Федеральное государственное бюджетное учреждение науки
Институт динамики систем и теории управления имени В.М. Матросова
Сибирского отделения Российской академии наук

На правах рукописи

Михайлов Андрей Анатольевич

# Методы декомпиляции объектного кода Delphi

05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

ДИССЕРТАЦИЯ
на соискание ученой степени
кандидата технических наук

Научный руководитель
к. т. н., доцент
Хмельнов Алексей Евгеньевич

Иркутск – 2017

# Оглавление

# Введение

**Актуальность темы исследования.**

Для разработки большинства сложных программных систем часто используются готовые компоненты, предоставляемые в виде скомпилированных модулей. Такой подход существенно сокращает время и стоимость создания программного обеспечения. С другой стороны, наличие сторонних модулей уменьшает надежность программного обеспечения и его информационную безопасность из-за возможного наличия уязвимостей, способствующих успешным атакам на информационную систему. Кроме того, сторонние компоненты могут содержать ошибки, устранение которых может быть затруднено из-за невозможности связаться с разработчиком, утраты разработчиком исходных кодов и т. д. В некоторых случаях может потребоваться доработка сторонних модулей, исходные тексты которых отсутствуют.

Во многих организациях используются унаследованные информационные системы, разработанные с использованием устаревших технологий и программно-аппаратных платформ, но содержащие большой объем важной информации. Унаследованное программное обеспечение трудно сопровождать и поддерживать, так как эти системы разрабатывались достаточно давно и очень часто единственное, что имеется в распоряжении специалистов – это скомпилированное представление программы.

Исполняемый и ассемблерный код труден для анализа человеком и требует от специалиста предельной внимательности и больших трудозатрат, а также знания архитектуры процессора и семантики машинных команд для восстановления даже не слишком сложных операторов языков высокого уровня (ЯВУ). Наличие восстановленной программы на языке высокого уровня позволяет преодолеть указанные трудности. Для повышения уровня абстракции программ, представленных в виде исполняемых файлов, содержащих исполняемый код, используются декомпиляторы.

Проблема декомпиляции, являющаяся задачей обращения процесса компиляции, свое развитие получила с начала активного распространения первых высокоуровневых языков программирования, которые повышают уровень абстракции, значительно упрощая процесс разработки.

Для декомпиляторов с момента их создания находилось множество различных приложений. В 1960-х годах они использовались для перевода программ с компьютеров второго поколения на компьютеры третьего. В 1970 - 1980-х годах декомпиляторы использовались для восстановления исходного кода и внесения изменений в исполняемый код. В 1990-х декомпиляторы стали серьезным инструментом обратной инженерии, который позволяет решать задачи верификации программ, обнаружения вредоносного кода, переноса программ с одной платформы на другую и т. д.

Одними из первых проблемы декомпиляции начали исследовать Дж. К. Доннели и Х. Энглендер из лаборатории NEL (1960 г) [1]. В разное время проблемами декомпиляции и анализа объектного кода занимались такие ученые как У. Сассаман (1966) [2], К. Р. Холландера (1973) [3], Ф. Л. Фридман (1974) [4], В. Шнайдер и Г. Уиниджер (1974) [5], Д.А. Уоркман (1978) [6], К. Цифуентес (1991) [7], А. Майкрофт (1999) [8], М. Ван Эммерик (2007) [9]. Наиболее значимой работой современности является диссертация Кристины Цифуентес, которая посвящена разработке техник декомпиляции 16 битного машинного кода в язык С. В своей работе Цифуентес впервые предложила методы структурирования управляющего графа для восстановления операторов высокоуровневых языков программирования, таких как `if-then`, `if-then-else`, `while`, `do`, `for`.

Достаточно много работ [10–18] по анализу бинарного и исходного кода ведется в Институте системного программирования РАН Аветисяном А. И., Падаряном А. А., Гайсоряном С. С. и другими.

Одна из первых попыток создания универсального декомпилятора, который бы не зависел от компилятора и опций, с помощью которых исполняе-

мый файл был получен, была предпринята в 2002 году в проекте Boomerang. Результаты показали, что декомпилятор очень сильно зависит от языка написания исходной программы. В 2007 году один из участников проекта (Ван Эммерик) в своей работе рассмотрел вопрос применения SSA (Static single assignment form) формы для упрощения некоторых аспектов декомпиляции. Ван Эммерик провел исследование методов компиляции, применимых при декомпиляции, а также, предложил новый алгоритм поиска рекурсий, рассмотрел вопросы восстановления типов данных, косвенных переходов и вызовов процедур.

Из последних работ, посвященных проблемам декомпиляции, можно выделить диссертацию [19] Трошиной Е. Н., которая была выполнена в 2009 году в МГУ имени М. В. Ломоносова. В данной работе предложено новое требование к методам декомпиляции, которое определяется полнотой восстановления высокоуровневых конструкций целевого языка, а также проведена сравнительная оценка качества восстановления программ различными декомпиляторами. В работе предложены методы структурного анализа управляющего графа, которые позволили восстанавливать высокоуровневые операторы `for` и `switch`. Основным результатом данной работы является новый метод восстановления составных и базовых типов данных на основе методов статического и динамического анализа, который был реализован в инструменте TyDec. На текущий момент данный проект развился в декомпилятор SmartDec, который доступен как в качестве плагина к интерактивному дизассемблеру IDA Pro, так и как самостоятельный продукт.

Большинство исследований, направленных на создание универсальных методов декомпиляции, на практике разрабатываются исходя из соображения, что исходная программа была написана на определенном языке программирования, чаще всего на C/C++. В итоге, результат декомпиляции очень сильно зависит от компилятора, с помощью которого была получена программа. На текущий момент автору не известны реализации декомпилято-

ров данного типа способные полностью в автоматическом режиме провести корректную декомпиляцию. Из всех известных современных инструментов (Boomerang, Dcc, REC, Hex-Rays, SmartDec) наиболее применимы на практике только два – это плагин к интерактивному дизассемблеру IDA Pro Hex-Rays и SmartDec. Но даже они довольно часто генерируют код, который семантически не эквивалентен исходному.

Более успешными с практической точки зрения можно считать исследования, направленные на декомпиляцию программ, скомпилированных в байт-код виртуальных машин (ILSpy, JavaDecompiler, NET Reflector и. т. д.). Качество работы декомпиляторов данного типа объясняется тем, что на вход они получают файлы, содержащие дополнительные метаданные. Например, байт-код CIL имеет следующие преимущества по сравнению с машинным кодом: код отделен от данных; машина является стековой; стек жестко типизирован и используется только для хранения промежуточных результатов; при вычислении выражений виртуальная машина использует объектные структуры данных.

Помимо исполняемых файлов, компилятором часто генерируется промежуточный объектный код, который содержит в себе дополнительную информацию, наличие которой повышает качество декомпиляции. Но объектные файлы обычно не предназначены для дальнейшего распространения, а используются только редактором связей для получения исполняемого кода.

Среди объектных файлов особое место занимают файлы DCU, используемые компиляторами различных версий Delphi. С одной стороны, такие файлы технически можно отнести к объектным файлам, поскольку в дальнейшем, с использованием редактора связей, из них собирается загрузочный модуль. С другой стороны, файлы DCU содержат больше сведений, чем типичные объектные файлы. Так, там кодируется вся информация, полученная компилятором из исходных текстов модуля, и, в том числе, информация об определенных в этом модуле типах данных. Файл DCU может полностью за-

менить исходный текст для той версии компилятора, при помощи которой он был создан: в отличие от объектных файлов, не требуется дополнительный заголовочный файл, чтобы воспользоваться файлом DCU при компиляции. Этой особенностью активно пользуются разработчики программных модулей, которые часто распространяют их в формате DCU без предоставления исходных текстов, в особенности тогда, когда это делается на коммерческой основе.

В том случае, когда разработчик прекращает развитие своих программных модулей, отсутствие исходных текстов не позволяет применить эти модули с новыми версиями компилятора. Также становится невозможным: исправить обнаруженные ошибки, проанализировать качество кода модуля, не говоря уже о его доработке.

В настоящее время практически все известные автору «декомпиляторы» Delphi (DelphiDecompiler, Revendepro, Exe2Dpr, IDR, EMS Source Rescuer) для проведения разбора объектного кода используют простой (настраиваемый при помощи спецификаций) статический дизассемблер[1], не использующий информацию о потоках данных, разработанный Хмельновым А. Е. в ИДСТУ СО РАН. Данный инструмент позволяет получить только интерфейсную часть модуля и ассемблерный код и по сути своей декомпилятором не является. Таким образом, актуальность данной работы обуславливается отсутствием декомпиляторов объектного кода Delphi и областью их применения.

**Цель работы** – разработка методов декомпиляции и анализа объектного кода Delphi.

Для достижения поставленных целей были решены следующие **задачи**:

1. Проведен анализ существующих методов декомпиляции объектного кода;

[1] http://hmelnov.icc.ru/DCU/index.ru.html

2. Впервые разработана технология декомпиляции объектного кода Delphi;

3. Реализовано инструментальное средство для анализа и декомпиляции объектного кода Delphi;

4. Проведена апробация созданной технологии на задачах автоматизации анализа объектного кода Delphi;

**Научная новизна**

1. Разработаны новые методы декомпиляции объектного кода Delphi, скомпилированного под платформу .NET, позволяющие восстанавливать программу на языке CIL в программу на языке Delphi.

2. На основе предложенных методов реализован оригинальный декомпилятор объектных файлов Delphi, скомпилированных под платформу .NET.

3. Разработан оригинальный метод визуализации управляющего графа на плоскости. Основной особенностью разработанного метода визуализации является возможность использования изобразительных соглашений, принятых при проектировании блок-схем алгоритмов.

**Теоретическая и практическая значимость.**

Отдельные результаты диссертации были получены в рамках программы фундаментальных исследований СО РАН проект IV.38.2.3. «Новые методы, технологии и сервисы обработки пространственных и тематических данных, основанные на декларативных спецификациях и знаниях» (2013-2015 гг.), а также научного проекта РФФИ № 15-37-20042 мол_а_вед. Разработанные в рамках диссертационной работы методы и инструментальное средство позволяют повысить эффективность, снизить трудозатраты и сократить сроки решения задач, связанных с анализом исполняемого кода. В частности, разработанное инструментальное средство может снизить трудозатраты при решении задач связанных с поддержкой унаследованного ПО, имеющего в

своем составе компоненты, представленные в виде скомпилированных модулей Delphi.

Созданное программное обеспечение зарегистрировано в Федеральной службе по интеллектуальной собственности, патентам и товарным знакам [№2014617137, №2016612670].

**Результаты, выносимые на защиту:**

1. Методы декомпиляции объектного кода Delphi, скомпилированного под платформу .NET.

2. Программная реализация декомпилятора объектных файлов Delphi скомпилированных под платформу .NET, позволяющего восстанавливать программы на низкоуровневом языке CIL в программы на языке Delphi.

3. Метод визуализации управляющего графа на плоскости, позволяющий использовать изобразительные соглашения, принятые при проектировании блок-схем.

**Степень достоверности и апробация результатов обеспечивается:**

- обоснованным использованием методов и технологий декомпиляции информационных систем и визуализации уграфов, опубликованных в открытой печати;

- согласованностью с результатами исследований других авторов, представленных в печатных изданиях;

- работоспособностью разработанного декомпилятора и модуля визуализации графов, адекватностью полученных данных в результате их тестирования и сравнении с аналогичными средствами;

Основные результаты диссертации и ее отдельные положения, а также результаты конкретных прикладных исследований и разработок, обсуждались на научных семинарах ИДСТУ СО РАН, ИСИ СО РАН, ИСП РАН,

докладывались на отечественных и международных научных конференциях: «Ляпуновские чтения – 2012» (Иркутск, 2012 г.); XIII Всероссийская конференция молодых ученых по математическому моделированию и информационным технологиям (Новосибирск, 2012 г.); Ляпуновские чтения – 2013 (Иркутск, 2013 г.); «Малые Винеровские чтения 2013» (Иркутск, 2013); XVIII Байкальская Всероссийская конференция «Информационные и математические технологии в науке и управлении». (Иркутск, 2013 г.); II Российско-Монгольской конференции молодых ученых (п. Ханх, Монголия, 2013 г.); «Ляпуновские чтения – 2014» (Иркутск, 2014 г.); III Российско-монгольской конференции молодых ученых по математическому моделированию, вычислительно-информационным технологиям и управлению Иркутск (Россия) - Ханх (Монголия) (п. Ханх, Монголия, 2015 г.); 5th International Workshop on Computer Science and Engineering — Russia, Moscow: Bauman Moscow State Technical University (Москва, 2015 г.); The 39th International ICT Convention – MIPRO 2016 (г. Опатия, Хорватия, 2016 г.)

**Публикации.** Материалы диссертации опубликованы в 16 печатных работах [20–35], из них 3 из списка ВАК [20–22], 1 статья из списка WOS [23], 2 авторских свидетельства [34, 35].

**Личный вклад автора.** Все выносимые на защиту научные положения получены соискателем лично. В основных научных работах по теме диссертации, опубликованных в соавторстве, лично соискателем разработаны: в [21, 24, 25, 27, 28, 31] – методы декомпиляции объектного кода Delphi, скомпилированного под платформу .NET; [22, 32–34] – программная реализация декомпилятора объектных файлов Delphi, скомпилированных под платформу .NET; [20, 23, 26, 29, 30, 35] – метод визуализации управляющего графа на плоскости.

**Структура и объем диссертации.** Диссертация состоит из введения, обзора литературы, 4 глав, заключения и библиографии. Общий объем диссертации 155 страниц, из них 108 страницы текста, включая 19 рисунков.

Библиография включает 98 наименований на 10 страницах.

# Глава 1

# Обзор задачи декомпиляции

**Определение 1.0.1.** *Декомпиляция – это процесс автоматического восстановления программы на языке высокого уровня из программы на языке низкого уровня. Под декомпилятором мы будем понимать инструментальное средство, получающее на вход программу на языке ассемблера или другое аналогичное низкоуровневое представление и выдающее на выход эквивалентную ей программу на некотором языке высокого уровня. [36]*

Проблема декомпиляции, являясь обратным процессом процессу компиляции, свое развитие получила с начала активного распространения первых высокоуровневых языков программирования. Высокоуровневые языки программирования повышают уровень абстракции, значительно упрощая процесс программирования. Что вкупе с ростом вычислительных и аппаратных возможностей вычислительной техники позволяло писать все более сложные программы, затрачивая при этом меньше ресурсов.

Проблемами создания зыков программирования в разное время занимались такие ученые как Ахо А., Сети Р., Ульман Д.Ё. [37]. Бьерн Страуструп [38], разработавший язык программирования C++. Никлаус Вирт [39], разработчик языков программирования Паскаль, Модула-2, Оберон. Из отечественных ученых, основоположником системного программирования является Ершов А. П. [40] [41], под руководством которого были созданы такие языки, как Альфа, Альфа-6 и трансляторы с них.

Одна из первых работ, посвященная декомпиляции была опубликована профессором Моррисом Холстедом в 1962 году [1], который руководил проектом по декомпиляции машинного кода в язык Neliac [42]. Результатом данной работы стали фундаментальные методы, многие из которых исполь-

зуются при создании декомпиляторов по сей день. Первые декомпиляторы использовались в основном для переноса программ с компьютеров второго поколения на компьютеры третьего поколения. В начале 90-х декомпиляторы стали серьезным инструментом, позволяющем решать задачи обратной инженерии, такие как проверка на наличие вредоносного кода, верификации, восстановления и переноса кода с одной архитектуры на другую и т. д.

Одной из самых первых и значимых работ начала 90-х годов является диссертационная работа [7] Кристины Цифуентес. Данная работа посвящена разработки техник декомпиляции 16 битного машинного кода в язык C. В своей работе Цифуентес впервые предложила методы структурирования управляющего графа [43], для восстановления операторов высокоуровневых языков программирования, таких как `if-then`, `if-then-else`, `while`, `do`.

Одна из первых попыток создать универсальный декомпилятор, который бы не зависел от компилятора и опций, с помощью которых исполняемый файл был получен, была предпринята в 2002 году в проекта Boomerang [44]. В 2007 году один из участников проекта Михаель Джеймс Ван Эммерик закончил написание своей диссертационной работы [9]. В своей работе он рассмотрел вопрос применения SSA формы для упрощения некоторых аспектов декомпиляции. Ван Эммерик провел исследование методов компиляции, применимых при декомпиляции. Также, в данной работе представлен новый алгоритм поиска рекурсий, рассмотрены вопросы восстановления типов данных, косвенных переходов и вызовов.

Одной из последних работ, посвященных проблемам декомпиляции является диссертационная работа [19] Трошиной Е. Н., которая была выполнена в 2009 году в МГУ имени М. В. Ломоносова. В данной работе предложено новое требование к методам декомпиляции, которое определяется полнотой восстановления высокоуровневых конструкций целевого языка. Проведена сравнительная оценка качества восстановления программ различными декомпиляторами. Дополнены методы структурного анализа управляющего

графа, которые позволили восстанавливать высокоуровневые операторы `for` и `switch`. Основным результатом данной работы является новый метод восстановления составных и базовых типов данных на основе методов статического и динамического анализа, который был реализован в инструменте TyDec [45]. На текущий момент данный проект развился в декомпилятор SmartDec [46], который доступен, как в качестве плагина к интерактивному дизассемблеру IDA Pro, так и как самостоятельный продукт.

