line 1713, in <module>
    content=Fa(ja).gen()
. . .
    parse_bi(self.Items, bi, self.Regis, self)
```

```
File "/home/maxwell/luna/scripts/../scripts/fcmp",  
line 833, in parse_bi  
    sub=ja[bi['code']]  
KeyError: 'print'
```

Повторная инициализация ФД

LuNA – язык единственного присваивания, вследствие этого при попытке повторной инициализации ФД генерирует ошибка выполнения. Пример, когда возникает данный тип ошибки представлен в листингах 9-11.

Листинг 9 – Пример LuNA файла для ошибки LuNA03

```
1 import c_init(int, name) as init;  
2  
3 sub main(){  
4     df x;  
5     init(1, x[1]);  
6     init(2, x[1]);  
7 }
```

Листинг 10 – Пример .cpp файла для ошибки LuNA03

```
1 #include <cstdio>  
2 #include "ucenv.h"  
3  
4 extern "C" {  
5 void c_init(int val, OutputDF &df){  
6     df.setValue(val);  
7 }  
8 }
```

Листинг 11 – Вывод компилятора для ошибки LuNA03

```
luna: fatal error: run-time error: errcode=-6  
err> 0 ERROR: Duplicate id in post: ID<0, 0, 1>  
./src/rts/rts.cpp:661  
err> 0 ABORT
```

```

err> terminate called after throwing an instance of
'RuntimeError'
err> what(): std::exception
err> [DESKTOP-PEANPOI:264313] *** Process received
signal ***
err> [DESKTOP-PEANPOI:264313] Signal: Aborted (6)
err> [DESKTOP-PEANPOI:264313] Signal code: (-6)
err> [DESKTOP-PEANPOI:264313] [ 0]
/lib/x86_64-linux-gnu/libc.so.6(+0x42520) [0x7f5777060520
]
err> [DESKTOP-PEANPOI:264313] [ 1]
. . .
/lib/x86_64-linux-gnu/libc.so.6(+0x126850) [0x7f57771
44850]
err> [DESKTOP-PEANPOI:264313] *** End of error
message ***
err>

```

Неиспользуемое имя

Происходит при декларации ФД, ФК или параметра ФК, которые не используются в дальнейшем коде. Компилятор не реагирует на этот недочет. Пример, когда возникает данный тип ошибки представлен в листинге 12.

Листинг 12 – Пример LuNA файла для ошибки LuNA10

```

1 import init() as init;
2
3 sub main() {
4   df x;
5   init();
6 }

```

Импорт нескольких разных функций под одним алиасом

При подключении атомарных ФК мы обязаны дать им алиас, чтобы использовать в контексте `.fa` программы. Может оказаться, что разным ФК присвоен один алиас. Компилятор будет использовать последнее определение ФК, игнорируя предшествующие. Пример, когда возникает данный тип ошибки представлен в листинге 13.

Листинг 13 – Пример LuNA файла для ошибки LuNA11

```
1 import c_init() as init;
2 import c_init1() as init;
3
4 sub main() {
5   init();
6 }
```

Отсутствие ФК main

ФК `main` – точка входа в программу, поэтому, если нет такого ФК, то программа не запустится. Пример, когда возникает данный тип ошибки представлен в листингах 14-15.

Листинг 14 – Пример LuNA файла для ошибки LuNA12

```
1 import init() as init;
2
3 sub main1() {
4   init();
5 }
```

Листинг 15 – Вывод компилятора для ошибки LuNA14

```
luna: fatal error: cpp-generation failed (see
below) :
err> Traceback (most recent call last):
err>
File
"/home/maxwell/luna/scripts/./scripts/fcmp2", line
1249, in <module>
err>     gja['main']['name']='main'
```

```
err> KeyError: 'main'  
err>
```

ФД с одинаковыми названиями в одной области видимости

Область видимости – участок кода, в контексте которого можно использовать ФД. Если окажется, что несколько ФД имеют одно название, то потеряется доступ ко всем ФД, которые были объявлены ранее. Компилятор не реагирует на эту проблему. Пример, когда возникает данный тип ошибки представлен в листинге 16.

Листинг 16 – Пример LuNA файла для ошибки LuNA13

```
1 import c_init(name) as init;  
2  
3 sub main(int x) {  
4     df x;  
5     init(x);  
6 }
```

Попытка использования необъявленного ФД

Данная ошибка встречается во многих языках программирования. Возникает, когда используется имя ФД, который не был объявлен в текущей области видимости. Пример, когда возникает данный тип ошибки представлен в листингах 17-18.

Листинг 17 – Пример LuNA файла для ошибки LuNA14

```
1 import c_init(name) as init;  
2  
3 sub main() {  
4     init(x);  
5 }
```

Листинг 18 – Вывод компилятора для ошибки LuNA14

```
luna: fatal error: cpp-generation failed (see  
below):  
err> Traceback (most recent call last):
```

```
err>File"/home/maxwell/luna/scripts/./scripts/fcmp2
", line 1279, in <module>
err>     bid, cpp=gen_struct(sub)
. . .
err>     raise UnresolvedName(name)
err> __main__.UnresolvedName: x
err>
```

Неверные границы циклов

Компилятор не будет выполнять циклы, нижняя граница которых больше или равна верхней. Однако и предупреждения не будет. Пример, когда возникает данный тип ошибки представлен в листинге 19.

Листинг 19 – Вывод компилятора для ошибки LuNA38

```
1 import c_init() as init;
2
3 sub main() {
4     for i=10..1 {
5         init();
6     }
7 }
```

1.4 Методы отладки

Отладка параллельных программ [7] является более сложной задачей по сравнению с последовательными программами. Ниже приведены некоторые основные методы и техники, используемые для отладки параллельных программ:

Пошаговая отладка

Программа выполняется пошагово, с возможностью наблюдения за значениями переменных после выполнения каждого оператора. Часто используются "точки останова" – преднамеренные прерывания выполнения программы, при которых вызывается отладчик. Главное преимущество этого подхода – возможность детального изучения состояния программы в любой момент времени.

Однако диалоговая отладка имеет и недостатки. Один из них - "эффект наблюдателя" (также известный как "эффект Гейзенберга"), когда сам факт наблюдения за поведением параллельной программы может влиять на ее работу, например, приводить к исчезновению ошибок гонок данных.

Примерами программных средств диалоговой отладки являются TotalView или Allinea DDT.

Анализ по трассе

Анализ по трассе (посмертный анализ) заключается в записи подробного журнала (трассы) выполнения параллельной программы. Далее происходит анализ, в результате которого обнаруживаются ошибки.

В отличие от диалоговой отладки, этот подход не влияет на поведение самой программы, так как трассировка происходит независимо от ее выполнения. Преимущества отладки по трассе:

1. Возможность восстановить и проанализировать все события, происходившие в программе, в том числе редкие и сложные ситуации, такие как гонки данных и дедлоки. Это позволяет выявлять ошибки, которые сложно обнаружить при обычном пошаговом (диалоговом) выполнении.

Недостатки:

1. Сбор данных трассы может значительно замедлять выполнение программы и требовать больших объемов памяти.
2. Анализ большой трассы может быть весьма трудоемким и неочевидным процессом.
3. Большая зависимость от данных – анализ осуществляется на одном наборе заготовленных исходных данных, которые не покрывают все возможные ситуации выполнения программы, что приводит к пропуску ошибок.

Примерами инструментов отладки по трассе являются Intel Inspector или Jumshot. Эти средства позволяют не только записывать трассы, но и предоставляют различные средства визуализации (рисунок 1) и анализа собранных данных.

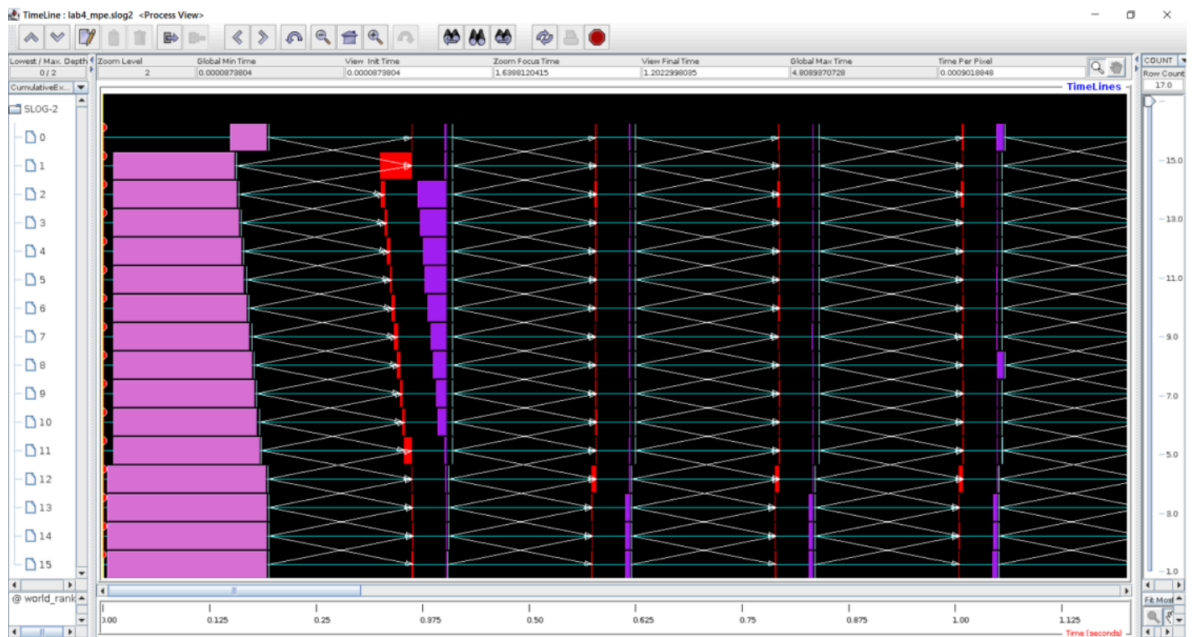


Рисунок 1 – Пример трассировки параллельной программы с помощью Jumshot

Статический анализ

Статический анализ – это метод анализа программы, который заключается в исследовании кода программы до ее запуска. Цель статического анализа – обнаружение ошибок, уязвимостей и других проблем в коде до его запуска. Статический анализ имеет несколько преимуществ:

1. Улучшает качество кода: статический анализ помогает обнаруживать ошибки и уязвимости на ранних стадиях разработки, что уменьшает количество ошибок в готовом продукте.
2. Ускоряет процесс разработки, путем обнаружения ошибок и уязвимостей до запуска программы.
3. Уменьшает затраты на исправление ошибок и уязвимостей, обнаруженных на поздних стадиях разработки.
4. Анализирует весь код целиком, а не определенный вариант исполнения программы.

Однако есть и недостатки:

1. Ложные срабатывания: Инструменты статического анализа выявляют могут определять корректные участки как ошибочные. Такие ложные срабатывания требуют от программиста ручной проверки списка

обнаруженных ошибок и отсеивания тех, которые на самом деле не являются ошибками.

2. Ограниченность анализа: не всегда можно найти ошибку до запуска программы. Например, статический анализ не позволяет обнаруживать ошибки утечек памяти. Для выявления таких ошибок необходимо виртуальное выполнение части программы.

Примеры средств статического анализа будут приведены в следующей главе.

Тестирование

Разработка и выполнение тестовых сценариев, специально нацеленных на проверку параллельного поведения.

1. Юнит тестирование – проверки каждого модуля (юнита) по отдельности без его тестирования в рамках всей системы. Поэтому их обычно пишут сами разработчики ПО.
2. Интеграционное тестирование – проверка нескольких модулей проекта на баги. Таких тестов меньше, чем модульных и их пишут тестировщики.
3. E2E – проверка всей системы в целом. Таких тестов еще меньше, но они еще сложнее чем интеграционные и модульные.

Верификация программ на моделях

Гарантировать качество программных систем можно только с помощью подхода, принципиально отличного от тестирования – верификации.

Верификация программной системы включает в себя три основных этапа:

1. Построение математической модели анализируемой системы, которая представляет собой граф, вершины которого называются состояниями, и изображают ситуации, в которых может находиться эта система в различные моменты времени, а ребра соответствуют переходам, по которым могут происходить изменения состояний в процессе функционирования этой системы.
2. Представление проверяемых свойств системы в виде формальной спецификации, например, в форме логической формулы.

3. Математическое доказательство наличия или отсутствия у системы свойств, описанных в спецификации.

Верификация, как правило, применяется для анализа корректности программных систем, что является ключевым требованием к их качеству. Если корректность нарушена, то эксплуатация системы невозможна, даже если она удовлетворяет всем остальным требованиям.

Глава 2. Статический анализ

2.1 Промежуточные представления исходного кода

Трехадресный код

Трехадресный код (Three-address code, TC) [8] - форма представления исходного кода программы, каждая инструкция которого состоит из трех адресов: Два операнда (источник и приемник) и одного оператора. Например:

Исходный код

```
1 res = (a + b) * (c + d)
```

Трехадресный код

```
1 t1 = a + b
```

```
2 t2 = b + c
```

```
3 t3 = t1 + t4
```

```
4 res = t4
```

Основные преимущества трехадресного кода:

1. Простота генерации и обработки
2. Удобство для оптимизации программ
3. Возможность эффективного распараллеливания вычислений

Абстрактное синтаксическое дерево

Абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) [9] - это конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены с операторами языка программирования, а листья — с соответствующими операндами (Рисунок 2).

Пример:

```
1
```

```
2 init(a);
```

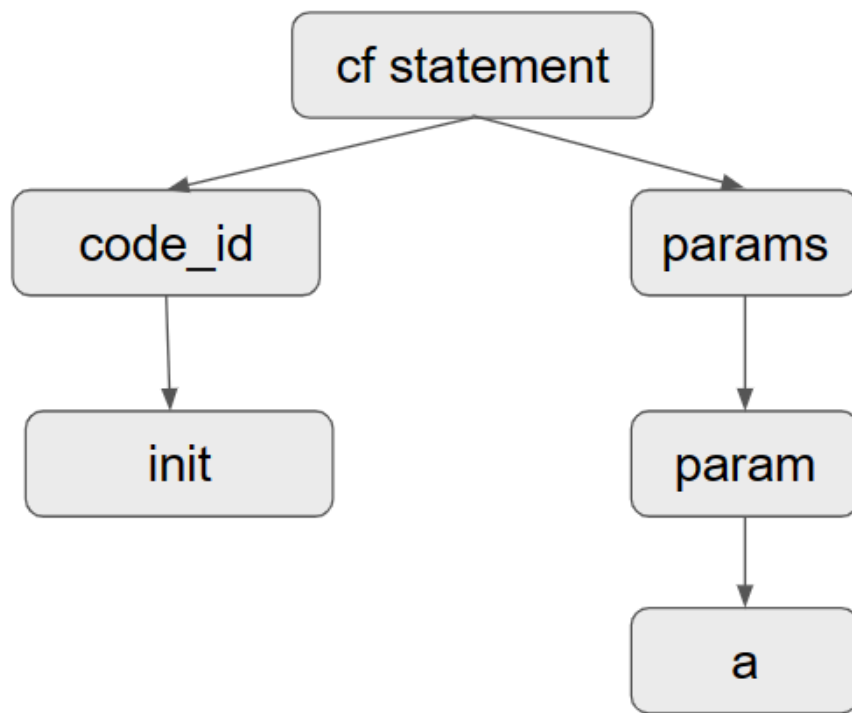


Рисунок 2 – Пример построения AST

Граф потока управления

Граф потока управления (Control Flow Graph, CFG) [10] – это графическое представление структуры управления в программе. Он отображает возможные пути выполнения программы. Может использоваться для нахождения бесконечной рекурсии – цикл в графе. Или обнаружение недостижимого кода – звеньев, без ребер. Важно отметить, что построение графа потока управления осуществляется на основе ранее полученного трехадресного кода.

Граф зависимости по данным

Граф зависимости по данным (Data Dependence Graph, DDG) [11] – это граф, который представляет зависимости между данными в программе. Он отображает, как значения переменных используются и модифицируются в ходе выполнения программы. Может использоваться для нахождения дедлоков или циклических зависимостей по данным.

2.2 Способы хранения абстрактного синтаксического дерева

XML (Extensible Markup Language) или JSON (JavaScript Object Notation)

Форматы для структурирования и представления данных в иерархическом виде. Пример хранения AST в XML формате представлен на рисунке 3.

<pre>1 public class C { 2 C field; 3 int x; 4 5 void foo() { 6 int y; 7 ... 9 y = field.x; 10 ... 15 } 16 }</pre>	<pre><BlockStatement allocation="false"> <Statement> <StatementExpression> <PrimaryExpression> <PrimaryPrefix> <Name image="y"/> </PrimaryPrefix> </PrimaryExpression> <AssignmentOperator image="="/> <Expression "> <PrimaryExpression> <PrimaryPrefix> <Name image="field.x"/> </PrimaryPrefix> </PrimaryExpression> </Expression> </StatementExpression> </Statement> </BlockStatement></pre>
---	---

Рисунок 3 – Представление AST в xml в статическом анализаторе PMD

Такие форматы подходят для небольших деревьев, потому что, хранения AST в таком виде занимает довольно много места в памяти процесса.

Реляционное представление

Хранение AST в форме отношений между таблицами в реляционной базе данных [12]. Каждая таблица содержит множество узлов одного типа, которые имеют уникальный идентификатор. Пример хранения AST в реляционной форме представлен на рисунке 4.

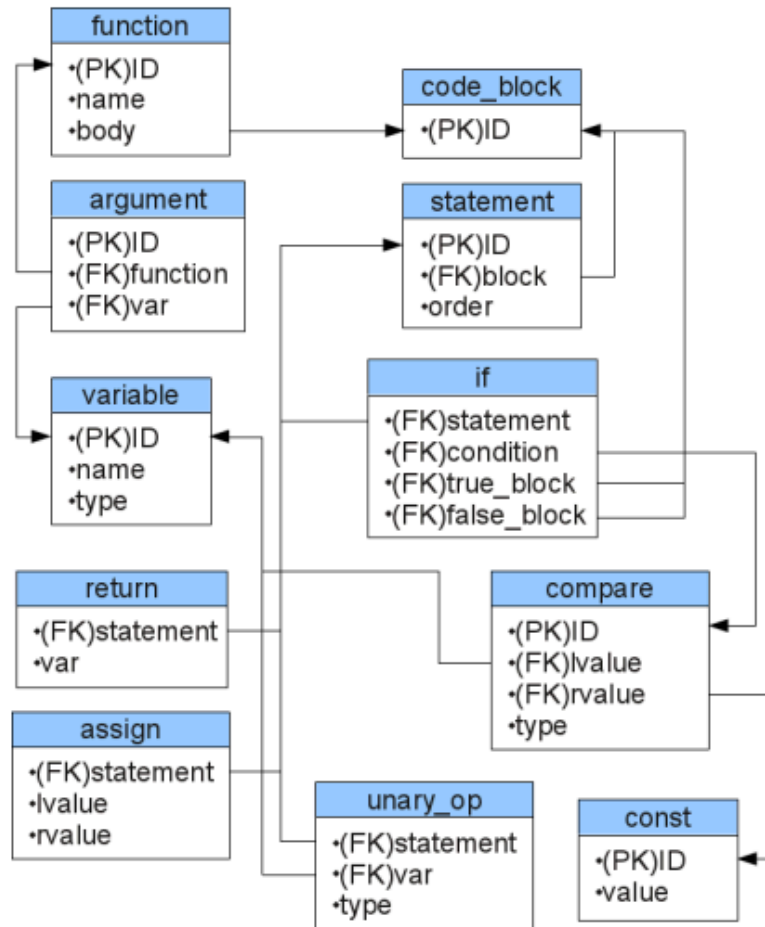


Рисунок 4 – Представление AST в реляционном формате

Такое представление очень полезно, если абстрактное синтаксическое дерево настолько большое, что не помещается в оперативную память, но при этом обращения к его узлам осуществляются довольно редко.

2.3 Способы обхода абстрактного синтаксического дерева

Существует два основных вида обходов [13]:

1. Обход в глубину

Принцип этого подхода заключается в том, что для любого узла вначале совершается полный обход всех его поддеревьев. Получается, что происходит глубокий спуск по одной ветки дерева, прежде чем перейти к другой. У листьев поддеревьев нет, поэтому на них алгоритм обхода заканчивается. Этот подход эффективен по памяти, т.к. хранится только текущая ветвь обхода.

2. Обход в ширину

Принцип состоит в том, что посещаются все узлы на одном уровне, прежде чем перейти на следующий. Реализуется с помощью очереди, в которую кладутся все узлы текущего уровня дерева. Этот подход неэффективен по памяти, т.к. может потребоваться хранить много узлов в очереди для текущего уровня. Отсюда вытекает, что он неэффективен для глубоких деревьев, так как нужно посетить все узлы на каждом уровне.

2.4 Проблемы синтаксических ошибок

В процессе написания программы пользователь может совершать грамматические ошибки. Это приводит к тому, что синтаксический анализатор не может распарсить исходный код. Это приводит к следующим проблемам:

1. **Выводится сообщение только об одной ошибке.** Приходится аварийно завершать процесс парсинга исходного кода. Выводить сообщение о найденной синтаксической ошибке, очищать ресурсы и завершать работающий процесс.
2. **Абстрактное синтаксическое дерево не построится.** В силу того, что непонятно, как продолжать анализировать токены, приходится останавливать работу синтаксического анализатора. AST может оказаться некорректным, так нужных узлов может не быть, чтобы каким-то образом производить семантический анализ.
3. **Необходимость вновь запускать анализ после исправления ошибки.** На основе единственного сообщения об ошибке программисту нужно отладить код. Далее вновь запустить анализатор. Это неоптимально, с точки зрения затрат временных ресурсов разработчика.

2.5 Методы статического анализа

Сопоставление с шаблоном

Этот метод [14, 15] основан на сравнении характеристик исследуемого программного обеспечения с заранее подготовленными описаниями потенциальных уязвимостей. Исходный код сравнивается с типовыми ситуациями, описанными в базе шаблонов. Если обнаруживаются совпадения, то сообщается о найденной уязвимости. Файлы шаблонов представляют собой

текстовые описания проблемных ситуаций, которые могут быть распознаны статическим анализатором.

Анализ потока данных

Этот метод [16] направлен на сбор информации о возможных значениях переменных в различных точках программы. Для этого используется граф потока управления.

Концепция достигающих определений является одной из широко используемых схем анализа потока данных. Она позволяет определить, где именно в программе может быть задано значение каждой переменной x при достижении конкретной точки p в ходе выполнения программы. Эта информация, в свою очередь, дает возможность установить, является ли переменная x константой в точке p , а также выявить случаи использования неинициализированной переменной x в точке p .

Вывод типов

Механизм, который позволяет автоматически определять типы выражений и переменных без явного указания типов. Вывод типов основан на анализе контекста использования переменных и выражений, а также на правилах типизации, определенных в языке. Это позволяет компилятору или интерпретатору вывести наиболее подходящие типы для объектов в коде.

Измерение программных метрик

Программные метрики представляют собой численные характеристики различных свойств исходного кода. Существует множество разнообразных метрик, которые могут быть вычислены с помощью различных инструментов.

Одной из наиболее распространенных метрик является количество строк кода. Изначально эта метрика использовалась для оценки трудозатрат на проект. Тем не менее, с появлением современных языков программирования, в которых в одной строке может быть записано несколько команд, применимость данной метрики стала ограниченной. Для преодоления этого ограничения, различают два типа строк кода:

1. Физические строки кода – это фактическое количество строк в исходном коде программы.
2. Логические строки кода – это количество отдельных команд или инструкций в программе.

Использование программных метрик позволяет оценивать различные аспекты программного обеспечения, такие как сложность, производительность, надежность и так далее.

2.6 Опробованные статические анализаторы

Cppcheck

CppCheck [17] – Статический анализатор кода для языков C и C++. Можно запускать через терминал или через графический интерфейс. Пример вывода анализатора cppCheck представлен на рисунке 5.

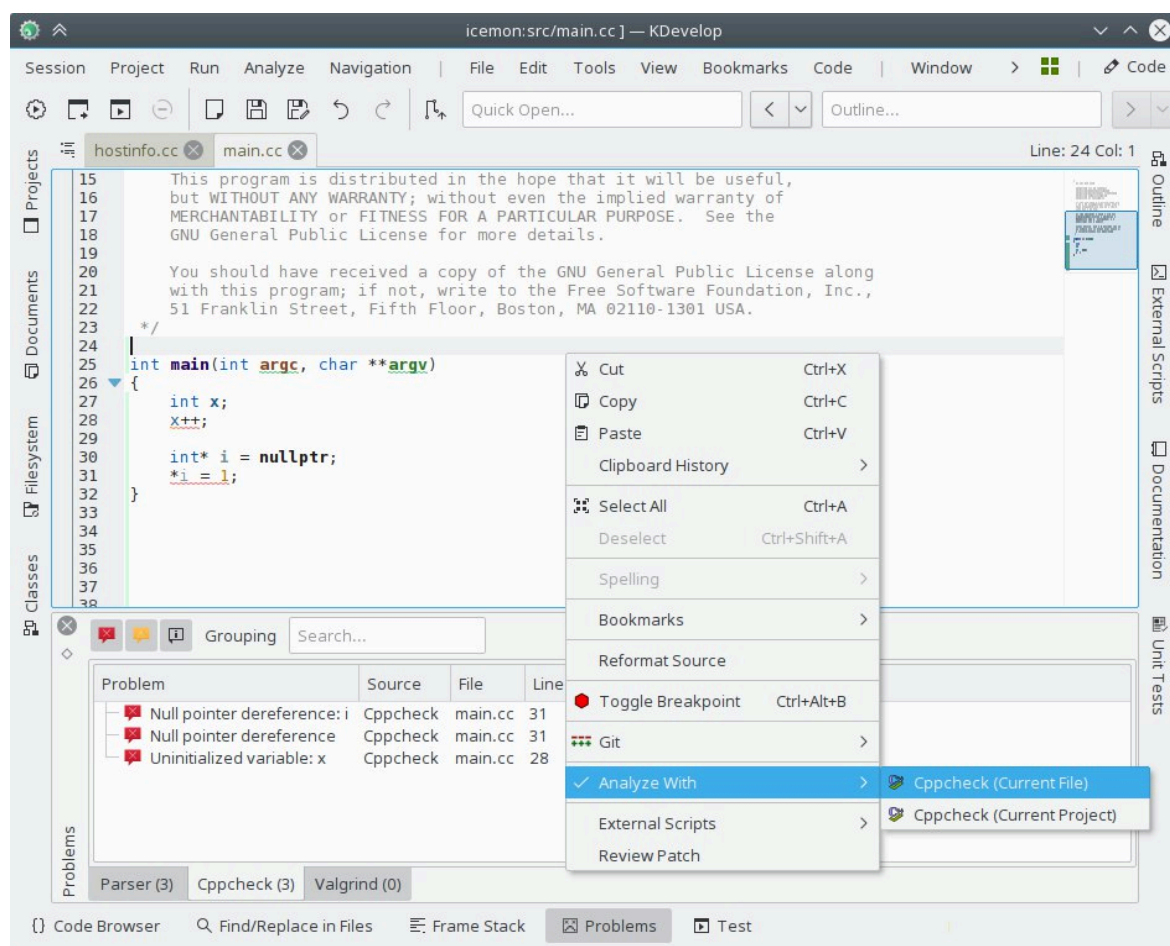


Рисунок 5 – Пример вывода Cppcheck

Основные классы ошибок cppCheck

1. `error` — явные ошибки, которые анализатор считает критическими и обычно они приводят к багам (включено по умолчанию);
2. `warning` — предупреждения, т.е. потенциальные ошибки, которые могут возникнуть, но не приведут к серьезным проблемам;
3. `style` — стилистические ошибки, сообщения появляются в случае неаккуратного кодирования;
4. `missingInclude` — проверка на недостающий `#include`.

PVS-Studio

PVS-Studio [18] – это коммерческий статический анализатор кода, написанный на C, C++, C#, Java. Он предназначен для поиска, анализа и исправления ошибок в исходном коде. Работает на платформах Windows, Linux и Mac OS.

На Windows есть возможность интегрировать анализатор в Microsoft Visual Studio, JetBrains Rider или Visual Studio Code в качестве плагина. PVS-Studio для Linux представляет собой консольное приложение.

В PVS-Studio присутствуют 3 уровня предупреждений (Рисунок 6):

1. Наиболее подозрительные места, приводящие к существенным проблемам.
2. Не критичная, но важная неточность в коде.
3. Несущественная неточность в коде или предупреждения, которые с большой вероятностью являются ложными срабатываниями

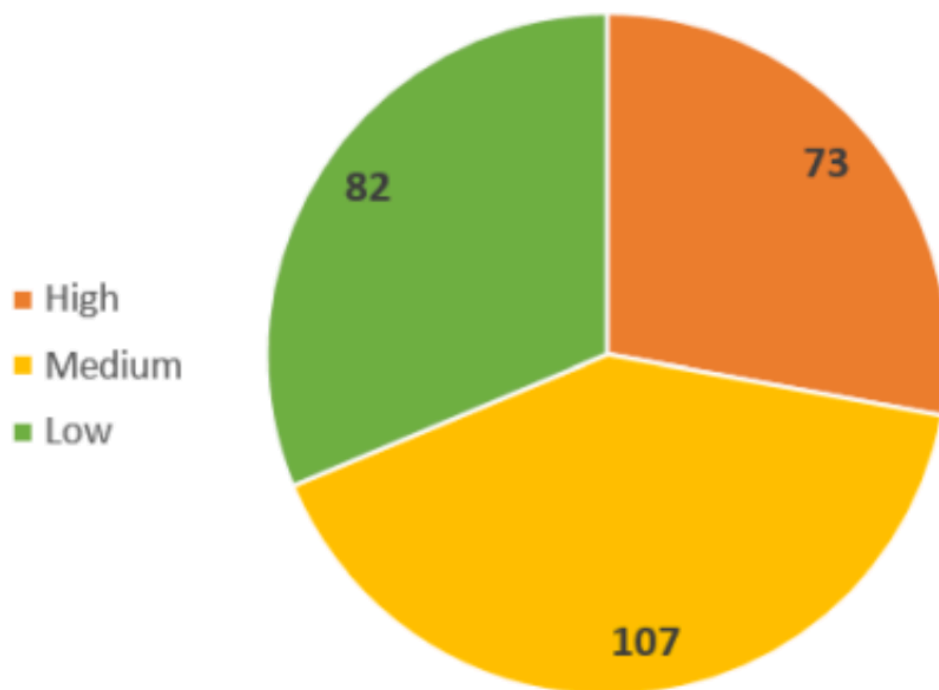


Рисунок 6 – Распределение количества ошибок по уровням
Есть возможность фильтровать сообщения по уровням ошибок.
Список ошибок, обнаруживаемых с помощью PVS-Studio [19]:

1. недостижимый код;
2. ошибки при использовании указателей;
3. синтаксические ошибки;
4. неиспользуемые аргументы;
5. бессмысленные выражения;
6. и др.

Глава 3. Анализатор AST языка LuNA

3.1 Лексический анализ

Лексический анализ – первый этап построения абстрактного синтаксического дерева. Он заключается в преобразовании последовательности символов – исходного кода в последовательность токенов. Токен – это последовательность символов, имеющая логический смысл. Каждый токен имеет свой смысл и определяется своим регулярным выражением.

Для лексического анализа был выбран инструмент Flex [20, 21], тк он используется в компиляторе языка LuNA, что дает возможность создавать анализатор на основе уже работающего. Flex доступен в официальном репозитории Ubuntu, также есть его аналог для Windows. Важно отметить, что Flex генератор лексических анализаторов, он не выполняет анализ. Общая структура Bison программ представлена в листинге 20.

Листинг 20 – Структура .lex программ