## 1.1. Проблемы декомпиляции

В общем случае для произвольного исполняемого файла задача декомпиляции чрезвычайно сложна, в силу того что в процессе компиляции утрачивается много информации о высокоуровневой программе. Часть информации, например комментарии, имена импортируемых модулей утрачивается безвозвратно. Программа на языке низкого уровня представляет собой последовательность инструкций с условными и безусловными переходами, в которой отсутствует информация об используемых типах данных, высокоуровневых операторах и т д. Однако, часть утраченной информации можно частично или полностью восстановить. В то же время существует ряд в общем случае алгоритмически неразрешимых проблем [47], таких как отделение кода от данных, отделение констант от адресов и т. д.

Программа, написанная на языке высокого уровня, содержит много информации, предназначенной в первую очередь для разработчика, и обладает более высоким уровнем абстракции. В ней содержится много избыточной информации упрощающей процесс разработки, поддержки, понимания кода. Исходная программа содержит комментарии, присутствует информация об используемых типах данных, разбиение на классы, имена процедур и функций и. д. Такая информация не нужна процессору, который обладает ограниченным набором низкоуровневых машинных команд, для исполнения

программы.

С развитием архитектуры процессора появляются новые инструкции и регистры. Все чаще используются векторные инструкции, которые позволяют выполнить операцию сразу над несколькими значениями, помещенными в один регистр. В таком случае компилятору необходимо выровнять данные, при этом используются инструкции, которые не имеют семантического эквивалента в исходной программе. В процессорах Intel на архитектуре «Haswell» [48] появились инструкции позволяющие делать GATHER-операции [49], при таких операциях не требуется, чтобы данные были выровнены. С развитием векторных регистров и набором инструкций для работы с ними, основной задачей компиляторов все больше становится подготовка данных для последующего выполнения одной массивной операции.

Перед разработчиком декомпилятора встает ряд теоретических и практических проблем, некоторые из них могут быть решены эвристически, другие требуют непосредственного участия человека. Из-за этого, как правило, существует класс программ, которые декомпилятор может обрабатывать полностью в автоматическом режиме, остальные должны обрабатываться в полуавтоматическом режиме с участием человека. Это отличает декомпилятор от компилятора, который производит автоматическую трансляцию всех программ, записанных на исходном языке. Ниже представлен список некоторых проблем, возникающих в процессе декомпиляции:

**Разделение кода и данных.** Хотя большинство форматов исполняемых и объектных файлов разделяют секцию кода и данных, многие компиляторы включают в исполняемый код данные. Например, оператор множественного выбора `case` может быть преобразован компилятором в таблицу переходов. Также в блок кода компилятор может записывать представления констант различных типов (например, строковых и действительных типов данных). Данная проблема типична только для языков компилируемых в машинный код, предназначенный для выполнения на процессорах построенных по архи-

тектуре фон Неймана [50]. В виртуальной машине .NET, которая является стековой, код полностью отделен от данных, при этом команды виртуальной машины могут содержать аргументы более сложных типов. Так, для выполнения множественного выбора используется специальная команда `Switch`, включающая в свой состав таблицу переходов в качестве аргумента.

**Идиомы.** Идиомы – это последовательности операций, которые представляют собой логическую единицу, действие которой не является просто использованием первичного назначения инструкций. Например, команда сдвига `SHL EAX, 1` может быть интерпретирована, как операция умножения числа на 2.

**Реконструкция потока управления.** Реконструкция высокоуровневых операторов языков программирования, таких как циклы, условные операторы и т. д. Граф потока управления – это множество всех возможных путей исполнения программы, представленное в виде графа.

**Анализ потоков данных программы.** Под анализом потоков данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте.

## 1.2. Структура декомпилятора

**Определение 1.2.1.** *Декомпиляция – это процесс, обратный процессу компиляции. По своей структуре декомпилятор, подобно компилятору, состоит из фаз, последовательно преобразующих программу из одного состояния в другое.*

Логические фазы, из которых состоит декомпилятор в общем случае перечислены ниже:

1. Синтаксический анализ.

2. Семантический анализ.

3. Генерация промежуточного кода.

4. Генерация графа потоков управления.

5. Анализ потока данных.

6. Анализ графа потоков управления.

7. Генерация целевого кода.

В отличии от компилятора, в декомпиляторе нет фазы лексического анализа. Основная проблема разбора машинных языков заключается в определении начала и конца инструкции. Первая стадия декомпиляции, которая включает в себя синтаксический и семантический анализ – это задача дизассемблирования. Машинный код необходимо перевести в язык ассемблера. Для анализа программы необходимо иметь её промежуточное представление. Промежуточное представление должно быть достаточно простым для того, чтобы его можно было легко получить из исходной программы, и, в то же время, должно быть достаточно легко получить из него код на целевом языке. Промежуточное представление используется для проведения машинно-независимых оптимизаций кода.

## 1.3. Виды декомпиляторов

*«Чем сильнее отличаются уровни абстракции между исходным кодом программы и кодом, в который она была скомпилирована, тем сложнее будет процесс ее восстановления».* [9]

Исходя из данного утверждения, все существующие декомпиляторы можно условно разделить по уровню абстракции представления, которое подается им на вход:

| **Проблема** | **Машинный код** | **Объектный код** | **Байт-код)** |
|---|---|---|---|
| Разделение кода и данных | + | +/– | – |
| Разделение констант и указателей | + | +/– | – |
| Анализ косвенных переходов | + | +/– | – |
| Оптимизация кода | + | + | + |
| Анализ потоков управления | + | + | + |
| Восстановление параметров функции | + | +/– | – |
| Объявление переменных | + | + | +/– |
| Анализ типов данных | + | + | +/– |
| **Итог** | **8** | **6** | **2,5** |

Таблица 1.1. Проблемы декомпиляции

### 1.3.1. Декомпиляторы машинного кода

Декомпиляторы данного типа решают самые сложные задачи обратной инженерии, потому что в процессе компиляции утрачивается вся информация, которая не требуется процессору для исполнения программы. На текущий момент автору не известны успешные реализации декомпиляторов данного типа. Из всех рассмотренных в данной работе инструментов (Boomerang, Dcc, REC, Hex-Rays, SmartDec) самые современные и единственные применимые на практике – это плагин к интерактивному дизассемблеру IDA Pro Hex-Rays [51] и декомпилятор SmartDec [46].

### 1.3.2. Декомпиляторы объектного кода

В объектном коде присутствует дополнительная информация, предназначенная для использования редактором связей. Такая информация может содержать в себе имена переменных, описания процедур и функций, импортируемых типов данных и т. д. Наличие такой информации может упростить процесс декомпиляции, тем самым повысив ее качество. Ниже (см. листинг 1.1) приведен пример разбора объектного файла с помощью инструмента DCU32INT.

Здесь имя вызываемой процедуры (`NewCmdSeqRef`) было определено из содержащейся в объектных файлах информации о перемещаемых адресах (`FixUp`), предназначенной для работы редактора связей. Привязка аргументов процедуры к некоторым регистрам (`EBX->@Self`, `ESI->@ATgt` и т.д.), определяется по сведениям, содержащимся в описаниях аргументов, и, более точно, по отладочной информации (при её наличии). Обращение к `DWORD PTR [EBX+52]` интерпретируется, как обращение к полю `Self.FNext`, благодаря информации о типах данных.

Листинг 1.1. Результат разбора функции с помощью инструмента DCU32INT

```
1 procedure TCILCtrlFlowNode.SetNext (Self: TCILCtrlFlowNode;
```

```
 2   ATgt: TCILCtrlFlowNode);
 3 begin
 4   // -- Part #0 --
 5   // -- Line #257 --
 6   00: S      [53                 | PUSH  EBX
 7   01: V      |56                 | PUSH  ESI
 8   02: <т     |8B F2              | MOV  ESI ,EDX
 9   04: <Ш     |8B D8              | MOV  EBX {@Self},EAX
10   // -- Part #1 --
11   // -- Line #258 --
12   06: <Ц     |8B D6              | MOV  EDX ,ESI {@ATgt}
13   08: <Г     |8B C3              | MOV  EAX ,EBX {@Self}
14   0A: и....  |E8(00 00 00 00      | CALL  NewCmdSeqRef{#\$60}
15   0F: % C4   |89 43 34           | MOV  DWORD PTR [EBX +52{@Self.FNext}],EAX
16   // -- Part #2 --
17   // -- Line #259 --
18   12: ^      |5E                 | POP  ESI
19   13: [      |5B                 | POP  EBX
20   14: Г      |C3                 | RET  NEAR
21 end;
```

### 1.3.3. Декомпиляторы байт-кода

Декомпиляторы данного типа являются наиболее успешными примерами в области обратной инженерии. Например, имея скомпилированную программу, написанную на языке программирования C# можно восстановить ее исходный код с помощью декомпилятора ILSpy, который практически всегда с точки зрения семантики будет соответствовать исходной программе. Качество работы декомпиляторов данного типа объясняется тем, что на вход они получают файлы, содержащие дополнительные метаданные. Например, байт-код CIL имеет следующие преимущества по сравнению с машинным кодом:

- код отделен от данных;
- машина является стековой, причем стек жестко типизирован;
- стек используется, как правило, только для хранения промежуточных результатов;
- большинство команд CIL получают свои аргументы на стеке, удаляют их со стека и помещают вместо них результат(ы) вычисления;
- машина является объектно-ориентированной: структура CIL отражает разбиение кода на классы, методы и т. д.

## 1.4. Обзор современных декомпиляторов

Все рассматриваемые декомпиляторы, кроме плагина Hex-Rays, на вход принимают исполняемый файл, и выдают программу на языке Си. В том случае, когда декомпилятор оказывается не в состоянии восстановить некоторый фрагмент исходной программы на языке Си, этот фрагмент сохраняется в виде ассемблерной вставки. Надо заметить, что даже небольшие исходные программы после декомпиляции зачастую содержат очень много ассемблерных вставок, что практически сводит на нет эффект от декомпиляции. В отличие от этого, плагин Hex-Rays принимает на вход программу, являющуюся результатом работы дизассемблера IDA Pro, то есть схему программы на ассемблеро-подобном языке программирования. В качестве результата Hex-Rays выдает восстановленную программу в виде схемы на Си-подобном языке программирования.

### 1.4.1. Декомпиляторы машинного кода

#### 1.4.1.1. Boomerang

Декомпилятор Boomerang является программным обеспечением с открытым исходным кодом. Разработка этого декомпилятора активно началась в 2002 году, на данный момент есть только альфа версия. Начиная с 2006 года проект не обновляется. Изначально задачей проекта была разработка такого декомпилятора, который восстанавливает исходный код из исполняемых файлов, вне зависимости от того, с использованием какого компилятора и с какими опциями исполняемый файл был получен. Для этого в качестве внутреннего представления было решено использовать представление программы со статическими одиночными присваиваниями. Однако, несмотря на поставленную цель, в результате декомпилятор не сильно адаптирован под различные компиляторы и чувствителен к применению различных опций, в частности, опций оптимизации. Еще одной особенностью, затрудняющей использование декомпилятора Boomerang, является то, что в нем не поддерживается распознавание стандартных функций библиотеки Си (и каких-либо иных системных библиотек).

#### 1.4.1.2. DCC

Проект по разработке этого декомпилятора был открыт в 1991 году и закрыт в 1994 году с получением главным разработчиком степени PhD. В качестве входных данных декомпилятор DCC принимает 16-битные исполняемые файлы в формате DOS EXE. Алгоритмы декомпиляции, реализованные в этом декомпиляторе, основаны на теории графов (анализ потока данных и потока управления). Для распознавания библиотечных функций используется сигнатурный поиск, для которого была разработана библиотека сигнатур. Однако надо заметить, что, несмотря на это, декомпилятор плохо справляется с выявлением функций стандартной библиотеки.

#### 1.4.1.3. REC

Этот проект был открыт в 1997 году компанией BackerStreet Software, но вскоре закрылся из-за ухода ведущего разработчика проекта. Позднее разработка декомпилятора продолжилась его автором в статусе собственного продукта. Сейчас декомпилятор распространяется свободно, но развивается медленно. Одной из особенностей рассматриваемого декомпилятора является то, что он восстанавливает исполняемые файлы различных форматов, в частности ELF и PE. Также декомпилятор REC можно использовать на различных платформах. В ходе тестирования этого декомпилятора было отмечено, что наиболее успешно он восстанавливает исполняемые файлы, полученные при компиляции с включением опций, которые отвечают за отключение оптимизаций и добавление отладочной информации.

#### 1.4.1.4. Hex-Rays

Hex-Rays не является самостоятельным программным продуктом, а распространяется в виде плагина к дизассемблеру IDA Pro. Это самое новое из рассматриваемых средств декомпиляции: плагин появился на рынке в 2007 году. Особенностью данного инструмента является то, что он, как отмечалось, восстанавливает программы, полученные на выходе дизассемблера IDA Pro. Среди алгоритмов, используемых в Hex-Rays, заслуживают внимания алгоритм сигнатурного поиска FLIRT [51] и алгоритм поиска параметров в стеке PIT (Parameter Identification and Tracking).

#### 1.4.1.5. SmartDec

Разработка декомпилятора SmartDec началась в рамках диссертационной работы [19] Е. Н. Трошиной в 2007 году. В 2009 появилась первая версия декомпилятора в язык C (TyDec). В 2010 году авторы продолжили исследования возможности восстановления программ из низкоуровневого представ-

ления в язык C++. В 2010 году появилась первая версия декомпилятора SmartDec, как плагина к интерактивному дизассемблеру IdaPro. В 2011 году компании разработала декомпилятор в языки C и C++ как самостоятельный продукт.

### 1.4.2. Декомпиляторы байт-кода

Именно декомпиляторы байт-кода являются в текущее время наиболее успешными. Для таких распространенных языков как Java и C# существуют инструменты, способные восстановить исходный код достаточно большого проекта, который будет пригоден для последующей его компиляции. Такие результаты обусловлены тем, что код предназначен для выполнения виртуальной машиной, которая имеет наиболее высокий уровень абстракции по сравнению с машинным кодом.

#### 1.4.2.1. ILSpy

Разработка декомпилятора ILSpy [52] началась в феврале 2011 года, после того, как компания Red Gate объявила о закрытии поддержки бесплатной версии декомпилятора .NET Reflector [53]. Данный декомпилятор использует проект Mono по созданию кроссплатформенного компилятора под .NET. Декомпилятор ILSpy практически всегда генерирует код, семантически эквивалентный исходному.

При этом достаточно легко найти пример «неудачной» декомпиляции с помощью декомпилятора ILSpy кода .NET, скомпилированного в Delphi:

Листинг 1.2. Результат разбора кода .NET, скомпилированного в Delphi, декомпилятором ILSPy

```
1 if (num <= num3) {
2   while (true) {
3     char c3 = s.get_Chars(num - 1);
4     if (c3 - '0' >= '\n') {
```

```
5       goto IL_205;
6     }
7     int num5 = (int)s.get_Chars(num - 1) - 48;
8     if (flag) {
9       long num6 = num2 * (long)((ulong)10) - (long)num5;
10      if (num6 > num2) {
11        goto IL_205;
12      }
13      num2 = num6;
14    } else {
15      long num6 = num2 * (long)((ulong)10) + (long)num5;
16      if (num6 < num2) {
17        goto IL_205;
18      }
19      num2 = num6;
20    }
21    if (num2 > MaxValue) {
22      break;
23    }
24    if (num2 < MinValue) {
25      goto Block_28;
26    }
27    num++;
28    flag2 = false;
29    if (num > num3) {
30      goto IL_205;
31    }
32  }
33  num2 = MinValue - (MaxValue - num2) - (long)((ulong)1);
34  goto IL_205;
35  Block_28:
36  num2 = MaxValue - (MinValue - num2) + (long)((ulong)1);
37 }
```

Как видно из листинга 1.2 даже в небольшом фрагменте декомпили-

рованнного кода присутствует пять безусловных операторов перехода `goto`. Это можно объяснить тем, что ILSpy изначально разрабатывался из предположения того, что исходная программа была написана на языке C# и не учитывает особенности компилятора Delphi.

### 1.4.3. Декомпиляторы Delphi

Существует немало инструментов для анализа программ, написанных на языке Delphi: Delphi Decompiler (DeDe), EMS Source Rescuer, IDR (Interactive Delphi Reconstructor), Revendepro, MultiRipper (MRip), Exe2Dpr. Но большая часть из них не поддерживается на данный момент, либо позволяет извлекать только ресурсы формы из исполняемого файла. Рассмотрим некоторые из этих инструментов.

#### 1.4.3.1. IDR

IDR (Interactive Delphi Reconstructor) – декомпилятор исполняемых файлов (EXE) и динамических библиотек (DLL), написанных на языке Delphi и выполняемых в среде 32х-разрядных операционных систем Windows.

Программа, прежде всего, предназначена для компаний, занимающихся разработкой антивирусного программного обеспечения. Она также может в значительной мере помочь программистам в восстановлении утраченных исходных текстов программ.

Текущей версией программы могут обрабатываться файлы (как GUI, так и консольных приложений), скомпилированные компиляторами версий Delphi 2 – Delphi XE3.

Конечной целью проекта является разработка программы, способной восстановить большую часть исходных Delphi-текстов из скомпилированного файла, но пока IDR, как и другие Delphi-декомпиляторы, сделать этого не может.