```
definitions
%%
rules
%%
user code
```

Definitions:

Раздел, который содержит

- Директивы препроцессора, такие как #include и #define.
- Объявления типов данных, используемых в семантических действиях.
- Объявления символов (терминальных и нетерминальных).
- Приоритеты и ассоциативность операторов.

Rules:

Раздел, который содержит ряд правил вида:

```
шаблон действие
```

, где шаблон не должен иметь отступов, а действие должно начинаться в той же строке.

User code:

Раздел, который содержит пользовательский код на языке C/C++, который будет использоваться в семантических действиях. Этот код может включать в себя объявления функций, переменных и другие определения, необходимые для обработки синтаксического анализа.

По умолчанию результатом работы Flex является файл lex.cc, который содержит сгенерированный код лексического анализатора на C/C++. Исходный код лексического анализатора языка LuNA представлен в приложении А.

3.2 Синтаксический анализ

Синтаксический анализ идет после лексического. Он заключается в анализе последовательности токенов по правилам. Для данного процесса используется утилита Bison [20, 22], которая также была использована в компиляторе. Общая структура Bison программ представлена в листинге 21.

Листинг 21 – Структура .ypp программ

```
definitions
%%
grammar rules
%%
user code
```

Разделы Definitions и User code совпадают по смыслу с .lex файлами, отличается только раздел второй раздел.

Grammar rules:

1. Определение правил грамматики, описывающих синтаксис языка.
2. Каждое правило состоит из левой части (нетерминальный символ) и правой части (последовательность терминальных и нетерминальных символов).
3. Каждое правило может иметь ассоциированные семантические действия, которые выполняются при применении этого правил.

Исходный код синтаксического анализатора AST-analyzer представлен в Приложении Б.

Алгоритм работы синтаксического анализатора

После каждого обнаруженного токена Flex кладет его в стек, дальше происходит поиск возможного правила Bison. Если такое правило существует, то токены, участвующие в нем снимаются со стека и осуществляется пользовательское действие. В нашем случае – это создание узла AST. Однако бывают ситуации, когда Bison не может однозначно определить, какое правило должно быть применено в конкретной ситуации. Существует два основных типа конфликтов:

1. Shift-Reduce conflicts (сдвиг-свертка): Возникает, когда Bison не может однозначно решить, нужно ли выполнить сдвиг токена или свернуть правило грамматики.
2. Reduce-Reduce conflicts (свертка-свертка): Возникает, когда Bison может применить более одного правила для сворачивания.

Такие конфликты разрешаются путем определения приоритетов и ассоциативности операций в секции Definitions.

По окончании работы Bison получается готовое абстрактное синтаксическое дерево. В нашем случае - это указатель на корень дерева.

3.3 Восстановление после грамматических ошибок

Bison предоставляет возможность восстановления после грамматической ошибки. Можно определять способ восстановления после синтаксической ошибки, создавая правила, которые распознают специальный терминальный символ *error*. Этот символ всегда определен и зарезервирован внутри Bison для обработки ошибок. Синтаксический анализатор генерирует лексему *error* каждый раз, когда обнаруживает синтаксическую ошибку. А если предусмотрено и написано специальное правило для распознавания этой лексемы в текущем контексте, то разбор может быть продолжен. Таким образом решаются следующие проблемы

1. Парсинг продолжается в случае грамматической ошибки.

2. Есть возможность находить много синтаксических ошибок. Это очень удобно для программиста, потому что, можно сразу править код в нескольких местах.
3. Абстрактное синтаксическое дерево будет построено всегда. Однако поддерево, соответствующее некорректному кусочку кода не будет включено в AST.

3.4 Анализ AST

Учитывая все недостатки и преимущества обоих видов обходов деревьев, для AST-analyzer был выбран обход в глубину.

AST-analyzer реализует классический подход анализа AST – обход в глубину с сохранением в контейнер всех узлов, нужных для обнаружения определенной ошибки. После этого осуществляется последующий анализ этих элементов, итерация по которым уже не представляет трудностей.

- Для анализа конкретной ошибки нужно ответить на следующие вопросы:
- Какие данные нам нужны для анализа данной ошибки?
- Какие узлы нам нужны для анализа этой ошибки?
- Нужны ли данные других узлов, для анализа данных текущего узла?
- Может ли быть, что не хватает данных для точного предсказания ошибки?

3.5 Параллельный анализ ошибок

Очень важно заметить из алгоритма анализа AST, что ни одна проверка ошибки не изменяет исходное дерево. Это дает возможность для параллельного поиска ошибок, вследствие чего увеличивается скорость работы анализатора.

AST-analyzer реализует параллельный анализ путем использования пула-поток [23] (Рисунок 7). Все проверки находятся в очереди, а простаивающие потоки забирают оставшиеся задачи анализа, пока очередь не опустеет.

Разумно распределить ошибки по тому, как стоит обходить AST. Нет смысла делать одинаковые обходы несколько раз для разных ошибок. На данный момент поиск ошибок разбит на 7 параллельных обходов AST:

1. LuNA13

2. LuNA34, LuNA4, LuNA2, LuNA6
3. LuNA11
4. LuNA10
5. LuNA14
6. LuNA16
7. LuNA12

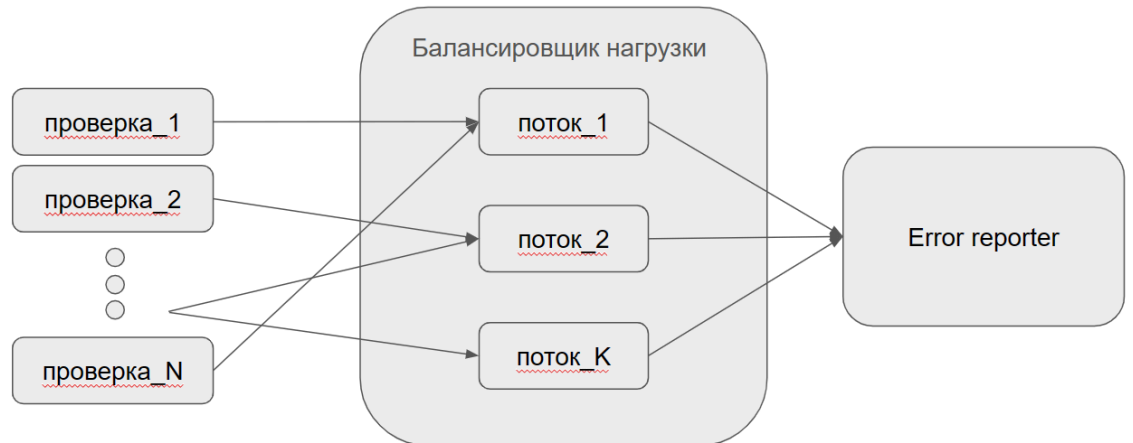


Рисунок 7 – Схема параллельного анализа внутри AST-analyzer

3.6 Формат вывода ошибок

Совместно с другими студентами кафедры был разработан общий формат вывода ошибок. Основная идея – это определить информацию о программных сущностях, которые вызвали ошибку. AST-analyzer использует следующие из них:

Элемент стека вызовов (call stack entry)

- file: строка - имя файла;
- line: целое число - номер строки;
- name: строка - имя элемента (для циклов - for, while; для вызовов процедур - имя процедуры).

Пример:

```
{
  "file": "main.fa",
  "line": 42,
  "name": "set_int"
```

```
}
```

ФД (df)

- name: строка - имя при объявлении;
- declared: список call_stack - места объявления;
- [опционально] initialized: список call_stack - места инициализации;
- [опционально] used: список call_stack - места использования.

Пример:

```
{  
  "name": "x",  
  "declared": [[  
    {  
      "file": "main.fa",  
      "line": 10,  
      "name": "main"  
    }  
  ]],  
  "initialized": [  
    [  
      {..., "line": 11}  
    ],  
    [  
      {..., "line": 13}  
    ],  
  ]  
}
```

ФК (sub)

- name: строка - имя;
- type: "struct" | "extern" - тип;
- file: строка - имя файла;
- line: целое число - строка объявления (строка импортирования).

Пример:

```
{
  "name": "set_int",
  "type": "extern",
  "file": "main.fa",
  "line": 3
}
```

Тогда можно описать ошибку в следующем формате

```
{
  "error_code": "номер ошибки",
  "details" : {
    "call_stack_entry": <call_stack_entry>,
    "cf": <cf>
      "df": <df>
      ...
  }
}
```

3.7 Визуализатор AST

Для каждого узла AST написан метод `to_json()`, который предоставляет возможность сериализации его в JSON. Следовательно, можно сериализовывать всё абстрактное синтаксическое дерево. Пример исходников для визуализации представлен в листингах 22, 23. AST в графическом виде представлено на рисунке 8.

Листинг 22 – Пример исходного кода для визуализации AST

```
1  sub main() {
2      df x;
3      cf a : init();
4      hello_world();
5  }
```

Листинг 23 – Json представление исходного кода

```

{
  "program": [
    {
      "sub": {
        "code_id": {"value": "main"},
        "opt_params": {},
        "block": {
          "dfdecls": {
            "name_seq": {
              "name_seq": [
                {
                  "name": { "value": "x" }
                }
              ]
            }
          },
          },
        },
      "statement_seq": [
        {
          "cf_statement": {
            "code_id": { "value": "init" },
            "opt_label": {

```

```
        "simple_id": {"value": "a" }
    },
    "opt_exprs": {},
    "opt_setdf_rules": {},
    "opt_rules": {},
    "opt_behavior": {}
}
},
{
    "cf_statement": {
        "code_id": {
            "value": "hello_world"
        },
        },
        "opt_label": {},
        "opt_exprs": {},
        "opt_setdf_rules": {},
        "opt_rules": {},
        "opt_behavior": {}
    }
}
```