#### 1.4.3.2. EMS Source Rescuer

EMS Source Rescuer – это утилита для Borland Delphi® и C++Builder®, позволяющая восстановить утерянный исходный код. Source Rescuer восстанавливает все формы и модули данных проекта со всеми заданными свойствами и событиями. Восстановленные процедуры обработки событий не имеют тела (Source Rescuer – не декомпилятор), но указывают на соответствующий адрес кода исполняемого файла.

## 1.5. История развития Delphi

В первых версиях Delphi поддерживалась только одна платформа – Win 32 и выполнялась компиляция в машинный код 80x86. С появлением Delphi 8.0 была сделана ставка на появившуюся в то время технологию .NET: в этой версии Delphi было возможно разрабатывать программы только для .NET. Судя по дальнейшему развитию продукта, эта ставка не сыграла: разработчики не стали голосовать долларом за столь резкие движения, поэтому в последующих двух версиях Delphi 2005 и 2006 была возвращена работа с Win 32 (в сочетании с .NET), после чего поддержка .NET в Delphi была прекращена. Точнее, был выпущен отдельный продукт Delphi Prism, который впоследствии продолжил своё развитие как отдельная система и язык программирования Oxygene. Несмотря на то, что в текущей версии Delphi платформа .NET не поддерживается, для нас она представляет значительный интерес: поскольку существуют работоспособные декомпиляторы в C# для загрузочных модулей .NET (например, ILSpy, NetRefclector). Существование таких декомпиляторов объясняется достаточно высокоуровневым характером инструкций байт-кода CIL и использованием исключительно стека для представления результатов промежуточных вычислений (что существенно облегчает задачу восстановления выражений). По аналогии с этими де-

компиляторами автором был разработан декомпилятор для файлов DCUIL для платформы .NET, особенности реализации которого будут рассмотрены далее более подробно в главе 6.

В Delphi XE2 началась поддержка кроссплатформенной разработки: была реализована компиляция программ для OS X. Поскольку к тому времени компьютеры Apple выпускались с процессорами от Intel, используемый при этом машинный код не изменился. Также была реализована 64-битная версия компилятора для Windows. В последующих версиях Delphi была поддержана компиляция для iOS и Android. Приложение для iOS может разрабатываться с использованием эмулятора, при этом оно компилируется в машинный код 80x86. При разработке компиляторов для собственно iOS и Android была использована библиотека LLVM [54]. При этом по генерируемой компилятором промежуточной информации средствами LLVM создаются объектные файлы (*.o), содержащие значения глобальных данных и машинный код для процессора ARM. Сами файлы DCU не содержат эти данные, а представляют лишь ту информацию, которая не вошла в объектный файл (этот случай наглядно демонстрирует отличия формата DCU от обычных объектных файлов). Поскольку файл *.dcu создаётся независимо от соответствующего ему файла *.o, для связи между этими двумя файлами, например, чтобы найти в объектном файле код тела подпрограммы, описанной в файле DCU, используются декорированные имена (mangling) переменных, подпрограмм, типов данных и других сделанных в программном модуле определений. Не исключено, что при этом синтаксис mangling LLVM расширяется для учёта специфики Delphi. По крайней мере, не все имена, встречающиеся в объектных файлах, могут быть получены в соответствии с исходными текстами LLVM (файл Mangle.cpp). К настоящему времени работа по описанию используемого механизма декорирования имён ещё не закончена, что не позволяет использовать информацию из объектных файлов .o при разборе файлов DCU, скомпилированных для iOS и Android.

Начиная с версии Delphi 2005 в язык Object Pascal в новом качестве были возвращены inline-подпрограммы. Модификатор inline после заголовка подпрограммы позволяет компилятору заменять вызов этой подпрограммы на подстановку её кода, если такая замена оказывается целесообразной. Ранее компилятор Turbo Pascal поддерживал такой модификатор, но в очень неудобной форме: после ключевого слова inline программисту приходилось самостоятельно записывать даже не ассемблерный код, а шестнадцатеричные значения байтов машинного кода подпрограммы (значения которых можно было посмотреть в отладчике или в справочнике по командам процессора). Для реализации inline в Delphi в файл DCU включаются специальные блоки с представлением информации об абстрактном синтаксическом дереве тела подпрограммы (которое при компиляции вызывающей подпрограммы после подстановки фактических параметров вместо формальных подставляется в соответствующий вызову узел её абстрактного синтаксического дерева).

Впоследствии в Delphi 2009 в языке Object Pascal появилась возможность использования шаблонов (template) подпрограмм и классов. Для реализации этой возможности был использован тот же самый inline байт-код. Теперь уже в узлы абстрактного синтаксического дерева тела подпрограммы из шаблона происходят подстановки после конкретизации параметров этого шаблона. При этом, в отличие от inline-подпрограмм, для подпрограммы из шаблона вообще не генерируется машинный код до конкретизации параметров (зато потом он генерируется для каждой конкретизации шаблона, используемой в программе, но уже в тех модулях, где эти конкретизации шаблонов используются). Анализ inline байт-кода позволяет получить хорошее представление о начальной стадии работы компилятора. Сам этот байт-код является достаточно высокоуровневым представлением операторов подпрограммы и, поэтому, может быть использован для её декомпиляции.

## 1.6. Формат объектных файлов Delphi

Формат хранения объектных файлов Delphi является закрытым. Наиболее полная информация обо всех версиях этого формата может быть получена из неофициальной спецификации [55] на языке FlexT [56], являющей результатом исследования формата DCU. Программа DCU32INT [57] (DCU32 INTerface) выполняет разбор содержимого файлов DCU всех известных версий, почти полностью восстанавливая интерфейсные части модулей, а также восстанавливая текст реализации модулей, но при этом вместо операторов языка Паскаль выводит ассемблерный код.

В объектных файлах Delphi, в отличие от исполняемого файла в формате PE, программа оказывается более структурированной, например, выделены блоки памяти, соответствующие коду каждой процедуры; имеется информация о типах данных; может присутствовать отладочная информация. В общем виде формат файла DCU выглядит следующим образом: сначала идет небольшой заголовок, в котором содержится общая информация о файле, такая, как размер, время компиляции и т. д. После заголовка следует поток теговой информации (см. листинг 1.3). По своему назначению теги можно разделить на следующие группы:

- Описания использованных файлов с исходными текстами и объектных файлов.
- Описания используемых модулей.
- Списки импортируемых из этих модулей определений (типов данных, процедур, и т. д.).
- Описания определений (типов данных, процедур и функций, и т. д.) из данного модуля.

- Блок памяти, составленный из блоков памяти для кода процедур и функций, образов констант, и т. д.
- Информация для редактора связей (в какие места блока памяти и в каком виде необходимо занести адреса используемых там определений).
- Отладочная информация.

Листинг 1.3. Фрагменты результата разбора файла DCU по спецификации на FlexT

```
1 Block: CODE
2 No code detected. Block size: 00002512.
3 0000:Magic: ulong = 0F0000DF
4 0004:Hdr: TDCUHeader = (FileSize:00002512;
5   CompileTime:26.2.2015 17:10:52; Inf:98F1B4B4; b00:00; b02:02;
6   Tbl: (0:[12](Tag:drUnitFlags{Ц}; D:(Flags:#0; Priority:#1E)),
7     1:[15](Tag:drSrc{p};
8       D:(Name:CILOpCodeInfo.pas; D:(FT:4.4.2014 12:54:52; B:#0))),
9     2:[2D](Tag:drUnit{d};
10      D:(Name:CILCodes;
11        D:(Inf:98F1B4B4; X:00000000;
12          L: (
13            0:[3F](Tag:drImpType{f};
14              D:(Name:TCILCode; D:6A2161A0)),1:drStop1{c}))))),
15              ....
16    38:[674](Tag:drArrayDef{L};
17    D:(Base:(RTTISz:#0; Sz:#36C; hAddr:#0); B1:03; hDTNdx:#1;
18      hDTEl:#C)),39:[67C](Tag:drCBlock{l}; D:(Sz:#15D4; D:
19 0000:4C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |L...............|
20 0010:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
21 0020:50 00 00 00 04 00 00 00 00 00 00 00 00 00 00 00 |P...............|
22 0030:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
23    ...
24    40:[1C53](Tag:drFixUp{m};
25      D:(Sz:#1EC;
26        D: (0:(dOfs:#0; B1:06; N2:#2C),
```

```
27          1:(dOfs:#0; B1:01; N2:#2C),2:(dOfs:#20; B1:01; N2:#2C),
28          3:(dOfs:#8; B1:01; N2:#17),4:(dOfs:#4; B1:01; N2:#19),
29          5:(dOfs:#4; B1:01; N2:#1A),6:(dOfs:#4; B1:01; N2:#1B),
```

Из листинга 1.3 видно что в объектных файлах Delphi содержится большой объем дополнительной информации, такой как имена импортируемых модулей и используемых функций из них, имена переменных и их типы данных и т. д.

Но, несмотря на большой объем метаданных, содержащихся в объектном файле, ряд задач остается нерешенным. Например, для вызова виртуального метода нет адресной привязки к имени вызываемого метода.

Листинг 1.4. Вызов виртуального метода

```
1 MOV EAX,EBX {Self}
2 MOV ECX,DWORD PTR [EAX] {TBM}
3 CALL DWORD PTR [ECX+64] {Константа 64 - смещение в TBM данного метода}
```

Для вызова виртуального метода компилятор Delphi генерирует инструкции, которые выполняют следующую последовательность действий:

1. Загружает в регистр EAX адрес экземпляра объекта (Self), который всегда является первым аргументом метода и, поэтому, при использовании соглашения по вызову register передаётся через регистр EAX в 32-разрядном режиме.

2. Загружает в промежуточный регистр (в данном случае ECX) указатель на таблицу виртуальных методов (TBM) из экземпляра объекта.

3. Выполняет косвенный вызов соответствующего данному виртуальному методу элемента таблицы.

Для определения имени вызываемого метода необходимо проанализировать последовательность присвоения регистров и ячеек памяти, чтобы определить тип данных находящегося в регистре EAX операнда второй инструк-

ции и по нему определить ТВМ какого класса используется в третьей инструкции: тогда по смещению в таблице виртуальных методов получится извлечь имя вызываемой процедуры. Для решения такой задачи может быть использован один из методов анализа потоков данных — достигающие определения.

## 1.7. Виды программного кода, встречающиеся в объектных файлах Delphi

Файлы DCU технически можно отнести к объектным файлам, поскольку в дальнейшем с использованием редактора связей из них собирается загрузочный модуль. С другой стороны, файлы DCU содержат больше сведений, чем типичные объектные файлы: там кодируется вся информация, полученная компилятором из исходных текстов модуля, и, в том числе, информация об определенных в этом модуле типах данных. Файл DCU может полностью заменить исходный текст для той версии компилятора, при помощи которой он был создан. Этой особенностью активно пользуются разработчики программных модулей, которые часто распространяют их в формате DCU без предоставления исходных текстов, в особенности тогда, когда это делается на коммерческой основе. С каждой новой версией компилятора в формат DCU вносятся изменения, поэтому разработчикам приходится распространять файлы DCU для всех тех версий Delphi, для которых предназначен их модуль.

В том случае, когда разработчик прекращает развитие своих программных модулей, отсутствие исходных текстов не позволяет применить эти модули с новыми версиями компилятора. Также становится невозможным: исправить обнаруженные ошибки, проанализировать качество кода модуля, не говоря уже о его доработке.

Таким образом, задача исследования и, в идеале, декомпиляции фай-

лов DCU часто становится очень актуальной для тех разработчиков, которые используют файлы DCU без исходных текстов. При этом, поскольку в файлах DCU содержится почти вся информация, полученная компилятором из исходных текстов, эта задача должна быть разрешимой, но для ее решения необходимо разработать методы декомпиляции для тех видов программного кода, которые встречаются в файлах DCU (x86, x64, ARM 32, ARM 64, CIL, Inline).

Таблица 1.2. Достигнутый уровень декомпиляции

| **Платформа** | **Код** | **Версия** | **№** | **Уровень** |
|---|---|---|---|---|
| Win 32 | x86 | 2.0 – 7.0, 2005 – | 2 – 7,9 – | Дизассемблер |
| Win 64 | x64 | XE2 – | 16 – | Дизассемблер |
| OS X,32 | x86 | XE2 – | 16 – | Дизассемблер |
| iOS,Simulator | x86 | XE4 – | 18 – | Дизассемблер |
| iOS,Device | ARM 32 | XE4 – | 18 – | Нет |
| iOS,Device 64 | ARM 64 | XE8 – | 22 – | Нет |
| Android | ARM 32 | XE5 – | 19 – | Нет |
| .NET | CIL | 8.0 – 2006 | 8 – 10 | Декомпилятор |
| * | Inline | 2005 – | 9 – | Декомпилятор |

Следует отметить что, Delphi продолжает развиваться, и в настоящее время поддерживает компиляцию для наиболее распространенных платформ: Windows, OS X, iOS, Android. Кроме того, в ряде версий (8.0 – 2006) выполнялась компиляция для .NET. В таблице 1.2 представлены виды байт-кода встречающегося в объектных файлах Delphi и достигнутый к настоящему времени уровень его анализа.

## 1.8. Особенности декомпиляции объектных файлов Delphi

Несмотря на то, что задача декомпиляции в общем случае неразрешима из-за ряда алгоритмических трудностей, для определённых видов кода существуют инструменты способные производить качественный анализ, декомпиляцию. Такие инструменты в большинстве своем изначально разрабатывались с учетом языка на котором была написана программа. На практике все универсальные средства показывают плохой результат и в большинстве своем интересны только с теоретической точки зрения.

Высокоуровневые языки программирования одного типа предоставляют примерно схожие возможности. Однако, иногда существуют различия, и в таких случаях трансляторам из одного языка высокого уровня в другой приходится моделировать код исходного средствами целевого. В процессе такого преобразования могут появляться артефакты трансляции, которые существенно затрудняют анализ. Таким образом, задача декомпиляции становится не проще задачи трансляции.

Рассмотрим некотрые особенности языка Delphi и объектных файлов DCU, отличающие их от результатов компиляци других языков, предназначенных для разработки под .NET:

- В процессе обработки объектных модулей Delphi редактором связей теряется информация об именах локальных переменных. Такая информация зачастую является очень ценной, поскольку позволяет быстрее понять специалисту назначение переменной, избавляя от необходимости изучать весь исходный код.

- Язык Паскаль позволяет использовать вложенные процедуры и функции. Такая возможность не учитывается декомпиляторами, рассчитанными на код C#.

- Язык Delphi, поддерживая объектно-ориентированные типы данных,

позволят писать программы и в процедурном стиле (так же, как C++). Например в отличии от C# в нем присуствуют не только методы, но и процедуры и функции. К тому же функцию можно вызвать как процедуру.

- Delphi и C# поддерживают использование значений параметров по умолчанию. Вызовы методов с не заданными значениями таких параметров заменяются компилятором на значения по умолчанию, после чего эта информация становится не нужна, и уже не содержится в исполняемом файле. В объектных файлах Delphi такая информация содержится.

- В Delphi поддерживается два вида объектных типов данных: class и object. При компиляции в код виртуальной машины .NET эта информация также теряется, в отличии от объектного файла DCU.

Таким образом, имеет смысл рассматривать задачу анализа объектного кода, написанного на языке Delphi для платформы .NET, с целью получения более качественных исходных текстов по сравнению с результатами декомпиляторов исполняемых файлов .NET.

## 1.9. Выводы

Несмотря на то, что задача декомпиляции в общем случае неразрешима из-за ряда алгоритмических трудностей, в частном случае существуют инструменты способные производить качественный анализ, декомпиляцию. Такие инструменты в большинстве своем изначально разрабатывались с учетом языка на котором была написана программа. На практике все универсальные средства показывают плохой результат и в большинстве своем интересны только с теоретической точки зрения.

Таким образом, имеет смысл рассматривать задачу анализа объектного кода, написанного на языке Delphi для платформы .NET, с целью получения более качественных исходных текстов по сравнению с результатами декомпиляторов исполняемых файлов .NET.

# Глава 2

# Методы декомпиляции объектного кода Delphi

В данной главе описываются методы декомпиляции объектного кода Delphi.

## 2.1. Общая схема процесса декомпиляции объектного кода Delphi

На рис. 2.1 представлена схема процесса декомпиляции объектного кода Delphi.

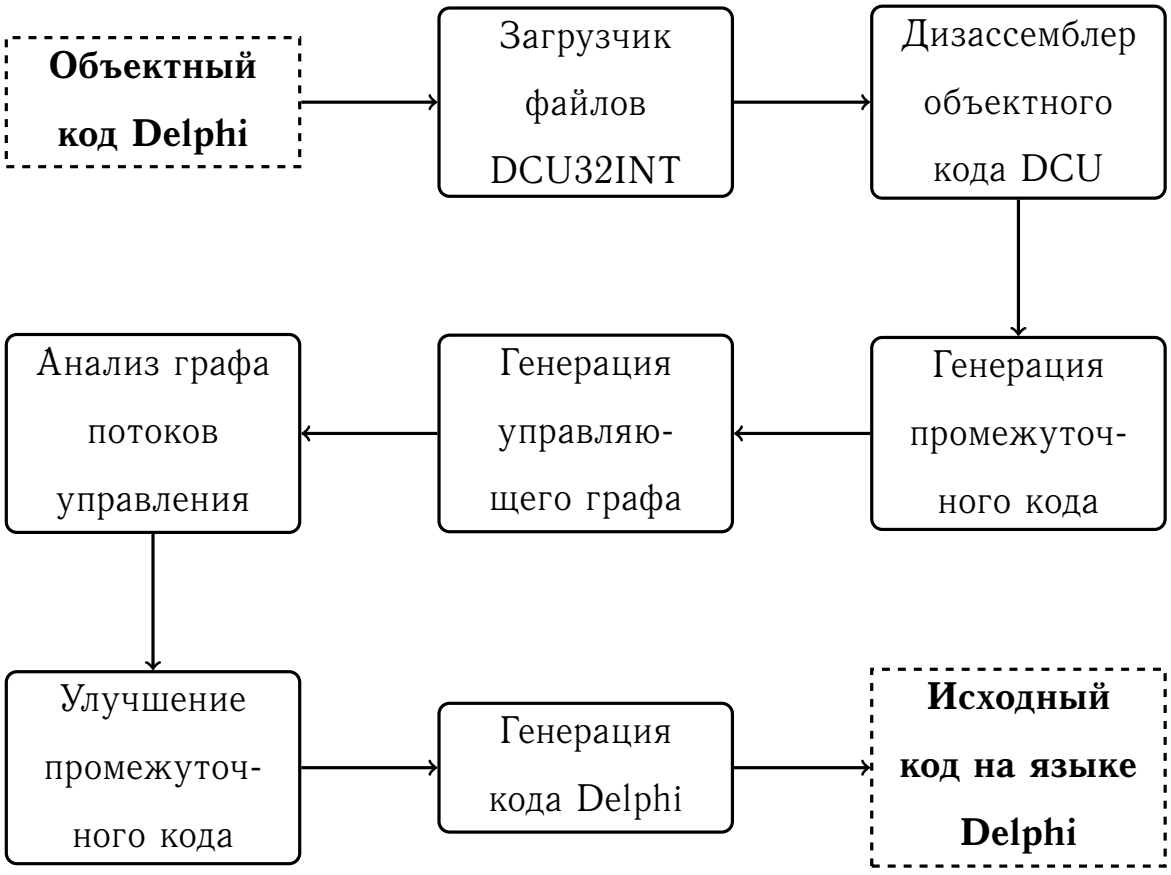



Рис. 2.1. Схема декомпиляции объектного кода Delphi

Декомпиляция кода в файлах DCU может выполняться отдельно для каждой подпрограммы, что существенно облегчает решение этой задачи.

Для реализации декомпилятора объектных файлов DCUIL необходимо решить следующие основные задачи (рис. 2.1): восстановление высокоуровневых операторов, реализация промежуточного представления, генерация кода и оптимизации, нацеленные на улучшение результата декомпиляции.

## 2.2. Лексический анализ байт-кода CIL

В декомпиляторе стадия лексического анализа сводится к сопоставлению последовательности байтов некоторого выражения, описывающего её семантику. Команда CIL представляет собой закодированное по определённым правилам [58] указание для виртуальной машины на выполнение некоторой операции. Команда всегда начинается с кода команды. Код команды может занимать от одного до двух байтов, у двухбайтовых кодов первый байт всегда буден равен `0xFE` (т.е. ряд команд закодирован в дополнительной таблице, т.к. они не поместились в основной).

Несмотря на то, что виртуальная машина .NET является стековой, большинство команд имеют встроенные операнды (см. таблицу 2.1), которые располагаются непосредственно за самой инструкцией.

Структура CIL команды довольна проста по сравнению с инструкциями x86 и имеет следующие особенности:

1. Код команды может состоять из одного или двух байтов.

2. После команды могут присутствовать метаданные.

Особое внимание стоит уделить операнду `FixUp`, поскольку он может содержать в себе адресную привязку как к обрабатываемому модулю, так и к импортируемому. В том случае, если `FixUp` ссылается на импортируемый модуль, его необходимо разобрать и сохранить ссылку для дальнейшего анализа декомпилятором. Иначе, результат декомпиляции будет некорректен из

| **Операнд** | **Размер** | **Описание** |
|---|---|---|
| `none` | 0 | Встроенный операнд отсутствует |
| `int8` | 1 | Знаковое 8-битовое целое число |
| `int32` | 4 | Знаковое 32-битовое целое число |
| `int64` | 8 | Знаковое 64-битовое целое число |
| `unsigned int8` | 1 | Беззнаковое 8-битовое целое число |
| `unsigned int16` | 2 | Беззнаковое 16-битовое целое число |
| `float32` | 4 | 32-битовое число с плавающей запятой |
| `float64` | 8 | 64-битовое число с плавающей запятой |
| `FixUp` | 4 | Адресная привязка |
| `switch` | переменный | Массив адресов переходов |

Таблица 2.1. Встроенные операнды CIL

за невозможности правильно интерпретировать работу со стеком (например, при вызове процедуры). Содержащаяся в `FixUp` информация может использоваться редактором связей для генерации метаданных.

Алгоритм разбора 1 считывает последовательно байты подпрограммы и ставит им в соответствие значение из таблиц инструкций опкодов B виртуальной машины.

**Алгоритм 1:** Алгоритм разбора блоков кода для процедур

**Исходные параметры**: Блок памяти, содержащий в себе байт-код для процедуры

**Результат**: Последовательность CIL команд

```
1  до тех пор, пока не конец кода выполнять
2      B ← ReadByte()
3      если b ≠ $FE тогда
4          ByteCode ← OneByteOpCodeTbl(B)
5      конец условия
6      иначе
7          B ← ReadByte()
8          ByteCode ← TwoByteOpCodeTbl(B)
9      конец условия
10     ByteCode.ReadOperand()
11 конец цикла
```

## 2.3. Промежуточное представление подпрограмм объектных файлов

Для более эффективного анализа программы декомпилятору, так же как и компилятору необходимо перевести программу в промежуточное представление. Оно служит промежуточным звеном между объектным кодом и результатом декомпиляции, над которым выполняются все промежуточные этапы. Промежуточное представление должно удовлетворять ряду требований:

- Оно должно быть достаточно простым для того, чтобы перевести в него исходную программу и сгенерировать код на целевом ЯВУ.
- Над ним должно быть легко проводить все необходимые операции в процессе декомпиляции.

Наиболее часто используемыми формами промежуточными представлениями являются ориентированный граф [59], трёхадресный код, префиксная и постфиксная запись [60, 61]. Одной из наиболее распространённых форм промежуточного представления является форма статического одиночного присваивания (Static Single Assignment, SSA). Применению данной формы в задаче декомпиляции посвящена докторская диссертация [9] Майка Ван Еммерика, результатом которой является декомпилятор Boomerang использующий форму SSA. Также данная форма представления используется в таких декомпиляторах, как dcc [7], REC, SmartDec [62], Hex-Rays и других.

### 2.3.1. Трёхадресный код

Трёхадресный код представляет собой последовательность операций вида $x \leftarrow y\ op\ z$, где $op$ – это бинарная операция. Все составные операции при построении такого промежуточного представления должны быть разбиты на подвыражения, при этом могут появляться дополнительные имена переменных. Например выражение $a \leftarrow b+c+d$ в трёхадресном представлении будет иметь следующий вид:

$t_1 \leftarrow b + c$

$a \leftarrow t_1 + d$

Такое представление делает трёхадресный код особенно подходящим для генерации целевого кода компиляторами из-за того, что практически все команды процессора также имеют, как правило, адрес назначения и два операнда инструкции. Обычно рассматриваются следующие виды трёхадресных команд:

### 2.3.2. Статическое единичное присваивание

Форма статического единичного присваивания (СЕП) – это промежуточное представление программы, которое значительно облегчает некоторые

| **Вид** | **Описание** |
|---|---|
| $x \leftarrow y\ op\ z$ | бинарная арифметическая операция |
| $x \leftarrow op\ y$ | унарная операция |
| $x \leftarrow y$ | команды копирования |
| $goto\ L$ | команды перехода |
| $if\ x\ goto\ L$, $if\ \neg\ x\ goto\ L$ | команды перехода по условию |
| $if\ x\ relop\ y\ goto\ L$ | переход с отношениями (<,==,>= и т. д.) |
| $x \leftarrow call\ p, n$ | вызов процедур и функций |
| $x[i] \leftarrow y$ | индексированные присваивания |
| $x \leftarrow \&y$ | присваивание адресов и указателей |

Таблица 2.2. Наиболее распространённые виды трёадресных команд

оптимизации проводимые компиляторм. В отличие от трёхадресного кода каждое новое значение переменной получает в СЕП уникальное имя. Если переменная может получить разные определения в разных путях потока управления, как в примере, показанном в листинге 2.1, то для выбора из этих двух определений используется $\phi$-функция [63], как показано в листинге 2.2.

Листинг 2.1. Фрагмент кода

```
1 if a > b then
2   Result:= a
3 else
4   Result:= b;
```

Листинг 2.2. СЕП представление

```
1 if a > b then
2   Result1:= a
3 else
4   Result2:= b;
5 Result3:= φ(Result1, Result2);
```

### 2.3.3. Ориентированный ациклический граф

Другой формой промежуточного представления является синтаксическое дерево программы. Ту же самую информацию, но в более компактной форме, дает представление в виде ориентированного ациклического графа. Здесь общие подвыражения объединены в одну вершину.

На рисунках ниже приведены представления для выражения $a \leftarrow b + c * b$ в виде синтаксического дерева рис. 2.2 и ориентированного ациклического графа рис. 2.3

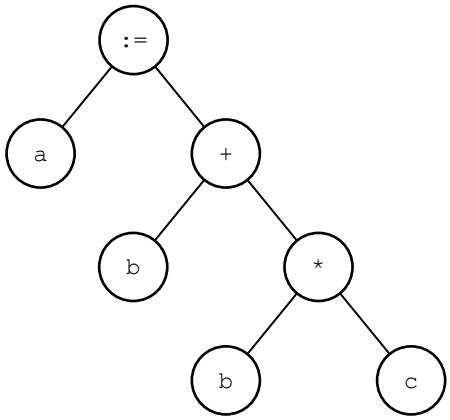



Рис. 2.2. Синтаксическое дерево

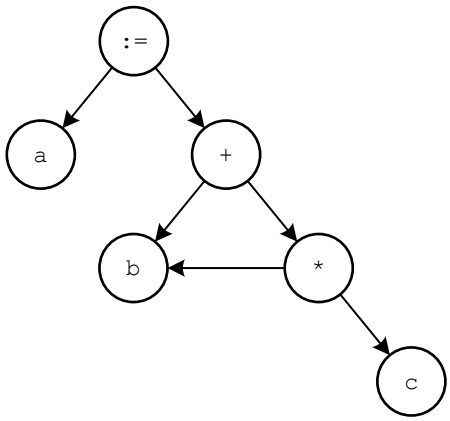



Рис. 2.3. Ориентированный ациклический граф

## 2.4. Генерация управляющего графа

**Определение 2.4.1.** *Ориентированный граф $G(X,U)$ называется графом потоков управления, если выполняются следующие условия:*

1) *граф $G$ не содержит параллельных дуг;*
2) *в множестве вершин графа выделена одна вершина $start$, которая является входом графа;*
3) *в множестве вершин графа выделена одна вершина $end$, которая является выходом графа;*
4) *каждая вершина $x \in X$ достижима из $start$;*
5) *каждая вершина $x \in X$ достигает выхода $end$;*

### 2.4.1. Базовые блоки

**Определение 2.4.1.** *Базовый блок (basic block) – это последовательность следующих друг за другом команд, обладающих приведенными ниже свойствами:*

1. *Поток управления может входить в базовый блок только через первую команду блока, т.е. переходы в середину блока отсутствуют.*
2. *Управление покидает все команды блока без ветвления и останова, за исключением последней команды, которая может быть командой перехода (как безусловного, так и условного) или выхода из подпрограммы.*

Алгоритм 2 для построения базовых блоков, представленный в [37], получает на вход последовательность трёхадресных команд. Далее в последовательности команд выделяются лидеры, которые разбивают её на линейные

подпрограммы, которые начинаются с команды-лидера. Язык Delphi поддерживает механизм обработки исключительных ситуаций, который требует отдельного внимания при построении базовых блоков для них. Для этих целей алгоритм приведенный ниже был дополнен шагом 4.

---

**Алгоритм 2:** Генерация базовых блоков.

---

**Исходные параметры**: Последовательность команд

**Результат**: Список базовых блоков для данной последовательности, в которой каждая команда принадлежит ровно одному базовому блоку

1. Первая команда кода является лидером
2. Любая команда, являющаяся целевой для условного или безусловного перехода, является лидером.
3. Любая команда, следующая непосредственно за условным или безусловным переходом, является лидером
4. Первая команда блока исключительной ситуации является лидером

Затем базовый блок каждого лидера определяется, как содержащий самого лидера и все команды до (но не включая) следующего лидера или до конца подпрограммы.

---

## 2.5. Дерево доминирующих вершин

**Определение 2.5.1.** *Узел $x$ является доминатором $y$ ($x\ dom\ y$) в направленном графе, если любой путь от $start$ до $y$ включает узел $x$.*

**Определение 2.5.2.** *Узел $x$ является постдоминатором $y$ ($x\ pdom\ y$), если любой путь от $y$ до $end$ включает $x$.*

**Определение 2.5.3.** *Узел $x$ является непосредственным доминатором $y$*

*($x$ $idom$ $y$), если $x$ $dom$ $y$, и не существует такого промежуточного узла $P$, что $x$ $dom$ $P$ и $P$ $dom$ $y$.*

**Определение 2.5.4.** *Узел $x$ является непосредственным постдоминатором $y$ ($x$ $pidom$ $y$), если $x$ $pdom$ $y$, и не существует такого промежуточного узла $P$, что $x$ $pdom$ $P$ и $P$ $pdom$ $y$.*

Задача вычисления доминаторов – известная задача, возникающая при анализе потоков данных и оптимизации программ компилятором. Еще одно полезное свойство дерева доминирования заключается в том, что, если в графе потоков управления все отступающие ребра являются обратными, то граф является сводимым.

Решение данной проблемы описано Евстигнеевым и Касьяновым [64], Ахо и Ульманом [65], Тарьяном и Ленгауэром [66], Дональдом Кнутом [67], Купером и Харви, Кеннеди [68]. Алгоритм вычисления дерева доминирующих вершин [68] имеет не самую лучшую теоретическую оценку сложности из рассмотренных, но на практике использование его является предпочтительным из-за ряда факторов:

- Он более прост в реализации по сравнению с остальными.
- Маленькая скрытая константа делает его более эффективным на графах менее 30 тысяч вершин.

**Алгоритм 3:** Итерационный алгоритм поиска доминаторов

**Исходные параметры**: Размеченный граф потока управления

**Результат**: Промежуточные доминаторы для каждого узла в графе

```
1  для каждого базового блока b выполнять
2  |  doms[b] ← ∅
3  конец цикла
4  doms[start] ← start
5  до тех пор, пока Changed выполнять
6  |  Changed ← false
7  |  для каждого b в обратном порядке, исключая start выполнять
8  |  |  doms[b] ← ∅
9  |  |  new_idom ← f  b
10 |  |  для каждого предшественника b, p исключая f выполнять
11 |  |  |  если doms[p] ≠ Undefined тогда
12 |  |  |  |  new_idom ← intersect(p, new_idom)
13 |  |  |  конец условия
14 |  |  |  если doms[b] ≠ idom тогда
15 |  |  |  |  doms[b] ← new_idom
16 |  |  |  |  Changed ← true
17 |  |  |  конец условия
18 |  |  конец цикла
19 |  конец цикла
20 конец цикла
```

Данный итерационный алгоритм продолжает свою работу до тех пор, пока вносятся изменяется хотя бы в один промежуточный доминатор для узла в управляющем графе. Таким образом, его вычислительная сложность равна произведению количества узлов в графе на длину самого большого цикла.

**Алгоритм 4:** Функция intersect

**Результат**: finger1

```
finger1 ← b1
finger2 ← b2
до тех пор, пока finger1 ≠ finger2 выполнять
    до тех пор, пока finger1 < finger2 выполнять
        finger1 ← doms[finger1]
    конец цикла
    до тех пор, пока finger1 < finger2 выполнять
        finger2 ← doms[finger2]
    конец цикла
конец цикла
```

## 2.6. Анализ потоков управления

Рассмотрим более подробно задачу восстановления высокоуровневых операторов. Большинство работ по структурированию уграфов направлены на удаление оператора неструктурного программирования `goto` путём введения новых логических переменных [69–74] или дублирования кода [70, 75, 76].

Лихтблау в своей работе [77] представил набор правил с помощью которых можно свести любой граф, тоесть в результате семантически эквивалентных преобразований привести его к одной абстрактной вершине. Каждый раз, когда правило неприменимо, удаляется дуга и вставляется инструкция безусловного перехода. На практике в большинстве работ посвященных декомпиляции используются два подхода к анализу потока управления отдельных процедур. Первый подход использует дерево доминаторов для поиска *естественных* циклов, и в дальнейшем использует их для оптимизации. Второй подход, называемый *интервальным анализом*, включает методы, которые позволяют анализировать структуру процедуры в целом и разбивать её

на вложенные участки, называемые *интервалами*. Теория интервалов была предложена Алленом [78] в начале 1970-х годов и использовалась для проведения оптимизаций при более тщательном анализе потоков данных. Наиболее глубокий вариант интервального анализа, называемый *структурным анализом*, был предложен Цифуентес [43]. Данный метод классифицирует абсолютно все структуры потока управления в процедуре. На первом этапе метод производит выделение и структурирование циклов. Далее в порядке обратном обходу в глубину на граф накладываются шаблоны, соответствующие высокоуровневым операторам, и с помощью семантически эквивалентных преобразований граф сводится к одной абстрактной вершине, которая содержит в себе всю иерархию вложенных интервалов. В теории компиляции методы анализа потока данных на основе анализа интервалов называются *методами устранения*.