```

],
"opt_behavior": {}
}
}
}
]
}

```

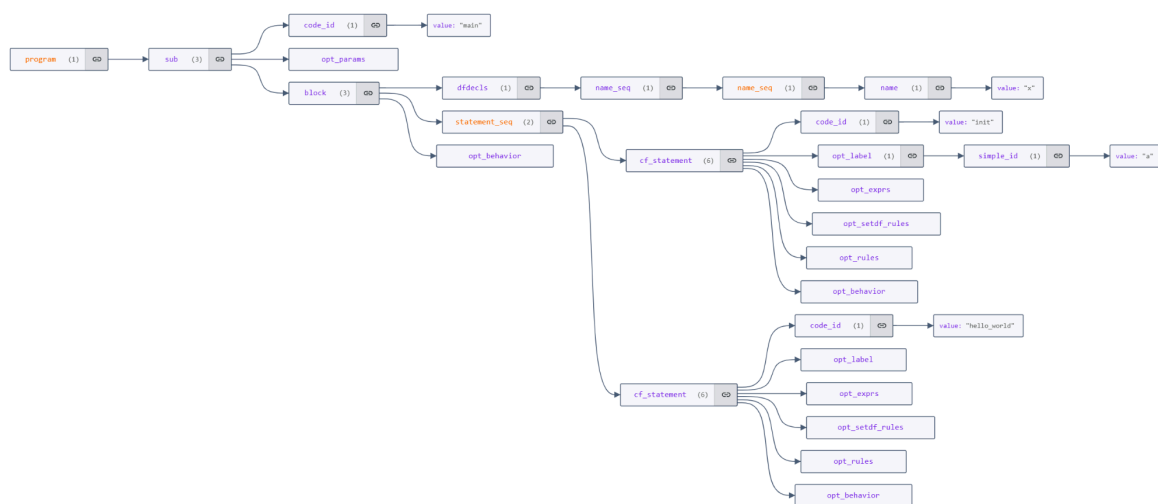


Рисунок 8 – Визуализация AST с помощью JSON Crack

3.8 Обнаруживаемые ошибки

1. Несоответствие типов аргументов при вызове атомарного ФК

LuNA поддерживает два вида преобразования типов:

1. Неявные преобразования. Происходят при передаче во ФК данных, не соответствующих сигнатуре ФК.
2. Явные преобразования (приведения). Способ явно указать, что необходимо выполнить преобразование и что вам известно, что может произойти потеря данных или приведение может завершиться сбоем во время выполнения.

Неявные преобразования **допускают**:

1. преобразование целых чисел и чисел с плавающей точкой друг к другу;
2. преобразование `int`, `real` к `string`;
3. преобразование ФД к примитивному типу, содержащемуся внутри ФД.
Или к примитивному типу, который может быть получен преобразованием примитивного типа, содержащемуся внутри ФД.
Например: `df(int)` можно привести к `real`, тк допустимо преобразование из `int` в `real`.

Неявные преобразование **не допускают**:

1. преобразование `int`, `real`, `string` к ФД;
2. преобразование строкового типа к любому другому типу;
3. преобразование неинициализированного ФД к типу `value`.

Явные преобразования типов (приведение типов) могут быть представленными одним из трех операторов:

1. `int(...)`;
2. `real(...)`;
3. `string(...)`.

Явные приведение типов **допускают** те же самые преобразования, что и неявные.

Логического типа как такового в языке нет, вместо него используется целочисленный тип. При этом считается, что нулевое значение соответствует лжи, а любое ненулевое значение — истине. Это соответствует логическим значениям языка C.

Вывод анализатора состоит из:

1. номер ошибки – LuNA04;
2. сообщение об ошибке – Call arguments do not match declared arguments;
3. места вызова ФК;
4. информация о сигнатуре ФК.

Пример исходного кода и сообщения AST-analyzer об этом типе ошибки представлены в листингах 24-25.

Листинг 24 – Пример программы с ошибкой LuNA04

```
1 import init(name) as init;
2
3 sub main() {
4     init(1);
5 }
```

Листинг 25 – Вывод AST-analyzer для программы с ошибкой LuNA04

```
Found 1 errors:
(1) error[LUNA04]: Call arguments do not match
declared arguments.
- Called at:
File
"/home/maxwell/adapt/ast_analyzer/tests/test_new_error_m
sg_format/luna4.fa", line 4, in
    init(1)
- Function information:
Name:      init,      type:      extern,      file
"/home/maxwell/adapt/ast_analyzer/tests/test_new_error_m
sg_format/luna4.fa", line 1
```

2. Несуществующая LuNA-подпрограмма

Может получиться, что программист захочет вынести часть логики кода в отдельный ФК, чтобы переиспользовать его во многих частях программы. Однако могут возникнуть ошибки использования ФК, которых не существует в текущем контексте. Вывод анализатора состоит из:

1. номер ошибки – LuNA02;
2. сообщение об ошибке – No function with name init exists;
3. места вызова ФК.

Пример исходного кода и сообщения AST-analyzer об этом типе ошибки представлены в листингах 26-27.

Листинг 26 – Пример программы с ошибкой LuNA02

```
1 sub main() {
2   init();
3 }
```

Листинг 27 – Вывод AST-analyzer для программы с ошибкой LuNA02

```
Found 1 errors:
(1)  error[LUNA02]: No function with name init
exists.
- Called at:
                                           File
"/home/maxwell/adapt/ast_analyzer/tests/test_new_error_m
sg_format/luna2.fa", line 2, in
    init()
```

3. Несоответствие типов аргументов при вызове структурированного ФК.

Данный тип ошибки очень похож на LuNA04. Однако анализ производится уже для структурированных ФК. Пример исходного кода и сообщения AST-analyzer об ошибке представлены в листингах 28, 29.

Листинг 28 – Пример программы с ошибкой LuNA06

```
1 import c_i() as i;
2
3 sub init() {
4   i();
5 }
6
7 sub main() {
8   init(1);
9 }
```

Листинг 29 – Вывод AST-analyzer для программы с ошибкой LuNA06

Found 1 errors:

(1) **error**[LUNA06]: Wrong number of arguments when calling the function `init`.

- Called at:

Name: `init`, type: `extern`, file
"/home/maxwell/adapt/ast_analyzer/tests/test_new_error_m
sg_format/luna6.fa", line 3

- Function information:

Name: `init`, type: `extern`, file
"/home/maxwell/adapt/ast_analyzer/tests/test_new_error_m
sg_format/luna6.fa", line 3

4. Несовпадение количества аргументов при объявлении ФК и его вызове.

Этот тип ошибки встречается и во многих других языках программирования. Пример исходного кода и сообщения AST-analyzer об ошибке представлены в листингах 30, 31.

Листинг 30 – Пример программы с ошибкой LuNA04

```
1 import init() as init;
2
3 sub main() {
4   df x;
5   init();
6 }
```

Листинг 31 – Вывод AST-analyzer для программы с ошибкой LuNA04

Found 1 errors:

(1) **error**[LUNA10]: Unused DF `x`.

Declared:

File
"/home/maxwell/adapt/ast_analyzer/tests/test_new_error_m
sg_format/luna10.fa", line 4, in
`df x`

5. Два или более объявлений ФК в подпрограмме.

Язык LuNA не поддерживает механизма перегрузки ФК, однако объявлять ФК с одним названием вполне допускается. Будет использовано последнее определение. Компилятор никак не отслеживает эту ошибку.

Вывод анализатора состоит из:

5. номер ошибки – LuNA16;
6. сообщение об ошибке – Duplicate CF declaration of function main;
7. все повторяющиеся места деклараций ФК в формате:
 - a. название файла;
 - b. номер строки;
 - c. название ФК;
 - d. тип ФК (структурированный или атомарный).

Пример исходного кода и сообщения AST-analyzer об этом типе ошибки представлены в листингах 32, 33.

Листинг 32 – Пример программы с ошибкой LuNA16

```
1 import init() as init;
2
3 sub main() {
4   init();
5 }
6
7 sub main() {
8   init();
9 }
```

Листинг 33 – Вывод AST-analyzer для программы с ошибкой LuNA16

```
Found 1 errors:
(1) error[LUNA16]: Duplicate CF declaration of
function main.
- Duplicate functions information:
Name:      main,      type:      struct,      file
```