Структурный анализ в задаче декомпиляции, как и в прямой задаче – компиляции основан на анализе графа потока управления. Граф потоков управления состоит из базовых блоков, объединяющих в себе инструкции которые гарантированно (с некоторыми оговорками [79]) выполняются последовательно. Дуги в графе соответствуют всем условным и безусловным командам передачи управления. На рис. 2.4 представлен пример графа потока управления.

После того, как построен граф потока управления, начинается поиск высокоуровневых операторов. В статье [80] рассматривается два метода анализа графа потока управления:

### 2.6.1. Анализ дерева доминирующих вершин

В этом методе по графу потоков управления строится дерево доминирующих вершин. Наиболее эффективный алгоритм построения дерева доминаторов был предложен в статье [68]. Также в этой работе проведен обзор

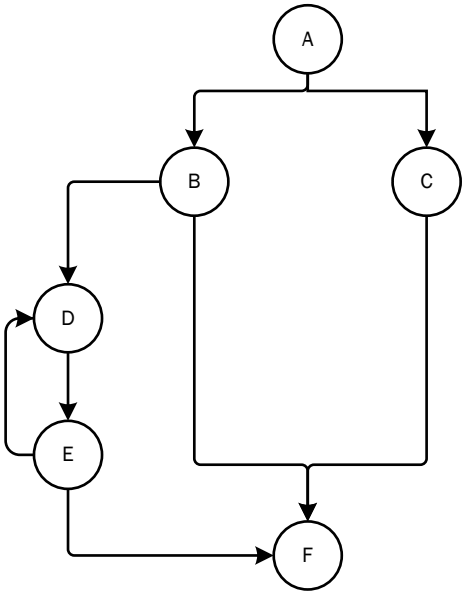



Рис. 2.4. Граф потока управления

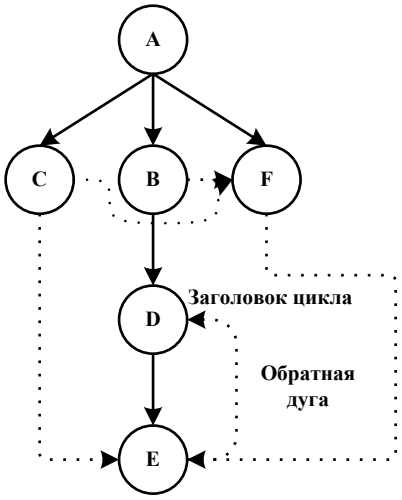



Рис. 2.5. Дерево доминаторов

альтернативных алгоритмов.

После того, как дерево доминаторов построено, на графе потока управления дуги помечаются как прямые, обратные и косые. Прямые дуги соединяют доминаторы и доминируемые ими вершины. На рис. 2.4 дуга, соединяющая узлы $A$ и $B$, является прямой, потому что узел $A$ доминирует над узлом $B$. Обратной дугой называется дуга, указывающая на предка, то есть узел, который был пройден раньше в процессе обхода графа потоков управления. Все оставшиеся дуги помечаются как косые.

Размеченный граф потока управления (рис. 2.5) позволяет выделить

только циклы. Так, обратной дуге $E \longrightarrow D$ соответствует цикл, состоящий из вершины $D$ и всех вершин доступных на пути из $D$ в $E$.

### 2.6.2. Интервальный анализ

Интервальный анализ – это одновременно название методов анализа потока данных и потока управления. Применительно к анализу потока управления интервальный анализ означает разбиение графа потока управления на области различного вида, называемые регионами, т.е. набор базовых блоков, имеющий не более одной входящей дуги. Простейшим случаем региона является базовый блок. На первой итерации работы алгоритма все базовые блоки помечаются, как самостоятельные регионы. Для выделения регионов строится дерево обхода графа потока управления в глубину. Вершины исследуются в порядке, обратном порядку их включения в дерево. Если два региона соединены только одной дугой, то они объединяются. Если вершина является входной точкой циклической или ациклической управляющей конструкции, то регион, соответствующий этой конструкции, выделяется в новую вершину, и соответствующим образом корректируются дуги. Тип конструкции определяется последовательным сравнением подграфов, включающих рассматриваемую вершину, с соответствующими шаблонами. Алгоритм заканчивает работу, когда преобразованный граф не содержит дуг и состоит только из одного региона.

Таким образом, в результате применения последовательности сворачивающих преобразований, строится управляющее дерево, определенное следующим образом:

- Корень управляющего дерева – это абстрактный граф, состоящий из одного узла и представляющий исходный граф.
- Листья управляющего дерева – это базовые блоки исходного графа.

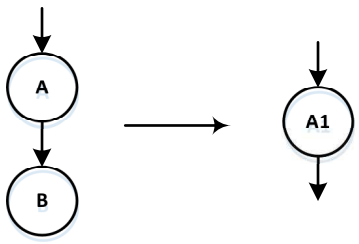



Рис. 2.6. T1 – преобразование

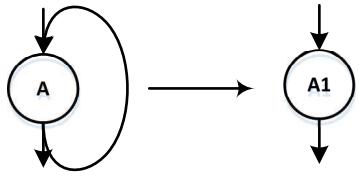



Рис. 2.7. T2 – преобразование

- Узлы, не являющиеся корнем или листьями – абстрактные узлы графа потока управления.
- Узлы $a_1, a_2, \ldots, a_n$ являются потомками узла $a_0$ тогда и только тогда, когда узел $a_0$ был получен сверткой из узлов $a_1, \ldots a_n$.

Самой простой формой интервального анализа является T1-T2 анализ. Он состоит всего из двух преобразований, показанных на рисунках 2.6 и 2.7.

## 2.7. Структурный анализ

В алгоритме структурного анализа [43] выделяются следующие шаблоны сводимых конструкций:

- шаблон `block` – линейная последовательность узлов графа;
- шаблон `if-then` – условный оператор если-то;
- шаблон `if-then-else` – условный оператор если-то-иначе;
- шаблон `switch` – оператор множественного выбора;
- шаблон `conditional` – для свертки сложных логических выражений;

- шаблон `self loop` - самоцикл – узел, зацикливающийся сам в себя;
- шаблон `natural loop` – цикл, заголовок которого доминирует над всеми его узлами;
- шаблон `while loop` – цикл с предусловием;
- шаблон, соответствующий несобственной области (improper region).

Существенно, что каждый из шаблонов имеет ровно одну входную дугу. Таким образом, например, несводимая область, помеченная данным алгоритмом, будет включать в себя ближайший общий доминатор всех ее входов. Алгоритм выделяет в графе потока управления структуры одного из этих типов, затем производит «свертку» графа, заменяя выделенную область новым абстрактным узлом, и перенаправляет входящие и исходящие дуги соответствующим образом.

Основные шаги алгоритма структурного анализа таковы:

**Алгоритм 5:** Алгоритм структурного анализа

1. Построение основного дерева поиска в глубину для графа потока управления.

2. Обратный обход вершин графа. При этом обходе алгоритм для каждой вершины пытается наложить на граф шаблон таким образом, чтобы рассматриваемая вершина была входной вершиной шаблона.

3. Если удалось наложить шаблон, то выделенная область сворачивается в абстрактный узел соответствующего типа, при этом входящие и исходящие дуги области перенаправляются соответственным образом. Иначе выделяется несобственная область.

4. Переход к шагу 2.

Процесс заканчивается тогда, когда количество узлов графа становится равным одному.

## 2.8. Алгоритм структурирования кода подпрограмм объектных файлов Delphi

Современные языки программирования в большинстве случаев поддерживаю стандартный набор высокоуровневых операторов (`if-then`, `if-then-else`, `for`, `while` и т. д.). Структурные конструкции в процессе компиляции порождают специфические только для них подграфы. Например, конструкция `if-then` породит фрагмент графа изображенного на рис. 2.9, где вместо блока `then` может находится другая управляющая конструкция.

Проанализировав все стандартные высокоуровневые операторы можно

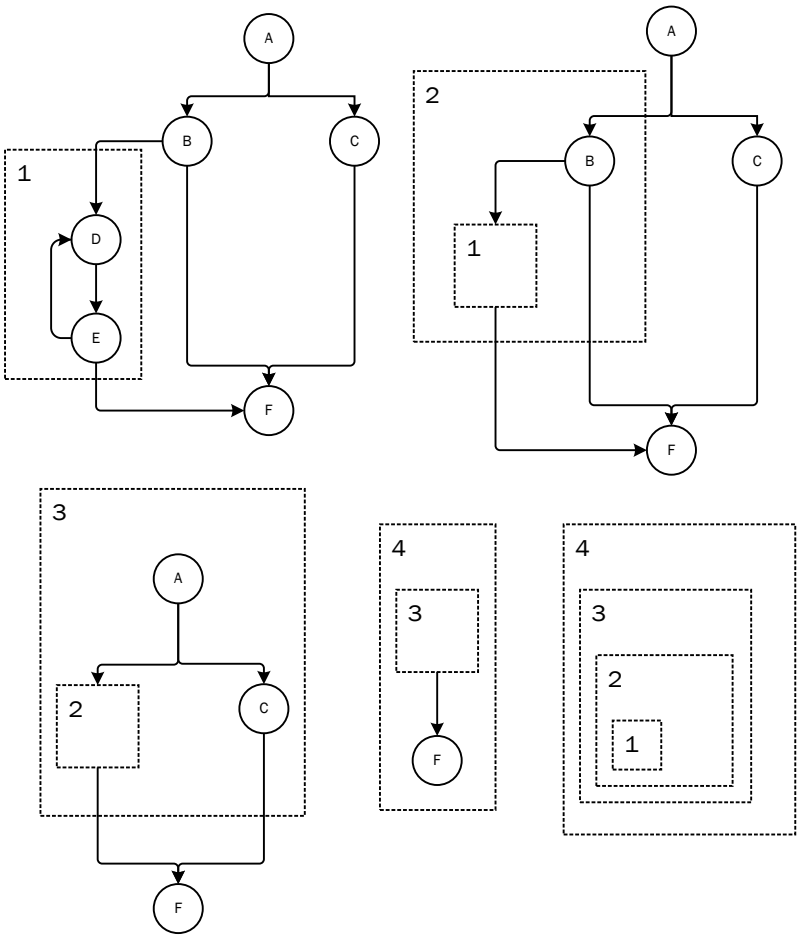



Рис. 2.8. Пример структурирования управляющего графа

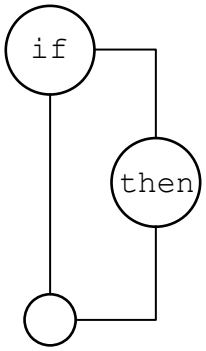



Рис. 2.9. Подграф для оператора if-then

для каждой управляющей конструкции описать шаблон порождаемого ею подграфа. Для ограниченного количества шаблонов можно выполнить поиск в управляющем графе подграфов, соответствующих одному из шаблонов. В случае обнаружения подграфа он заменяется новым абстрактным узлом, при этом входящие и исходящие дуги перенаправляются соответствующим образом. После этого процесс поиска шаблонов повторяется для полученного графа. Процесс поиска шаблонов может завершиться двумя способами. Во-первых, граф потоков управления может свернуться в одну абстрактную вершину, такие графы называются *сводимыми*. В противном случае может встретиться подграф, не соответствующий ни одному из заданных шаблонов. Такая область называется *неопределенной*, а весь граф – *несводимым*.

Синтаксис языков программирования, которые заставляют программиста придерживаться идеологии структурного программирования, позволяет писать только такие программы, управляющий граф которых всегда сводим. Данное утверждение верно и для большинства других языков, до тех пор, пока явно или неявно не будет использован оператор безусловной передачи управления. Существует два подхода к решению этой проблемы:

1. Избавление от несводимых областей. Существенным недостатком такого метода является необходимость добавления или удаления вершин и дуг в графе, изменяя семантику программы.

2. Выделение *несводимой* области в неопределенный регион и замена его на абстрактную вершину.

На основе анализа дерева доминирующих вершин для кода из объектных файлов Delphi разработан алгоритм структурирования управляющего графа, в основе которого лежит предложенный Джонсоном с коллегами в 1994 году метод [81] структурирования управляющего графа путем представления его в виде иерархии SESE-регионов (Single entry single exit). В

отличии от рассмотренного метода, вместо SESE регионов, предлагается выделяются регионы, имеющие одну входную и одну выходную вершину, которые в дальнейшем классифицируются и заменяются одной абстрактной вершиной. Свертка продолжается до тех пор, пока граф не преобразуется в одну абстрактную вершину, содержащую в себе всю иерархию вложенных программных структур. После завершения преобразования алгоритм производит рекурсивное развертывание абстрактных вершин, применяя заранее предопределенные шаблоны генерации кода в зависимости от типа обрабатываемого региона.

Следует отметить, что метод предложенный Джонсоном изначально разрабатывался для более эффективного анализа потоков данных в процессе компиляции, и в нём не предусматривается классификация выделяемых регионов.

Ниже приведен разработанный алгоритм структурирования:

**Алгоритм 6:** Алгоритм структурирования

**Исходные параметры**: G, D, P

**Результат**: Абстрактный узел, содержащий в себе иерархию вложенных регионов

```
1  для каждого v из D в обратном порядке выполнять
2      для каждого p ∈ (v) выполнять
3          если p pidom v тогда
4              S ← (v) \ p
5              C ← КлассифицироватьРегион(S)
6              если C ≠ неопределенный тогда
7                  НаложитьШаблон(C, S)
8              конец условия
9              иначе
10                 ВыделитьНеопределенныйРегион(S)
11             конец условия
12             Модифицировать(G, D, P)
13         конец условия
14     конец цикла
15 конец цикла
```

Построим дерево доминаторов $D$ и постдоминаторов $P$ для графа $G(E, V)$. Обход дерева доминаторов совершается снизу вверх, в порядке обратном обходу в ширину. Таким образом, на каждой итерации алгоритма выделяется TT-регион, имеющий наибольший уровень вложенности. Для этого используется правило $p\ pidom\ v$. Постдоминатор вершины $v$ из $S$ является терминальной вершиной, на которой сходится поток управления, прошедший через $v$. Эту вершину нельзя включать в регион потому, что в управляющем графе она может являться терминальной для другого региона.

После того, как выделено множество вершин гамака, он классифици-

руется. На выделенный подграф последовательно накладываются шаблоны, показанные на рис. 2.10. Если регион соответствует одному из этих шаблонов, то он считается *определённым*, иначе – *неопределенным*.

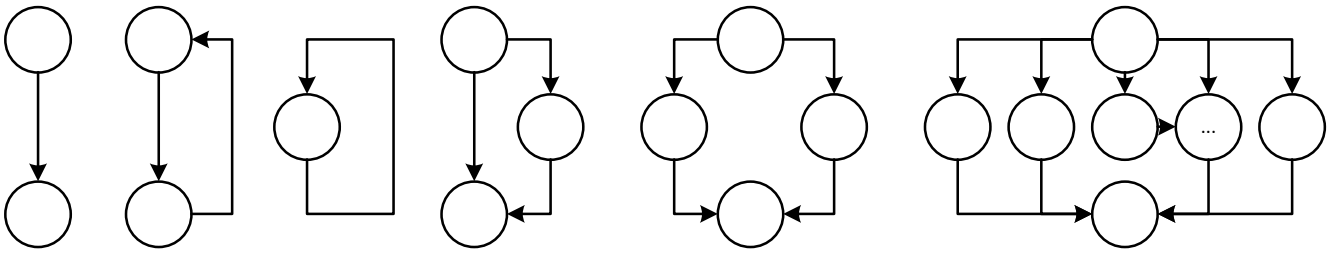



Рис. 2.10. Выделяемые шаблоны

Неопределенный регион также заменяется новым абстрактным узлом. В конечном счёте останется один абстрактный регион, не имеющий входящих и исходящих дуг, который содержит в себе всю иерархию вложенности подпрограммы.

## 2.9. Анализ потоков данных подпрограмм

Под анализом потоков данных понимают совокупность задач, нацеленных на выяснение некоторых глобальных свойств программы, то есть извлечение информации о поведении тех или иных конструкций в некотором контексте. Информация о потоке данных может быть собрана путем решения системы уравнений, связывающей информацию в разных точках программы. Типичное уравнение потока данных имеет вид:

$$out[S] = gen[S] \cup (in[S] - kill[S])$$

и может быть прочитано следующим образом: информация в конце инструкции либо генерируется в инструкции, либо приходит извне в начале инструкции и не уничтожается в процессе прохода по ней. Детали создания и решения уравнений потока данных зависят от трех факторов:

- Понятия уничтожения и генерации зависят от выбранной задачи анализа потоков данных. Кроме того, для решения некоторых задач вместо определения $out[S]$ через $in[S]$ требуется определить $in[S]$ через $out[S]$ (следовать в обратном направлении).

- На анализ потока данных влияют управляющие инструкции программы.

- Дополнительные трудности связаны с такими инструкциями, как вызов процедур и функций, присваивание посредством указателей и т. д.