```
"/home/maxwell/adapt/ast_analyzer/tests/test_new_error_m  
sg_format/luna16.fa", line 7
```

3.9 Комплекс автоматизированной отладки ADAPT

Совместно с другими студентами был создан комплект автоматизированной отладки LuNA программ ADAPT. Схема его работы представлена на рисунке 9. Он состоит из:

1. AST-analyzer

Анализатор на основе AST, его задача построить дерево, проанализировать все типы ошибки, которые он способен обнаружить, а далее передать контроль следующему анализатору.

2. DeGSA

Анализатор на основе графа зависимости по данным. Первый этап его работы – это построение графа на основе абстрактного синтаксического дерева. Вершины графа соответствуют разным операторам в языке. Например WhileVertex – вершина, представляющая оператор while. Содержит указатели на начальное значение итератора (Expression), условие (Expression), итератор (WhileIteratorName) и выходной ФД (Identifier).

3. prolog-analyzer

В данном средстве статического анализа LuNA-программа представляется в виде набора фактов на языке логического программирования Prolog, а для обнаружения ошибок применяется набор правил. Такой подход позволяет описывать в декларативном стиле то, как ошибки выглядят в терминах анализируемых абстракций, абстрагируясь от процесса их обнаружения.

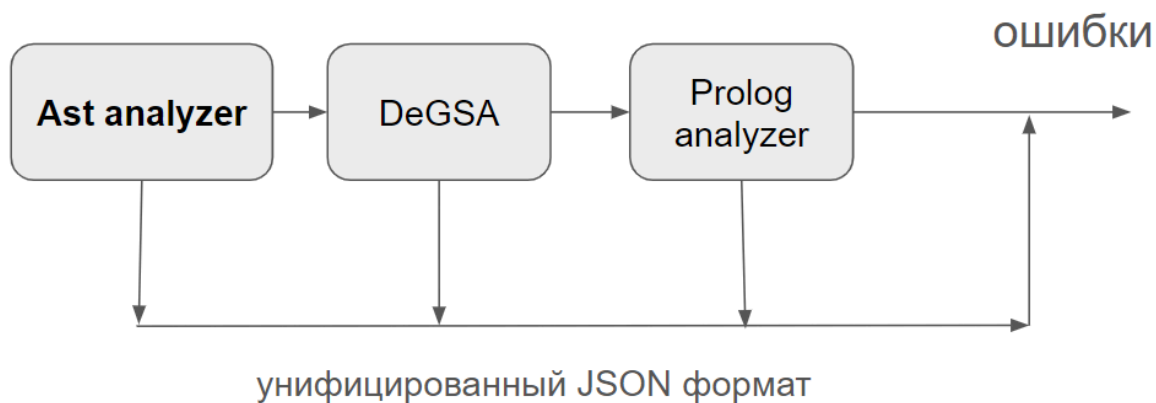


Рисунок 9 – схема работы ADAPT

AST-analyzer и DeGSA способны работать параллельно, используя общий пулл потоков, а Prolog-analyzer запускается последовательно после того, а отработали предшествующие анализаторы. Все анализаторы используют унифицированный формат вывода ошибок, а общий модуль error-reporter выводит сообщение об обнаруженных ошибках в удобном для пользователя формате.

Руководство оператора по применению данной системы приведено в приложении С.

Глава 4. Тестирование

4.1 Исходные данные тестирования

Для замеров были взяты следующие программы из репозитория LuNA: задача решения СЛАУ методом прогонки [24] и задача о движении протопланетного диска методом PIC [25]. Все программные метрики отражены в таблице 1.

Таблица 1 – Метрики исходных файлов для теста

	Метод прогонки	Протопланетный диск
Исходный код (Мб)	0,0214	0,5930
Исходный код(строки)	471	10401
Всего токенов	11936	398969
Токенов в AST	3799	131236

Тестирование проводилось по следующему критериям:

1. время парсинга;
2. время анализа.

Тестирование было проведено на машине с процессором Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.70 GHz, с 4 ядрами и 4 Гб оперативной памяти. Компилятор g++ с ключом оптимизации -O2. Результаты представлены на рисунках 10 - 12.

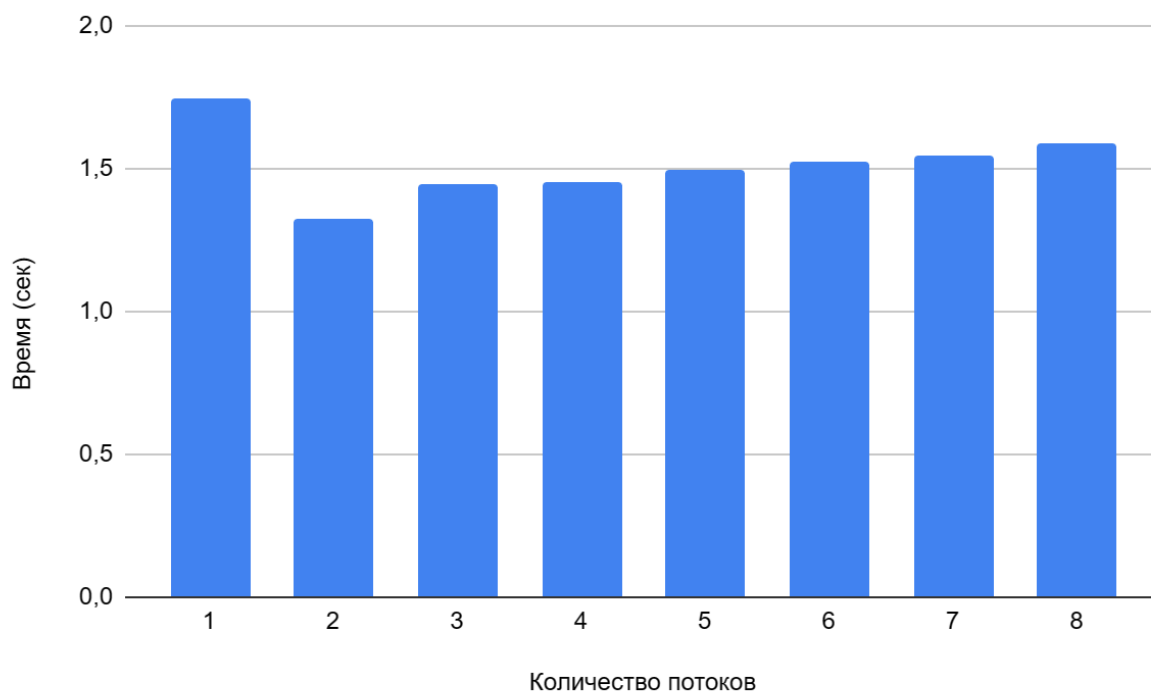


Рисунок 10 – Время работы анализатора в зависимости от количества потоков анализа для метода Прогонки решения СЛАУ

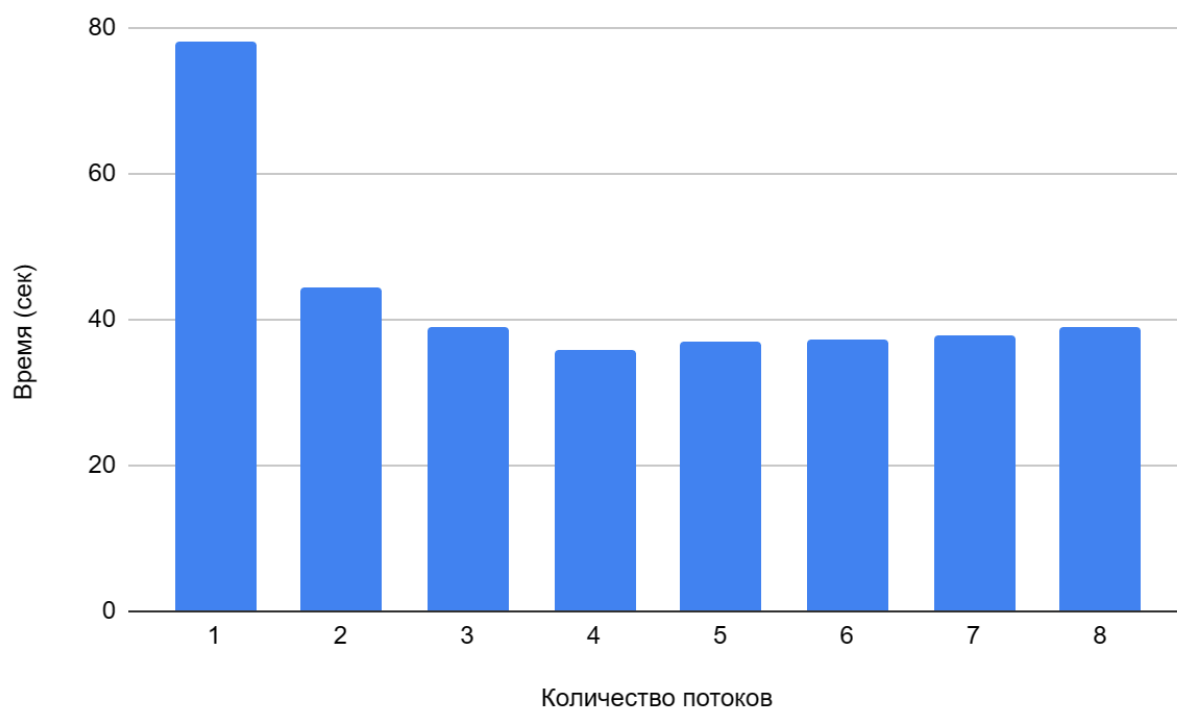


Рисунок 11 – Время работы анализатора в зависимости от количества потоков анализа для решения задачи о движении протопланетного диска методом PIC

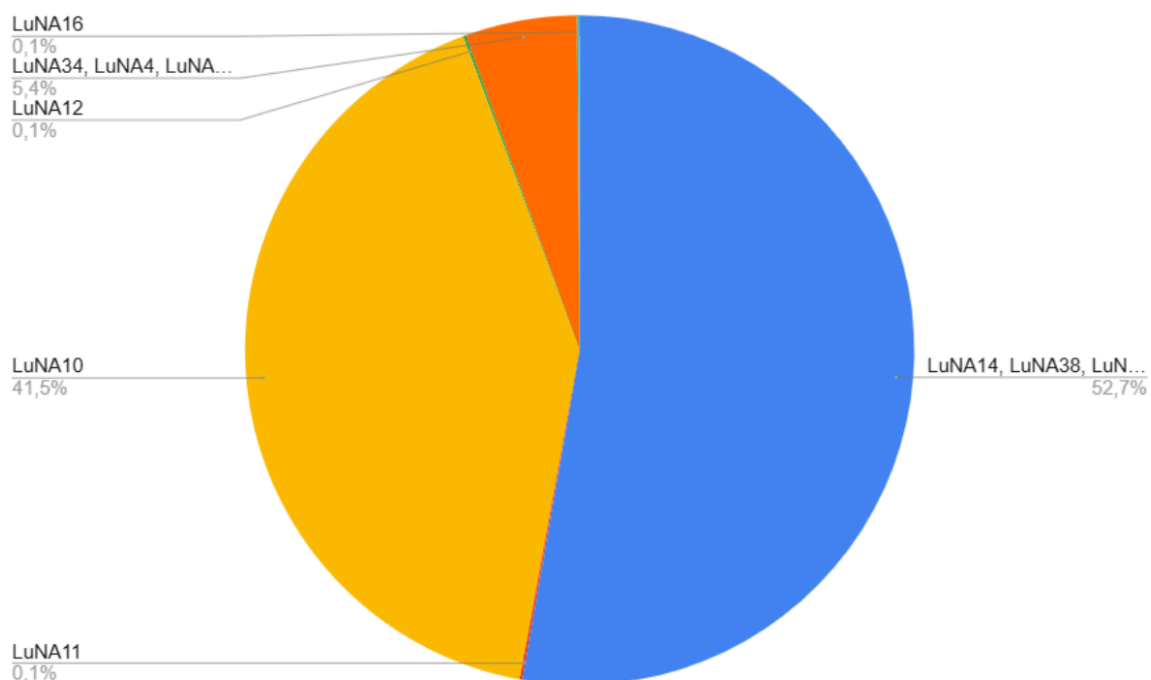


Рисунок 12 – Доля времени работы каждой проверки AST от общего времени работы анализатора на примере решения задачи о движении протопланетного диска методом PIC

4.2 Анализ результатов тестирования

Нетрудно заметить, что две проверки AST занимают 94.5% от всего времени работы анализатора. Каждая из них работает существенную долю от общего времени (41.6% и 52.9%). Значит анализатор не будет повышать скорость своей работы при количестве параллельных потоков, больших чем 2. Результаты запуска на двух тестовых программах это подтверждают.

ЗАКЛЮЧЕНИЕ

За ВКР было произведено ознакомление с языком LuNA, написаны тестовые программы, чтобы изучить синтаксис и семантику языка. Принято участие в составлении базы ошибок, свойственных фрагментированным программам. Были изучены структуры данных, алгоритмы и подходы к анализу исходного кода. Также произведено ознакомление с известными статическими анализаторами: PVS-Studio и cppCheck. Спроектирован и разработан анализатор языка LuNA на основе абстрактного синтаксического дерева. Грамматика языка была доработана, чтобы AST строилось в случае грамматических ошибок. На тестовых файлах из официального репозитория LuNA произведена оценка потребляемых ресурсов и добавлена возможность распараллеливать анализ для его ускорения. Совместно с другими студентами был разработан унифицированный формат ошибок, чтобы интегрировать 3 средства статического анализа в комплекс автоматизированной отладки ADAPT. Также в течение ВКР были написаны 3 тезиса на научные конференции.

Выпускная квалификационная работа выполнена мной самостоятельно и с соблюдением правил профессиональной этики. Все использованные в работе материалы и заимствованные принципиальные положения (концепции) из опубликованной научной литературы и других источников имеют ссылки на них. Я несу ответственность за приведенные данные и сделанные выводы.

Я ознакомлен с программой государственной итоговой аттестации, согласно которой обнаружение плагиата, фальсификации данных и ложного цитирования является основанием для не допуска к защите выпускной квалификационной работы и выставления оценки «неудовлетворительно».

Курбатов Максим Андреевич

ФИО студента

Подпись студента

« ____ » _____ 20 __ г.

(заполняется от руки)

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. Воеводин В.В., Воеводин Вл. В. "Параллельные вычисления"-СПб.: БХВ-Петербург, 2002.-608 с. ISBN 5-94157-160-7
2. Perepelkin V. A. LuNA system for automatic construction of numerical parallel programs for multicomputers // Проблемы информатики. 2020. № 1 (46).
3. Малышкин В. Э. Технология фрагментированного программирования // Вестник Южно-Уральского государственного университета. Серия: Вычислительная математика и информатика. – 2012. — № 46 (305).
4. Чмиль А.В. Разработка динамического балансировщика нагрузки для runtime-системы LuNA // Информационные технологии : Материалы 58-й Междунар. науч. студ. конф. 10–13 апреля 2020 г. / Новосиб. гос. ун-т. — Новосибирск : ИПЦ НГУ, 2020. — С. 48.
5. Курбатов, М. А. Статический анализ исходного кода на примере языка LuNA [Текст] / М. А. Курбатов // ВЫЗОВЫ ГЛОБАЛИЗАЦИИ И РАЗВИТИЕ ЦИФРОВОГО ОБЩЕСТВА В УСЛОВИЯХ НОВОЙ РЕАЛЬНОСТИ. — 2022. — № 1. — С. 5.
6. Шилов Н. В. Основы синтаксиса, семантики, трансляции и верификации программ : учебное пособие [Текст] / Шилов Н. В. — 1. — Новосибирск: НГУ, 2011 — 292 с.
7. Vlasenko A.Yu., Gudov A.M. The Use of Erratic Behavior Templates in Debugging Parallel Programs by the Automated Validity Verification Method //Journal of Computer and Systems Sciences International, 2017, Vol. 56, №4, pp.708-720.
8. Ахо, А. Компиляторы: принципы, технологии и инструментарий, 2-е издание: Пер. с англ. [Текст] / А. Ахо [и др.] // М. ООО" ИД Вильямс", 2008. - 1177 с.
9. Nonnan, R.E. An algorithm for generating abstract syntax trees [Текст] / Nonnan, R.E. // Journal of Theoretical Computer Science. — 1985. — № 15(3). — С. 18.

10. Allen, F. Control flow analysis [Текст] // ACM Sigplan Notices. – ACM, 1970. – Vol. 5. – №. 7. – P. 1-19.
11. Khedker, U., Sanyal, A., & Sathe, B. (2009). Data Flow Analysis: Theory and Practice (1st ed.). CRC Press. <https://doi.org/10.1201/9780849332517>.
12. Дж.Ульман, Дж.Уид Реляционные базы данных [Текст] / Дж.Ульман, Дж.Уид — 2. — Москва: Лори, 2018 — 374 с.
13. Кормен Т.Х, Лейзерсон Ч.И, Ривест Р.Л., Штайн К. АЛГОРИТМЫ ПОСТРОЕНИЕ И АНАЛИЗ [Текст] / Томас Кормен, Чарльз Лейзерсон, Рональд Ривест, Клиффорд Штайн — 2. — Москва: Вильямс, 2011 — 1277 с.
14. Афанасьев, К.Е. Автоматизированный анализ корректности MPI-программ на основе определенных пользователем шаблонов ошибочного поведения [Текст] / К.Е. Афанасьев, А.Ю. Власенко // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. – Томск, 2014. – №. 1 (26).
15. Пичугин, Д.П. Использование шаблонов при статическом анализе MPI-программ / Д.П. Пичугин // Материалы 55-й международной научной студенческой конференции : сб. ст. – Новосибирск, 2017. – С. 130.
16. Strout, M. Data-flow analysis for MPI programs [Текст] / M. Strout, B. Kreaseck, P. Hovland // Parallel Processing, 2006. ICCP 2006. International Conference on. – IEEE, 2006. – P. 175-184.
17. Cppcheck – static analysis tool for C/C++ code [Электронный ресурс]. URL: <https://cppcheck.sourceforge.io/> (дата обращения: 18.05.2024)
18. Официальный сайт PVS-Studio [Электронный ресурс] // URL: <https://www.viva64.com/ru/pvs-studio/> (дата обращения: 18.05.2024)
19. Список ошибок, находимых с помощью PVS-Studio // URL: <https://www.viva64.com/ru/w/#GeneralAnalysisCPP> (Дата обращения: 18.05.2024)

20. Vern Paxson, Will Estes, John Millaway. Flex / Vern Paxson, Will Estes, John Millaway. [Электронный ресурс] // Lexical Analysis With Flex, for Flex 2.6.2 : [сайт]. — URL: <https://docs.jade.fyi/gnu/flex.html> (дата обращения: 18.05.2024).
21. VINU V. DAS Compiler Design Using FLEX and YACC // PHI Learning Pvt. Ltd., 2007 p. 72-85
22. Charles Donnelly, Richard Stallman / Charles Donnelly, Richard Stallman [Электронный ресурс] // Bison : [сайт]. — URL: <http://gnu.ist.utl.pt/software/bison/manual/pdf/bison.pdf> (дата обращения: 18.05.2024).
23. Lion Kortlepel Writing a Simple and Effective Thread-Pool in C++ / Lion Kortlepel [Электронный ресурс] // Lion's Blog : [сайт]. — URL: <https://blog.kortlepel.com/tutorials/c++/2022/05/16/simple-effective-thread-pool-cpp.html> (дата обращения: 18.05.2024).
24. Шарый, С. П. Курс ВЫЧИСЛИТЕЛЬНЫХ МЕТОДОВ [Текст] / С. П. Шарый — 1. — Новосибирск: Институт вычислительных технологий СО РАН, 2016 — 530 с.
25. Киреев С.Е., Использование системы фрагментированного программирования LuNA для параллельной реализации РС-метода // Параллельные вычислительные технологии – XI международная конференция, ПАВТ'2017, г. Казань, 3-7 апреля 2017 г. Короткие статьи и описания плакатов. Челябинск: Издательский центр ЮУрГУ, 2017, с. 519.

ПРИЛОЖЕНИЕ А

Исходный код лексического анализатора для AST-analyzer