### 2.9.1. Итерационный алгоритм для достигающих определений

Для сбора информации об операндах инструкций в потоке данных используется метод достигающих определений [82]. Компилятор Delphi генерирует код таким образом, что все обращения к памяти происходят в пределах адресного пространства данной процедуры, что позволяет производить анализ в рамках процедуры. Анализ потоков данных проводится на графе потока управления для подпрограммы, состоящем из базовых блоков. Базовый блок представляет собой последовательность инструкций с одним входом и одним выходом. То есть в рамках одного базового блока не может быть больше одной операции условного или безусловного перехода.

**Алгоритм 7:** Алгоритм анализа потоков данных

**Исходные параметры**: граф потока управления, в котором, для каждого блока $B$ вычислены множества $kill_B$ и $gen_B$

**Результат**: $IN[B]$ и $OUT[B]$, множества достигающих определений для входа и выхода каждого блока $B$ графа потока управления

```
1  OUT[Вход] ← ∅
2  для каждого базового блока B, отличного от входного
   выполнять
3  |  OUT[B] ← ∅
4  конец цикла
5  до тех пор, пока внесены изменения в OUT выполнять
6  |  для каждого базового блока B, отличного от входного
   |  выполнять
7  |  |  IN[B] ← ∪_{P−предшественник B} OUT[P];
8  |  |  OUT[B] ← gen_B ∪ (IN[B] − kill_B);
9  |  конец цикла
10 конец цикла
```

Первые три строки инициализирует некоторые значения потока данных. Внешний цикл выполняется до тех пор, пока вносятся изменения в выходное множество хотя бы одного базового блока. Внутренний цикл применяет уравнение потока данных к каждому базовому блоку, отличному от входного. Данный алгоритм распространяет выражения до тех пор, пока это возможно без их «уничтожения», моделируя все возможные пути выполнения программы.

Множества $kill_B$ и $gen_B$ базового блока $B$ получаются путем композиции передаточных функций содержащихся в нем инструкций. В Листинге 2.3

приведен пример вычисления множеств $kill$ и $gen$ для базового блока.

Листинг 2.3. Множество kill и gen для базового блока B

```
1 d_1: ADD  EDX ,EAX          gen_1 = {d_1}; kill_1 = {}
2 d_2: ADD  ECX ,EDX          gen_2 = {d_2}; kill_2 = {}
3 d_3: ADD  ECX ,34           gen_3 = {d_3}; kill_3 = {d_2}
4 d_4: ADD  ECX ,000000EA     gen_4 = {d_4}; kill_4 = {d_3}
5 d_5: ADD  EAX ,ECX          gen_5 = {d_5}; kill_5 = {}
6 d_6: SUB  EAX ,10           gen_6 = {d_6}; kill_6 = {d_5}
7 RET  NEAR
```

$kill_B = kill_1 \cup ... \cup kill_6 = d_2, d_3, d_5$ $gen_B = gen_6 \cup (gen_5 - kill_6) \cup (gen_4 - kill_5 - kill_6) \cup \ldots \cup (gen_1 - kill_1 - \ldots - kill_6) = d_1, d_4, d_6$

## 2.10. Методы генерации целевого кода

Перед генератором кода компилятора стоят такие задачи, как распределение регистров, упорядочение команд. Остальные задачи зависят от выбора промежуточного представления и целевого языка. В отличие от компилятора, декомпилятор решает обратную задачу, и процесс кодогенерации для него можно считать одной из самых простых подзадач.

Кодогенератору на вход поступает промежуточное представление программы, полученное в результате предыдущего анализа. Если учитывать тот факт, что нам известно каким компилятором была скомпилирована исходная программа, то задача генерации целевого кода ЯВУ сводится к определению соответствия между промежуточными структурами и их высокоуровневыми аналогами. В противном случае может возникнуть необходимость моделирования одних языковых конструкций средствами других. В таком случае задача декомпиляции равносильна задаче трансляции и в данной работе не рассматривается.

## 2.11. Выводы

Современные методы декомпиляции в большинстве своем основаны на методах построения компиляторов. Набор применяемых методов зависит от исходного и целевого ЯВУ. В данной главе рассмотрены основные методы, применимые к задаче анализа объектного кода Delphi. Рассмотрены методы и алгоритмы анализа потоков данных подпрограмм. На основе рассмотренных методов предложен алгоритм структурирования управляющего графа.

# Глава 3

# Инструментальное программное средство анализа объектного кода Delphi

В этой главе рассматриваются наиболее интересные детали реализации декомпилятора объектных файлов Delphi, скомпилированных под платформу .NET

## 3.1. Архитектура декомпилятора

На рис. 3.1 многоугольниками представлены компоненты декомпилятора, а стрелками отображается поток данных между ними. В архитектуре DCUIL2PAS можно выделить следующие составные блоки:

- *CILSeq* – это внутренне представление входной программы в виде последовательности CIL инструкций;
- Модуль *CILCtrlFlowGraph* выполняет построение графа потоков управления;
- Модуль *структурного анализа* восстанавливает высокоуровневые конструкции языка посредством модификации графа потоков управления;
- Модуль *промежуточного представления* отвечает за генерацию промежуточного кода;
- Модуль *оптимизаций* производит некоторые изменения промежуточного представления для улучшения качества генерируемого кода. Одной из важных оптимизаций является объединение условий операторов ветвления;

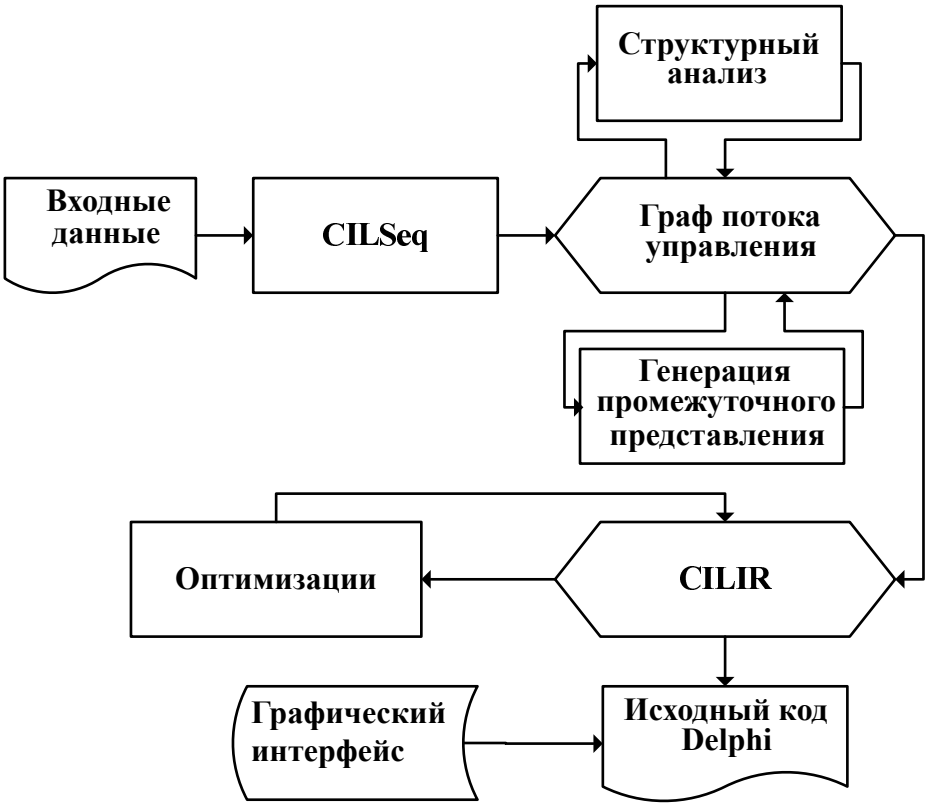



Рис. 3.1. Архитектура декомпилятора

- Модуль GUI реализует графический интерфейс пользователя с поддержкой подсветки синтаксиса.

На вход декомпилятору подаются файлы в формате DCUIL. С помощью функционала реализованного в программе DCU32INT [57] выделяются блоки кода для процедур и функций. С использованием реализованного на основе библиотеки Mono [83] дизассемблера генерируется байт-код CIL, соответствующий подпрограммам. Далее строится граф потока управления путем выделения базовых блоков и связей между ними. Для каждого базового блока строится промежуточное представление. Каждой инструкции ставится в соответствие выражение, полностью описывающее семантику кода. При этом производятся все необходимые операции со стеком, локальными переменными и параметрами подпрограммы. Далее с помощью методов структурного анализа и дерева доминирующих вершин производится структурный анализ. Полученное промежуточное представление подвергается некоторым оптимизациям, нацеленным на улучшения качества производимого кода. На выходе

декомпилятор производит код на языке Delphi семантически эквивалентный исходной программе.

## 3.2. Загрузчик файла

Загрузчик осуществляет разбор входного файла в формате DCUIL, DCU. В качестве загрузчика использована программа DCU32INT [57], которая выполняет разбор файла в соответствии со спецификацией формата DCU на языке FlexT [56]. Загрузчик считывает последовательность тегов в исходном файле и ставит им в соответствие структуры данных, описывающие прочитанные утверждения. В ходе чтения теговых структур данных для каждого файла DCU формируется две таблицы: таблица адресов и таблица типов. Большинство структур данных файла DCU ссылаются на другие структуры по индексам в этих таблицах. Описания подпрограмм содержат информацию о смещении соответствующего фрагмента в блоке памяти модуля и размере этого фрагмента.

За блоком памяти следует запись с таблицей перемещаемых адресов (`FixUp`), которая содержит информацию о том, в какие места блока памяти должны быть подставлены адреса различных процедур, переменных, описаний типов данных и других определений после их назначения редактором связей. Эта информация используется дизассемблером при разборе байт-кода для контроля правильности и отображения информации. Так, перемещаемые адреса могут встречаться только в операндах инструкций и не могут пересекаться с кодами команд. При наличии перемещаемого адреса в операнде информация об этом адресе используется при выводе операнда.

## 3.3. Процедура дизассемблирования

В программе DCU32INT реализован примитивный статический дизассемблер, который умеет:

1. Определять размер, занимаемый одной машинной командой;
2. Определять, что команда безвозвратно передаёт управление;
3. Обнаруживать ссылки из машинной команды (переходы на другие команды);
4. Отображать машинные команды.

Таких возможностей недостаточно для реализации полноценного декомпилятора, которому необходимо иметь всю информацию о семантике команды, доступную виртуальной машине исполняющей её.

Для получения семантики инструкций был использован проект Mono, реализованный на языке C#. Для этого был модифицирован декомпилятор ILSpy таким образом, чтобы на выходе он производил код на языке Delphi.

В результате декомпиляции части библиотеки Mono были получены две таблицы опкодов CIL:

- `OneByteOpCode` – опкоды длиной один байт
- `TwoBytesOpCode` – опкоды длиной два байта

Каждое значение в таблице представляет собой объект типа `TCILOpCode` (листинг 3.1), который содержит в себе всю необходимую информацию о семантике опкода:

- имя опкода
- информацию о типе передачи управления

- тип самого опкода
- типы операндов
- информацию о состоянии стека до выполнения команды и после

Листинг 3.1. Класс TCILOpCode

```
1  TCILOpCode = class
2  protected
3    FOp1 : Byte;
4    FOp2 : Byte;
5    FCode : TCILCode;
6    FFlowControl : TFlowControl;
7    FOpCodeType : TOpCodeType;
8    FOperandType : TOperandType;
9    FStackBehaviorPop : TStackBehaviour;
10   FStackBehaviorPush : TStackBehaviour;
11   function GetName: String;
12   function GetSize: Byte;
13 public
14   constructor Create(x: integer; y: integer);
15   destructor Destroy;
16   function GetVal: integer;
17   property Op1: Byte read FOp1;
18   property Op2: Byte read FOp2;
19   property Code: TCILCode read FCode;
20   property FlowControl: TFlowControl read FFlowControl;
21   property OpCodeType: TOpCodeType read FOpCodeType;
22   property OperandType: TOperandType read FOperandType;
23   property StackBehaviorPop: TStackBehaviour read FStackBehaviorPop;
24   property StackBehaviorPush: TStackBehaviour read FStackBehaviorPush;
25   property Name : String read GetName;
26   property Size : Byte read GetSize;
27 end;
```

В приложении Б представлена процедура дизассемблирования CIL кода.

## 3.4. Генерация выражений

Промежуточное представление CILIR (CIL Intermediate Representation) разработанное в декомпиляторе DCUIL2PAS реализовано в виде иерархии классов и строится для каждого базового блока графа потоков управления. Сначала определяется состояние перед началом исполнения для каждого блока. Начальное состояние характеризуется значениями параметров подпрограммы и локальных переменных, а также состоянием стека. Затем каждой инструкции CIL путём последовательного обхода линейного участка кода ставится в соответствие выражение (экземпляр класса), реализующее семантику опкода.

Поскольку переменная на линейном участке кода может иметь несколько вхождений в выражения, то при генерации новых выражений для экономии памяти каждая переменная снабжается счетчиком ссылок. Вначале счетчики всех переменных устанавливаются равными 1. При каждом вхождении переменной в выражение счетчик увеличивается на единицу. При переопределении переменной ей присваивается ссылка на новый объект, счетчик ссылок при этом снова устанавливается равным 1. В дальнейшем для вычисления результатов выражений используются ссылки на существующие объекты.

Иерархия классов, реализующая промежуточное представление изображена на рис 3.2. Все классы промежуточного представления наследуются от базового класса `TCILExpr`, который реализует логику работы со счетчиками ссылок и задает набор обязательных методов:

- `Eval` – вычисляет значение выражения;
- `Eq(E: TCILExpr)` – возвращает true если переданное выражение эквивалентно текущему;
- `AsString(BrRq: boolean)` – возвращает текстовое представление выражения;

- `Show` – выводит на печать текстовое представление выражения.

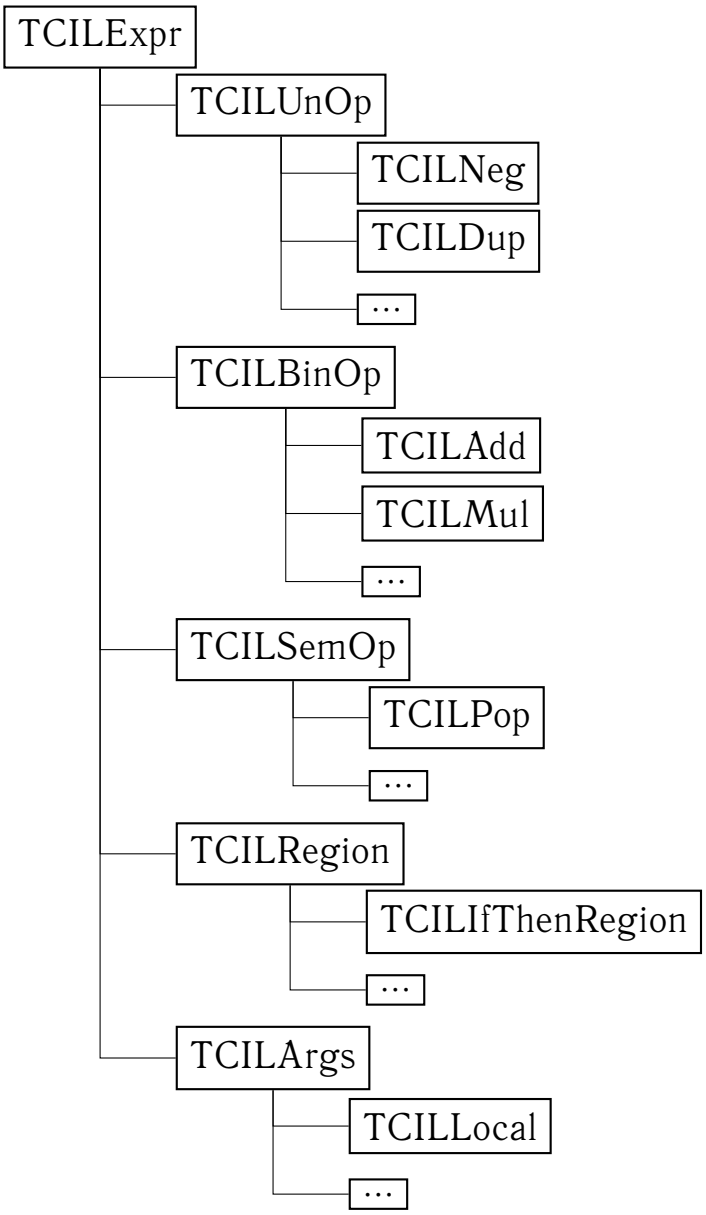



Рис. 3.2. Иерархия классов промежуточного представления

Наследники класса `TCILUnOp` описывают все «опкоды», аргументом которых является один операнд. Аналогично все наследники `TCILBinOP` описывают семантику всех бинарных операций. Аргументы и локальные переменные наследуются от класса `TCILArgs`. Для описания оставшихся инструкций в качестве базового используется класс `TCILSemOp`. Все регионы, выделяемые в процессе анализа потоков управления, также являются наследниками базового класса `TCILExpr`. Каждому из выделяемых регионов соответствует свой

класс, реализующий его семантику. Список классов, реализующих регионы в программе DCUIL2PAS: `TCILIfThenElse`, `TCILWhile`, `TCILRepeat`, `TCILCaseSt`, `TCILBasicBlack`. Все условные и безусловные переходы в процессе разбора преобразуются в выражение вида – `CILCond(Next, Target, Cond)`, где `Next` и `Target` – это блоки назначения перехода, `Cond` – условие перехода.