```
1  %{
2
3      #include "../parser/ast.hpp"
4      #include "grammar.tab.hpp"
5
6
7      #include <string>
8      #include <cstring>
9
10     void yyerror(const char*);
11     extern "C" int yywrap() { return 1; }
12
13     void next_line();
14     std::string yystring();
15     void next_token(const std::string&);
16     extern bool has_errors;
17     using namespace std;
18
19
20     extern int cur_position_in_line;
21     extern int line_num;
22     extern ast* ast_;
23
24     bool is_end_of_define = false;
25
26     #define SAVE_TOKEN std::string* token = new std::string(yytext); \
27                                     luna_string* l = new
luna_string(token); \
28
29     l->set_position(cur_position_in_line); \
30                                     yylval.string_ = l; \
31                                     ast_->push_token(token)
32
33     #define YY_USER_ACTION yylloc.first_line = line_num; \
34                             yylloc.first_column = cur_position_in_line;
35
36     %}
37
```

```

38      %%
39
40      as { next_token(yystring()); return KW_AS; }
41      cf { next_token(yystring()); return KW_CF; }
42      df { next_token(yystring()); return KW_DF; }
43      import { next_token(yystring()); return KW_IMPORT; }
44      for { next_token(yystring()); return KW_FOR; }
45      if { next_token(yystring()); return KW_IF; }
46      else { next_token(yystring()); return KW_ELSE; }
47      let { next_token(yystring()); return KW_LET; }
48      in { next_token(yystring()); return KW_IN; }
49      out { next_token(yystring()); return KW_OUT; }
50      SIZE {next_token(yystring()); return KW_SIZE; }
51      sub { next_token(yystring()); return KW_SUB; }
52      while { next_token(yystring()); return KW_WHILE; }
53      int { SAVE_TOKEN; next_token(yystring()); return KW_INT; }
54      real { SAVE_TOKEN; next_token(yystring()); return KW_REAL; }
55      string { SAVE_TOKEN; next_token(yystring()); return KW_STRING; }
56      name { SAVE_TOKEN; next_token(yystring()); return KW_NAME; }
57      value { SAVE_TOKEN; next_token(yystring()); return KW_VALUE; }
58      rush { next_token(yystring()); return KW_RUSH; }
59      static { next_token(yystring()); return KW_STATIC; }
60      static_for { next_token(yystring()); return KW_STATIC_FOR; }
61      unrolling { next_token(yystring()); return KW_UNROLLING; }
62      CUDA { next_token(yystring()); return KW_CUDA;}
63      NOCPU { next_token(yystring()); return KW_NOCPU;}
64
65      "$" {next_token(yystring()); return BUCK; }
66      "&" {next_token(yystring()); return AMP; }
67      "=" {next_token(yystring()); return EQ; }
68      "<" {next_token(yystring()); return LT; }
69      ">" {next_token(yystring()); return GT; }
70      ";" {next_token(yystring()); return SCOLON; }
71      ":" {next_token(yystring()); return COLON;}
72      "?" {next_token(yystring()); return QMARK;}
73      "," {next_token(yystring()); return COMMA; }
74      "." {next_token(yystring()); return DOT; }
75      ".." {next_token(yystring()); return DIAP; }
76      "{" {next_token(yystring()); return LCB; }
77      "}" {next_token(yystring()); return RCB; }
78      "(" {next_token(yystring()); return LB; }
79      ")" {next_token(yystring()); return RB; }
80      "[" {next_token(yystring()); return LSB; }

```