## 3.5. Генерация управляющего графа

Одной из наиболее важных задач декомпиляции является восстановление высокоуровневых операторов, таких как `if-then-else`, `if-then`, `while`, `for`, `case` и т. д. Для этих целей используется метод, рассмотренный в главе 2.

Для разбиения дизассемблируемой программы на базовые блоки используется информация о типе передачи управления, которая содержится в описывающей инструкцию структуре данных:

Листинг 3.2. Процедура разбиения на базовые блоки

```
1 procedure CmdPartRefs(CI: TCILOpCode; DP: Pointer; IP: Pointer);
2 var
3   Cnt: integer;
4 begin
5   with TCmdRefCtx(IP^) do begin
6     if CI.GetFlowControl = fcBranch then
7       Res := crJmp;
8     case CI.GetOperandType of
9      ShortInlineBrTarget: begin
10       if Res<0 then
11         Res := crJCond;
12       RegRef(CmdOfs+ShortInt(DP^),Res,IPRegRef);
13      end;
14     InlineBrTarget: begin
15       if Res<0 then
```

```
16          Res := crJCond;
17        RegRef(CmdOfs+LongInt(DP^),Res,IPRegRef);
18       end ;
19      InlineSwitch: begin
20        Res := crJCond;
21        Cnt := integer(DP^);
22        while Cnt>0 do begin
23          Inc(TIncPtr(DP),SizeOf(integer));
24          RegRef(CmdOfs+LongInt(DP^),Res,IPRegRef);
25          Dec(Cnt);
26        end;
27       end;
28     end;
29   end;
30 end;
```

Генерация управляющего графа совершается в один проход и объединена с процессом дизассемблирования. В качестве параметра в функцию разбора `ProcessCommand` из Приложения Б передаётся указатель на процедуру `CmdPartRefs`, которая производит разбиение блока памяти на последовательности инструкций (класс Д), которые соответствуют базовым блокам управляющего графа, и находит переходы между ними.

Особого внимания заслуживает обработка операторов обработки исключительных ситуаций. В Delphi для обработки исключительных ситуаций используется два оператора:

- `try/finally` – применяется когда необходимо, чтобы код в секции `finally` выполнился в любом случае.
- `try/except` – при возникновении исключительной ситуации исполнение основного фрагмент кода прекращается и выполнение передается в секцию `except`.

Для каждого блока кода подпрограммы, если в нём используются опе-

раторы обработки исключительных ситуаций, хранится специальная таблица с записями, структура которых приведена в Листинге 3.3. В ней содержится вся необходимая информация для их обработки: смещение до `try`; размер защищаемого блока кода; адрес и размер кода обработчика исключительных ситуаций; тип оператора (`finally`, `except`). В зависимости от рамера кода подпрограммы могут применяться более компактные версии записей `TMSILSmallExcClause` или менее компактные `TMSILFatExcClause`.

При генерации управляющего графа оператор `try/finally` обрабатывается достаточно просто: необходимо разбивать последовательность операторов на части, соответствующие блокам `try` и `finally`, в соответствии с информацией об адресах и размерах защищаемого блока и обработчика исключительных ситуаций. При обработке `try/except` необходимо сформировать новый базовый блок, поскольку поток управления в случае ошибки не достигает кода, следующего за оператором `except`.

Листинг 3.3. Структура для представления операторов исключительных ситуаций

```
1 PMSILSmallExcClause = ^TMSILSmallExcClause;
2 TMSILSmallExcClause = packed record
3   Flags: Word;
4   TryOffset: Word;
5   TryLength: Byte;
6   HandlerOffset: Word;
7   HandlerLength: Byte;
8   case Integer of
9     0:(ClassToken: LongInt);
10    1:(FilterOffset: LongInt);
11 end ;
12
13 PMSILFatExcClause = ^TMSILFatExcClause;
14 TMSILFatExcClause = packed record
15   Flags: LongInt;
16   TryOffset: LongInt;
```

```
17   TryLength: LongInt;
18   HandlerOffset: LongInt;
19   HandlerLength: LongInt;
20   case Integer of
21     0:(ClassToken: LongInt);
22     1:(FilterOffset: LongInt);
23 end ;
```

## 3.6. Восстановление высокоуровневых операторов

В декомпиляторе DCUIL2PAS реализован метод восстановления высокоуровневых конструкций рассмотренный в предыдущей главе с некоторыми модификациями.

Граф потока управления в процессе анализа перестраивается в семантически эквивалентный граф регионов. При построении графа потоков управления каждому узлу добавляется метка со счетчиком ссылок (количество ссылок соответствует числу входящих дуг). При этом все инструкции условного и безусловного перехода приводятся к единому виду:

- `if x op y then goto label` – переход по условию
- `goto label` – безусловный переход

Затем строится дерево доминаторов и постодминаторов. После этого выполняется итеративный алгоритм выделения регионов. Шаблоны, используемые при выделении регионов, соответствуют высокоуровневым конструкциям: `block`, `while`, `repeat`, `if-then`, `if-then-else`, `case`. Подграфы, соответствующие вышеназванным шаблонам, представлены на рис. 3.3 в порядке их упоминания.

Для наложения шаблонов выполняется обход дерева доминаторов в порядке, обратном обходу в ширину. Если рассматриваемый узел соответствует

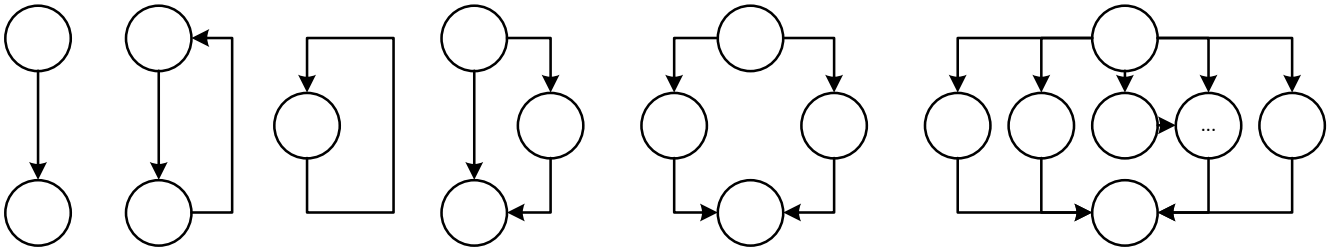


Рис. 3.3. Подграфы выделяемых регионов

одному из шаблонов, то все базовые блоки соответствующего подграфа выделяются в новый регион. Для каждого блока подграфа, в зависимости от его вида, уменьшаются счетчики ссылок для меток и удаляются выражения условных и безусловных переходов.

## 3.7. Декомпиляция вызовов процедур и функций

Для вызова процедур и функций в виртуальной машине NET. предусмотрено три команды: `Call, Calli, Callvirt`. В общем случае вызов метода на языке CIL выглядит следующим образом:

**Алгоритм 8:** Алгоритм вызова метода на языке CIL.

1. Аргументы метода помещаются на стек.
2. Точка входа метода помещается на стек.
3. Аргументы метода и точка входа извлекаются из стека; выполняется вызов метода;
   результат возвращается вызывающему коду.
4. Результат помещается на стек.

Для определения фактических параметров метода необходимо иметь его описание и корректным образом интерпретировать работу со стеком в соответствии с алгоритмом 8.

В листинге 3.4 приведен пример обработки инструкции `Call`. Здесь сна-

чала извлекается определение метода из текущего или импортируемого модуля. Далее в зависимости от его типа и количества параметров происходит интерпретация работы со стеком, так, как это делает виртуальная машина. В конечном счете создаётся выражение, которое содержит в себе информацию о имени метода и его аргументах.

Листинг 3.4. Фрагмент кода для обработки инструкции Call

```
Call: begin
  D := TUnit(FixUnit).GetGlobalAddrDef(Instr.Fix, U);
  OldU:= CurUnit;
  if D = nil then
    D:= TUnit(FixUnit).GetAddrDef(Instr.Fix)
  else
    CurUnit:= U;
  if D.ClassType = TTypeDecl then begin
    CallExpr:= TCILCall.Create(TTypeDecl(D).Name^.GetStr,nil);
    FInCtx.CILStack.PushExpr(CallExpr);
  end;
  if D.ClassType = TProcDecl then begin
    ProcArgs:= TList.Create;
    Args:= TProcDecl(D).Args;
    while Args<>nil do begin
      ProcArgs.Add(FInCtx.CILStack.PopExpr);
      Args:= TNameDecl(Args.Next);
    end;
    CallExpr:= TCILCall.Create(TProcDecl(D).Name^.GetStr,ProcArgs);
    if TProcDecl(D).IsProc then
      Instr.FExpr:= Call;
    FInCtx.CILStack.PushExpr(IfSt);
  end;
  CurUnit:= OldU;
end;
```

В языке Delphi можно вызывать функцию как процедуру, не используя возвращаемый результат. В этом случае компилятором генерируется код, ко-

торый после вызова функции удаляет со стека результат её вычисления. Если после команды вызова функции следует команда очистки вершины стека, то вызов функции происходит в форме вызова процедуры.

## 3.8. Оптимизация кода

Для логических выражений, включающих логические связки `and` и `or`, компилятор может генерировать код для операторов условного перехода двумя разными способами:

- Полное вычисление выражений. Условие преобразуется в последовательность инструкций вычисления выражения, помещающих вычисленный результат на стек для последующего извлечения в качестве аргумента для опкода условного перехода.

- Сокращённое вычисление логических выражений (short-circuit boolean evaluation). С помощью условных выражений, основанных на следующих правилах:

  1. $A\ and\ B \Rightarrow if\ A\ then\ B\ else\ False$,
  2. $A\ or\ B \Rightarrow if\ A\ then\ True\ else\ B$.

  Этот режим используется компилятором по умолчанию и позволяет не выполнять часть вычислений, если становится известно, что они уже не могут повлиять на результирующее значение выражения.

Случай полного вычисления логических выражений является наиболее простым для декомпилятора, поскольку результатом вычисления выражения, извлеченного со стека в качестве операнда команды условного перехода, будет исходное логическое условие.

При декомпиляции логического выражения, вычисляемого сокращённым способом, выполняется объединение логических условий более простых логи-

ческих выражений по заранее определённому набору правил. Например, выражение `IfThenElse(Cond, IfThenElse(Cond1, lblTrue1, lblFalse), lblFalse)` будет приведено к виду `IfThenElse(Cond and Cond1, lblTrue1, lblFalse)`. А в случае, если вложенное условное выражение содержится в ветке `else`: `IfThenElse(Cond, lblTrue, IfThenElse( Cond1, lblTrue, lblFalse))`, результирующее выражение примет следующий вид: `IfThenElse(Cond or Cond1, lblTrue, lblFalse)`. Аналогичным образом объединение происходит для условий циклов. Так, выражение вида `WhileSt(Cond, WhileSt (Cond1,Trg))` будет приведено к виду `WhileSt(Cond and Cond1, Trg)`. Например, промежуточное представление для кода:

Листинг 3.5. Исходный код функции

```
1 procedure  TWinForm.Dispose (Disposing: Borland.Delphi.System.Boolean);
2   overload ;
3 begin [Flags:3013,MaxStack:2,CodeSz:1E,LocalVarSigTok:0]
4   if (Disposing <> 0) then
5     if (TWinForm.Components <> 0) then
6       Container.Dispose(TWinForm.Components);
7   Form.Dispose(Self, Disposing);
8   Result := Form.Dispose(Self, Disposing);
9 end ;
```

будет выглядеть следующим образом:

```
1 procedure  TWinForm.Dispose (Disposing: Borland.Delphi.System.Boolean);
2   overload ;
3 begin [Flags:3013,MaxStack:2,CodeSz:1E,LocalVarSigTok:0]
4   if (Disposing <> 0) and (TWinForm.Components <> 0) then
5     Container.Dispose(TWinForm.Components);
6   Form.Dispose(Self, Disposing);
7   Result := Form.Dispose(Self, Disposing);
8 end ;
```

## 3.9. Генерация кода

В программе DCUIL2PAS используется технология размеченных потоков вывода. Вывод результата поддерживается в слежующих форматах:

- RTF – межплатформенный формат хранения размеченных текстовых документов
- text – неразмеченный текст
- HTML – язык гипертекстовой разметки
- LATEX – язык компьютерной вёрстки TeX

Во всех случаях используется одна процедура записи результата декомпиляции в размеченный блок памяти. Для вывода результата в графический интерфейс используется компонент Delphi TRichEdit, который поддерживает разметку текста формата RTF.

В текущей версии DCUIL2PAS для вывода кода в графический интерфейс поддерживаются следующие виды разметки:

- rtfKeyWord;
- rtfEnd;
- rtfRem;
- rtfCloseRem;
- rtfClose;
- rtfStrConst;
- rtfOpName;
- rtfNL;

- rtfRemOpen;
- rftRemClose;
- siOpName;
- siRemOpen;
- siRemClose;

Целевой код Delphi генерируется рекурсивно из промежуточного представления. В конечном счете результатом анализа программы является абстрактный узел, который содержит в себе всю иерархию программы. Каждый экземпляр класса промежуточного представления является наследником класса `TCILExpr` у которого определены методы:

- `function AsString(BrRq: boolean): String; virtual;` – возвращает исходный код на языке Delphi.
- `procedure Show(BrRq: boolean); virtual;` – выводит на печать исходный код на языке Delphi в заранее предопределенный поток вывода.

Ниже приведен пример реализации генерации кода для `TCILIfThenBlock`, которое соответствует высокоуровнему оператору `if-then`:

```
1 procedure TCILIfThenBlock.Show(BrRq: boolean);
2 begin
3   PutKW('if'); {печать оператора if}
4   if (FCond=nil) then
5     PutKW('false')
6   else
7     PutS(FCond.AsString(false));
8   PutSpace;
9   PutKW('then');
10  PutSpace;
```

```
11   if (FTrue.LinesCnt>1) then {если более 1 строки, печать begin}
12     PutKW('begin');
13   PutSpace;
14   ShiftNLOfs(2); {табуляция}
15   FTrue.Show;
16   ShiftNLOfs(-2);NL;
17   if (FTrue.LinesCnt>1) then
18     PutKW('end');
19 end;
```

Сначала выводится ключевое слово оператора условного перехода `if`. Далее для выражения, в котором хранится условие перехода, вызывается метод `AsString()` с параметром `false` (круглые скобки не нужны). Следует отметить, что условие не обязательно должно быть простым. Так как у всех выражений промежуточного представления все параметры являются наследниками класса `TCILExpr`, метод `Show` или `AsString` вызывается рекурсивно до тех пор, пока не будет сгенерирован код для всего выражения полностью.

## 3.10. Описание пользовательского интерфейса

Разработанный декомпилятор имеет графический пользовательский интерфейс 3.4 с подсветкой синтаксиса.

1. Строка заголовка – верхнее обрамление окна декомпилятора, на котором отображается название программы и располагается выпадающие списки режимов декомпиляции и главное меню управления:

   - Process File – обработать файл
   - Process Folder – обработать директорию
   - Save as – сохранить как
   - Exit – выход из программы

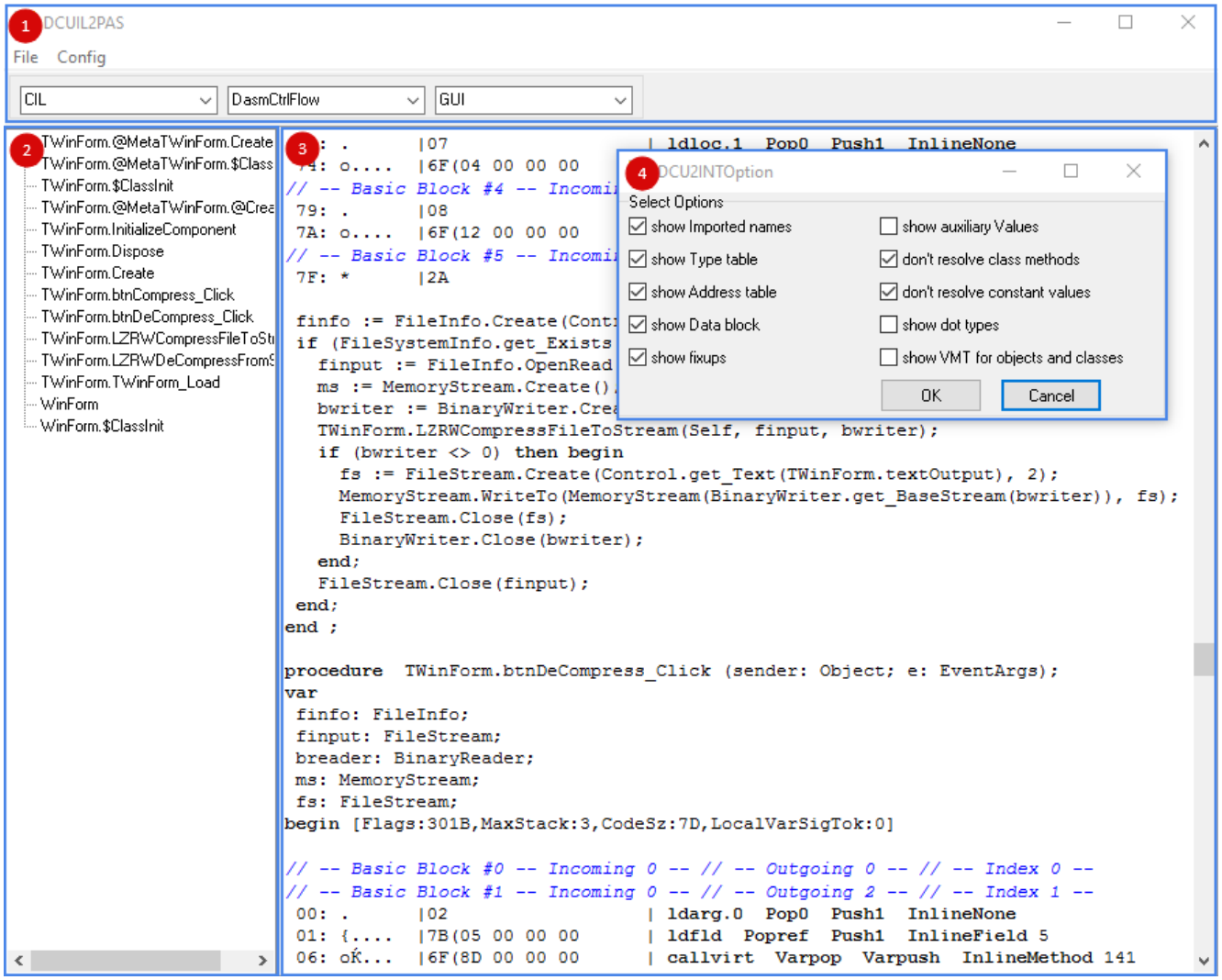



Рис. 3.4. Графический пользовательский интерфейс

2. Левая секция: список процедур, функций, методов обрабатываемого модуля.