```

81     "]" {next_token(yystring()); return RSB; }
82     "+" {next_token(yystring()); return PLUS; }
83     "-" {next_token(yystring()); return MINUS; }
84     "*" {next_token(yystring()); return MUL; }
85     "/" {next_token(yystring()); return DIV; }
86     "%" {next_token(yystring()); return MOD; }
87     "#" {next_token(yystring()); return SHARP; }
88     "@" {next_token(yystring()); return AT; }
89     "==" {next_token(yystring()); return DBLEQ; }
90     "<=" {next_token(yystring()); return LEQ; }
91     ">=" {next_token(yystring()); return GEQ; }
92     ">" {next_token(yystring()); return GEQ; }
93     "!=" {next_token(yystring()); return NEQ; }
94     "&&" {next_token(yystring()); return DBLAMP; }
95     "||" {next_token(yystring()); return DBLPIPE; }
96     "-->" {next_token(yystring()); return ARROW; }
97     "<--" {next_token(yystring()); return LARROW; }
98     "<<" {next_token(yystring()); return LARR; }
99     ">>" {next_token(yystring()); return RARR; }
100    "C++" {next_token(yystring()); return KW_CPP; }
101    "__block" {next_token(yystring()); return KW_BLOCK; }
102
103    [0-9]+ {
104        SAVE_TOKEN;
105        next_token(yystring());
106        return INT;
107    }
108
109    [0-9]+ "." [0-9]+ {
110        SAVE_TOKEN;
111        next_token(yystring());
112        return REAL;
113    }
114
115    \\\/.*\\*\\\/ { std::cerr << std::string(yytext, yyleng) << std::endl;
/* comments */ }
116
117
118    \\`.*\\n { std::cerr << std::string(yytext, yyleng) << std::endl; /*
comments */ }
119
120    \\\/.*$ { next_token(yystring()); std::cerr << std::string(yytext,
yyleng) << std::endl; /* comments */ }

```

```

121
122     \"[^\"]*\" {
123         SAVE_TOKEN;
124         next_token(yystring());
125         return STRING;
126     }
127
128     [A-Za-z_][A-Za-z0-9_]* {
129         SAVE_TOKEN;
130         next_token(yystring());
131         return NAME;
132     }
133
134     \n {
135         next_line();
136         next_token(yystring());
137
138     }
139
140     [ \t]+ { next_token(yystring()); }
141
142     . { std::cerr << ((string("invalid symbol: ") + yystring()).c_str())
<< std::endl; }
143
144     %%
145
146     extern int cur_position_in_line;
147     extern int line_num;
148     extern string line;
149     extern string prev_line;
150     extern uint tokens;
151
152     void next_line() {
153         if (!line.empty()) {
154             prev_line = line;
155         }
156         line = "";
157         line_num++;
158         cur_position_in_line = 1;
159     }
160
161     void next_token(const std::string &s) {
162         tokens++;

```

```

163         yyloc.first_column = cur_position_in_line;
164
165         cur_position_in_line += s.size();
166         line += s;
167
168         yyloc.last_column = cur_position_in_line;
169     }
170
171     std::string yystring() {
172         return std::string(ytext, yyleng);
173     }
174
175     void yyerror(const char* msg) {
176         /* std::cerr << "yyerror\n"; */
177         /* if (has_errors) return; */
178
179         std::cerr << "Syntax error: Unexpected token at line : " <<
line_num << "\n" << line << std::endl;
180     }

```

ПРИЛОЖЕНИЕ Б

Исходный код синтаксического анализатора для AST-analyzer

```
all:
    program |
;

program:
    sub_def
    | program sub_def
;

sub_def:
    KW_SUB control_pragma code_id opt_params block
    | KW_CPP KW_SUB code_id opt_params KW_BLOCK LB INT RB
    | KW_IMPORT code_id LB opt_ext_params RB KW_AS code_id SCOLON
    | KW_IMPORT code_id LB opt_ext_params RB KW_AS code_id COLON KW_CUDA
SCOLON
    | KW_IMPORT code_id LB opt_ext_params RB KW_AS code_id COLON KW_CUDA
COMMA KW_NOCPU SCOLON
    | KW_CPP NAME KW_BLOCK LB INT RB
;

opt_ext_params:
    ext_params_seq |
;

ext_params_seq:
    code_df_param
    | ext_params_seq COMMA code_df_param
;

code_df_param:
    type code_df
;

code_df:
    NAME |
;

type:
    KW_INT
    | KW_REAL
```

```

    | KW_STRING
    | KW_NAME
    | KW_VALUE AMP
    | KW_VALUE
;

block:
    statement | LCB opt_dfdecls statement_seq RCB opt_behavior
;

opt_dfdecls:
    dfdecls |
;

dfdecls:
    KW_DF name_seq SCOLON;

name_seq:
    NAME |
    name_seq COMMA NAME
;

statement_seq:
    statement |
    statement_seq statement
;

control_pragma:
    LARR where_type COMMA expr RARR |
    LARR where_type COMMA expr COMMA expr RARR |
    LARR where_type RARR |
;

statement:
    cf_statement
    | let_statement
    | for_statement
    | while_statement
    | if_statement
;

cf_statement:
    opt_label code_id opt_exprs opt_setdf_rules opt_rules opt_behavior

```

SCOLON

```

;

opt_behavior:
    AT LCB behv_pragmas_seq RCB |
;

behv_pragmas_seq:
    behv_pragma
    | behv_pragmas_seq behv_pragma
;

behv_pragma:
    NAME id EQ expr SCOLON
    | NAME id EQG expr SCOLON
    | NAME id_seq SCOLON
    | NAME COLON expr SCOLON
    | name_seq SCOLON
;

id_seq:
    id
    | id_seq COMMA id
;

let_statement:
    KW_LET assign_seq block
;

for_statement:
    KW_FOR control_pragma NAME EQ expr DIAP expr block
;

while_statement:
    KW_WHILE control_pragma expr COMMA NAME EQ expr DIAP KW_OUT id block
;

if_statement:
    KW_IF expr block %prec IFX
;

assign_seq:
    assign
    | assign_seq COMMA assign

```

```

;

assign:
    NAME EQ expr
;

opt_label:
    KW_CF id COLON |
;

id:
    NAME
    | id LSB expr RSB;

opt_exprs:
    LB exprs_seq RB {
        $$ = new opt_exprs($2);
    }
    | LB RB {
        $$ = new opt_exprs(nullptr);
    }
;

exprs_seq:
    expr
    | exprs_seq COMMA expr
;

opt_setdf_rules:
    RARR opt_exprs |
;

opt_rules:
    ARROW opt_exprs |
;

code_id:
    NAME
;

expr:
    INT
    | REAL

```

```

    | STRING
    | KW_INT LB expr RB
    | KW_REAL LB expr RB
    | KW_STRING LB expr RB
    | expr PLUS expr
    | expr MINUS expr
    | expr MUL expr
    | expr DIV expr
    | expr MOD expr
    | expr LT expr
    | expr GT expr
    | expr LEQ expr
    | expr GEQ expr
    | expr DBLEQ expr
    | expr NEQ expr
    | expr DBLAMP expr
    | expr DBLPIPE expr
    | LB expr RB
    | id
    | expr QMARK expr COLON expr
;

opt_params:
    LB params_seq RB
    | LB RB;

params_seq:
    param
    | params_seq COMMA param
;

param:
    type NAME
;

where_type:
    KW_RUSH
    | KW_STATIC
    | KW_STATIC_FOR
    | KW_UNROLLING
    |
;
%%

```

ПРИЛОЖЕНИЕ В

Статический анализатор LuNA-программ AST-analyzer

Руководство оператора

Листов 5

Новосибирск 2024

АННОТАЦИЯ

В данном программном документе приведено руководство оператора по применению модуля статического анализа LuNA-программ AST-analyzer.

В данном программном документе, в разделе «Назначение программы» указаны сведения о назначении программы и информация, достаточная для понимания функций программы и ее эксплуатации.

В разделе «Условия выполнения программы» указаны требования, необходимые для выполнения программы.

В разделе «Выполнение программы» указана последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы.

Оформление программного документа «Руководство оператора» произведено по требованиям ЕСПД: 19.101-77, 19.105-78, ГОСТ 19.505-79.

СОДЕРЖАНИЕ

Аннотация	68
1 Назначение программы	70
1.1 Функциональное назначение программы	70
1.2 Эксплуатационное назначение программы	70
1.3 Состав функций	70
2 Условия выполнения программы	70
2.1 Минимальный состав аппаратных средств	70
2.2 Минимальный состав программных средств	70
2.3 Требование к персоналу	71
3 Выполнение программы	71
3.1 Загрузка и запуск программы	71
3.2 Выполнение программы	71
3.3 Завершение работы программы	72

1 Назначение программы

1.1 Функциональное назначение программы

Программа предназначена для анализа исходного кода на языке LuNA.

1.2 Эксплуатационное назначение программы

Программа создана для разработчиков, пишущих на языке LuNA. Она призвана помочь в отладке исходного кода.

1.3 Состав функций

Программа обеспечивает возможность выполнения перечисленных ниже функций:

- Построение абстрактного синтаксического дерева препроцессированного кода.
- Предоставление информации об обнаруженных в программе ошибках в формате JSON.
- Предоставление абстрактного синтаксического дерева в формате JSON.

2 Условия выполнения программы

2.1 Минимальный состав аппаратных средств

Программа предназначена для использования на персональном компьютере. Устройство, на котором запускается программа, должно иметь следующие характеристики:

1. Оперативная память объемом не менее 4 Гб;
2. Жёсткий диск объёмом не менее 64 Гб;
3. Процессор: два или более ядра с тактовой частотой не менее 1.5 ГГц;
4. Монитор;
5. Клавиатура.

2.2 Минимальный состав программных средств

1. Операционная система семейств Windows или Linux;
2. Компилятор языка C++20 (GCC, Clang, MSVC);
3. Командный интерпретатор bash.

2.3 Требование к персоналу

Конечный пользователь программы (оператор) должен обладать практическими навыками использования командного интерпретатора выбранной ОС.

3 Выполнение программы

3.1 Загрузка и запуск программы

Запуск программы осуществляется вызовом команды `adapt` с указанием пути к анализируемому файлу, который должен содержать текст программы на языке LuNA.

Пользователь может указать следующие опциональные параметры:

1. `--run [используемый анализатор]` — запустить анализатор указанного типа; варианты – `degssa` (запуск модуля DeGSA, использующего граф зависимостей по данным), `ast` (запуск модуля AST-analyzer, использующего абстрактное синтаксическое дерево) и `prolog` (запуск модуля `prolog-analyzer`, использующего логическое программирование).
2. `--no-cleanup` – не очищать порожденные во время работы анализатора файлы.
3. `--help` – вывести информацию об опциях запуска комплекса и сразу завершить работу.

3.2 Выполнение программы

После запуска комплекс ADAPT проведет анализ указанной программы на языке LuNA и напечатает в консоль информацию обо всех найденных ошибках. Пример его вывода:

Листинг 34 – Пример вывода комплекса отладки ADAPT

```
Found 1 errors:
(1) error[LuNA38]: Using non-integer variables within loop
boundaries.
For: Iterator i, first 1.1, last 10
Declared:
    File "main.fa", line 2, in main.
        for i=1.1..10 { // error
```

3.3 Завершение работы программы

Программа завершается автоматически после вывода информации об обнаруженных ошибках. В случае необходимости программу можно прервать в любой момент, воспользовавшись комбинацией клавиш Ctrl + C.