3. Правая секция: форма отображения результата декомпиляции.

4. Настройки: форма дополнительных настроек вывода исходного кода:

   - show Imported Names – отображать импортируемые имена
   - show Type table – отображать таблицу типов
   - show Address table – отображать таблицу адресов
   - show Data block – выводить на печать блок данных
   - show fixups – отображать адресные привязки

- show auxiliary Values – выводить на печать дополнительные значения
- show don't resolve class methods
- show don't resolve constant values
- show dot types
- show VMT for objects and classes – выводить на печать таблицу виртуальных методов

## 3.11. Пример использования разработанного декомпилятора

Для демонстрации результатов работы декомпилятора DCUIL2PAS рассмотрим пример переноса программы архиватора, представленной в виде объектных модулей Delphi, скомпилированных под платформу .NET на платформу 80x86.

Программа сжатия включает в себя три файла, которые необходимо восстановить:

- WinForm.dcuil – содержит главную форму с графическим интерфейсом;
- LZRW1_EIS.dcuil – основной алгоритм сжатия;
- bitwiseclass.dcuil – вспомогательные функции.

Для облегчения процесса переноса необходимо в настройках, перед декомпиляцией выключить все дополнительные опции. В таком случае из вывода результата будет исключена вся вспомогательная информация, которая требуется только специалисту для более глубокого анализа кода.

С одной стороны декомпилятор генерирует код, который чаще всего компилируется без дополнительных (в рамках одной процедуры) изменений. С

другой стороны компилятор Delphi в процессе компиляции генерирует дополнительные служебные методы, которые необходимо в результате разбора для повторной компиляции удалить. Например для файла WinForm.dcuil компилятором сгенерированы методы, представленные на листинге 3.6. Все методы, которые были добавлены компилятором начинаются с символа @. Таким образом их достаточно просто исключить из результирующего файла, что можно сделать, как в ручном, так и в автоматическом режиме.

Листинг 3.6. Методы сгенерированные компилятором

```
1 constructor  TWinForm.@MetaTWinForm.Create;
2 begin [Flags:3013,MaxStack:2,CodeSz:12,LocalVarSigTok:0]
3  @TClass.Create(Self);
4  @TClass.FInstanceTypeHandle := TWinForm;
5  Result := @TClass.Create(Self);
6 end ;
7
8 procedure  TWinForm.@MetaTWinForm.$ClassInit; overload ;
9 begin [Flags:3013,MaxStack:1,CodeSz:15,LocalVarSigTok:0]
10  TWinForm.@MetaTWinForm+1 := TWinForm.@MetaTWinForm.Create();
11  Result := RuntimeHelpers();
12 end ;
13
14 procedure  TWinForm.$ClassInit;
15 begin [Flags:3013,MaxStack:1,CodeSz:B,LocalVarSigTok:0]
16  RuntimeHelpers.RunClassConstructor($Unit)
17  Result := RuntimeHelpers.RunClassConstructor($Unit);
18 end ;
19
20 function  TWinForm.@MetaTWinForm.@Create: TObject;
21 var
22  Result: TObject;
23 begin [Flags:3013,MaxStack:1,CodeSz:8,LocalVarSigTok:0]
24  Result := TWinForm.$ClassInit();
25  Result := Result;
```

```
26 end;
```

Обращение к полям визуальных компонентов Delphi .NET происходит через вызов метода `get_Text` класса `Control` из библиотеки .NET, как показано в примере 3.7 в 9 строке. В качестве параметра в метод `get_Text` передается объект `textInput`, который возвращает текст поля данного экземпляра класса `TEdit`. Такую конструкцию генерирует компилятор и её необходимо заменить на строку вида: `finfo := FileInfo.Create(textInput.Text);`. В строке 14 в методе `LZRWCompressFileToStream(Self, finput, bwriter)` необходимо удалить параметр `Self`, который добавляется компилятором.

Листинг 3.7. Процедура обработки нажатия на кнопку

```
 1 procedure TWinForm.btnCompress_Click(sender: Object; e: EventArgs);
 2 var
 3   finfo: FileInfo;
 4   finput: FileStream;
 5   bwriter: BinaryWriter;
 6   ms: MemoryStream;
 7   fs: FileStream;
 8 begin [Flags:3013,MaxStack:3,CodeSz:80,LocalVarSigTok:0]
 9   finfo := FileInfo.Create(Control.get_Text(TWinForm.textInput));
10   if (FileSystemInfo.get_Exists(finfo) <> nil) then begin
11     finput := FileInfo.OpenRead(finfo);
12     ms := MemoryStream.Create();
13     bwriter := BinaryWriter.Create(ms);
14     TWinForm.LZRWCompressFileToStream(Self, finput, bwriter)
15     if (bwriter <> nil) then begin
16       fs := FileStream.Create(Control.get_Text(
17                       TWinForm.textOutput), 2);
18       MemoryStream.WriteTo(MemoryStream(
19                   BinaryWriter.get_BaseStream(bwriter)), fs);
20       FileStream.Close(fs);
21       BinaryWriter.Close(bwriter);
22     end;
```

```
23     FileStream.Close(finput);
24   end;
25 end ;
```

Специальным образом необходимо обработать ключевое слово `inherited`, которое позволяет вызвать метод родительского класса. Компилятор Delphi в таком случае генерирует код, который обращается к одноимённому методу родительского класса и передаёт в него те же параметры, что и у вызывающего метода.

В том случае, если код не удалось полностью структурировать, могут появляться операторы безусловного перехода `goto`. Наличие таких операторов нежелательно в исходном коде, но при этом в Delphi они допустимы, и содержащая их программа будет корректно компилироваться. Для избавления от операторов `goto` чаще всего необходимо производить более существенные изменения исходного кода, которые требуют более тщательного анализа и понимания алгоритма работы исходного кода.

После всех вышеописанных действий над результатом декомпиляции, полученный код модуля (приложение E) может быть откомпилирован любой версией компилятора Delphi, при условии наличия всех необходимых модулей, которые в нём подключены.

## 3.12. Пример декомпиляции функции

Для демонстрации результатов работы декомпилятора DCUIL2PAS рассмотрим следующий небольшой пример. В листинге 3.8 представлен результат разбора функции, полученный с помощью программы DCU32INT:

Листинг 3.8. Код функции CmpIf на языке CIL

```
1
2 function  CMpIf (n: Integer; b: Integer): Integer;
3 var
```

```
 4   Result: Integer;
 5   i: Integer;
 6 begin [Flags:3013,MaxStack:3,CodeSz:37,LocalVarSigTok:0]
 7   00: .  |16     | ldc_i4_0
 8   01: .  |0A     | stloc_0
 9   02: .  |03     | ldarg_1
10   03: .  |1B     | ldc_i4_5
11   04: ю. |FE 02  | cgt
12   06: .  |02     | ldarg_0
13   07: .  |1D     | ldc_i4_7
14   08: ю. |FE 02  | cgt
15   0A: a  |61     | xor
16   0B: ,. |2C 06  | brfalse_s  $13
17   0D: .  |06     | ldloc_0
18   0E: .  |1B     | ldc_i4_5
19   0F: X  |58     | add
20   10: .  |0A     | stloc_0
21   11: +. |2B 04  | br_s  $17
22   13: .  |06     | ldloc_0
23   14: .  |18     | ldc_i4_2
24   15: X  |58     | add
25   16: .  |0A     | stloc_0
26   17: .  |03     | ldarg_1
27   18: .. |1F 0F  | ldc_i4_s  15
28   1A: /. |2F 06  | bge_s  $22
29   1C: .  |06     | ldloc_0
30   1D: .  |1B     | ldc_i4_5
31   1E: X  |58     | add
32   1F: .  |0A     | stloc_0
33   20: +. |2B 04  | br_s  $26
34   22: .  |06     | ldloc_0
35   23: .  |18     | ldc_i4_2
36   24: X  |58     | add
37   25: .  |0A     | stloc_0
38   26: .  |03     | ldarg_1
```

```
27: .< |1F 3C  | ldc_i4_s  60
29: /. |2F 06  | bge_s  $31
2B: .  |06     | ldloc_0
2C: .  |1B     | ldc_i4_5
2D: X  |58     | add
2E: .  |0A     | stloc_0
2F: +. |2B 04  | br_s  $35
31: .  |06     | ldloc_0
32: .  |18     | ldc_i4_2
33: X  |58     | add
34: .  |0A     | stloc_0
35: .  |06     | ldloc_0
36: *  |2A     | ret
end;
```

Результат разбора в таком виде не предоставляет необходимого уровня абстракции для его анализа специалистом за приемлемое время и трудозатраты. Для того, чтобы провести анализ такого кода, необходимо знать состояние стека, значения переменных и аргументов функции в каждый момент выполнения программы. Также необходимо знать семантику всех представленных опкодов. В листинге 3.12 приведен результат декомпиляции функции из листинга 3.8. Результат разбора в таком виде семантически эквивалентен исходному коду, не содержит операторов неструктурных переходов и представляет собой код на языке высокого уровня Delphi.

```
function  CMpIf (n: Integer; b: Integer): Integer;
var
  Result: Integer;
  i: Integer;
begin
  Result := 0;
  if (((b > 5) xor (n > 7)) <> 0) then
    Result := Result + 5
  else
```

```
10      Result := Result + 2;
11    if (b < 15) then
12      Result := Result + 5
13    else
14      Result := Result + 2;
15    if (b < 60) then
16      Result := Result + 5
17    else
18      Result := Result + 2;
19  end;
```

## 3.13. Результаты тестирования

Разработанный декомпилятор был протестирован на специально подготовленном наборе процедур, взятых из реализации алгоритма LZW [84] на языке Delphi. Для сравнения был выбран декомпилятор с открытым исходным кодом ILSpy. Для оценки качества использовалась *мера качества декомпиляции* [19], которая выражается формулой 3.1.

$$C_{decom} = \sum_{prog \in TS} \frac{max(0, K^{'} - K)}{KLOC(prog)} \tag{3.1}$$

Здесь

- $TS$ – тестовый набор программ
- $prog$ – это исходная программа
- $KLOC(prog)$ – количество тысяч значимых строк кода программы prog
- $K$ – сумма штрафов исходной программы
- $K^{'}$ – сумма штрафов восстановленной исходной программы

Штрафы за артефакты трансляции и неполноту восстановления (таб. 3.1) были изменены в соответствии с требованиями декомпиляции объектного

кода Delphi. Подсчет меры качества производился для каждой процедуры отдельно, при этом не учитывалось качество восстановления интерфейсной части модуля.

Таблица 3.1. Штрафы за артефакты трансляции и неполноту восстановления

| **Конструкции программы** | **Назначаемые штрафы** |
|---|---|
| невосстановление имени переменной | 1 |
| оператор перехода `goto` | 3 |
| оператор выхода из середины цикла `break` | 1 |
| оператор прерывания цикла `continue` | 1 |
| невосстановление оператора `for` | 1 |

Вычисленные меры качества декомпиляции представлены в таблице 3.2. На всех примерах мера качества разработанного декомпилятора выше чем у ILSpy. Это связано в первую очередь с тем, что в процессе обработки объектных модулей редактором связей теряется часть информации об исходной программе, а также из-за того, что декомпилятор ILSpy изначально разрабатывался исходя из соображений, что исходная программа была написана не на языке Delphi.

Таблица 3.2. Мера качества

| **Название** | **DCUIL2PAS** | **ILSpy** |
|---|---|---|
| BitWise | 62,5 | 133,3 |
| Compression | 18,6 | 146 |
| LZRW1KHCompressor | 75 | 140 |
| GetMatch | 0 | 166,6 |

Помимо сравнительного анализа, декомпилятором в пакетном режиме была разобрана стандартная библиотека VCL Delphi 8. Результаты, которые в большей степени демонстрируют производительность работы данного де-

компилятора приведены в таблице 3.3.

Таблица 3.3. Результаты пакетной обработки стандартной библиотеки VCL Delphi 8 (производительность)

| **Название** | **Кол-во файлов** | **Размер (мб)** | **Время обработки (с)** |
|---|---|---|---|
| Delphi 8 VCL | 325 | 39 | 396 |

Для оценки качества декомпиляции стандартной библиотеки VCL Delphi 8 в автоматическом режиме было просчитано количество процедур и функций восстановленных в структурном виде (без использования оператора `goto`). Тестирование показало (см. таблицу 3.4), что в 98,7% случаях удается восстановить программу без операторов `goto`.

Таблица 3.4. Результаты пакетной обработки стандартной библиотеки VCL Delphi 8 (качество)

| **Название** | **Кол-во процедур** | **Без `goto`** | **С `goto`** | **%** |
|---|---|---|---|---|
| Delphi 8 VCL | 9003 | 8879 | 124 | 1,3 |

## 3.14. Выводы

Разработанный декомпилятор объектных файлов DCUIL позволяет восстанавливать исходный код на языке Delphi, который в большинстве случаев пригоден для дальнейшей его компиляции и полностью семантически эквивалентен исходному представлению программы. Данное программное средство позволяет существенно сократить время на решение задач, связанных с поддержкой и переработкой унаследованного и стороннего программного обеспечения, исходные тексты которого не предоставлялись или были утрачены; позволяет находить закладки в готовых модулях и компонентах Delphi под .NET; искать и исправлять ошибки.

На разработанный декомпилятор объектных файлов DCUIL получено авторское свидетельство о государственной регистрации программ для ЭВМ.

# Глава 4

# Применение методов декомпиляции в задаче визуализации управляющего графа

В программировании графы являются одной из основных структур данных. Управляющий граф – это естественное представление программы, которое может быть вычислено автоматически, как по исходному, так и по бинарному коду. Граф используется в качестве промежуточного представления программы компилятором для проведения внутренних оптимизирующих преобразований.

Исходный код в текстовом представлении содержит в себе всю необходимую информацию о поведении программы. Однако его анализ часто является сложной задачей, даже при том, что современные интегрированные среды разработки поддерживают интеллектуальные механизмы, значительно её упрощающие.

Для анализа бинарного кода используются специализированные программы – дизассемблеры и декомпиляторы. Анализ ассемблерного кода – сложная и трудоемкая задача, требующая от специалиста обширных знаний. В то же время декомпиляция произвольного исполняемого файла не всегда возможна, и в некоторых случаях восстановленный код более труден для восприятия, чем ассемблерный.

Альтернативным способом является анализ визуального представления потока управления программы. В настоящее время существует большой выбор универсальных систем визуальной обработки графовых моделей, таких как uDraw (adaVinci) [85], VCG [86], Graphlet [87], GraVis [88], Graph Drawing Server [89], graphViz [90], VisualGraph [91]. Несмотря на то, что таких систем достаточно много, все они обладают недостатками. Например, применительно к задаче визуализации графа потоков управления подобные

системы не учитывают особенности и характерные черты таких графов.

Исходя из вышесказанного, визуализация атрибутивных графовых моделей является актуальной задачей и требует отдельного рассмотрения с учётом специфики природы их возникновения.

В работе рассматривается вопрос применения методов структурного анализа [20] графа потоков управления к задаче визуализации, представляющих его графовых моделей.

## 4.1. Критерии качества визуализации графа потоков управления

**Определение 1.** *Раскладка графа на плоскости (или в пространстве) — это отображение вершин и ребер графа в множество точек плоскости (или пространства) [92].*

Один и тот же граф можно визуализировать разными способами (рис. 4.1), причем качество одного и того же изображения может оцениваться по разному в зависимости от характера использования и вида отображаемой информации. Основным критерием оценки качества визуализации является соответствие изображения природе возникновения информации. Помимо этого для определения качества визуализации выделяют такие понятия, как изобразительное соглашение, эстетичность и ограничения [92]:

- *Изобразительное соглашение* – это одно из основных правил, которому должно удовлетворять изображение графа, чтобы быть допустимым.
- *Эстетические критерии* определяют такие свойства изображений, которые желательно применять в наибольшей степени, насколько это возможно, чтобы повысить наглядность изображения.

- *Ограничения.* Если соглашения и эстетические критерии формулируются по отношению ко всему графу и его изображению, то ограничения относятся к отдельным подграфам и частям изображений.

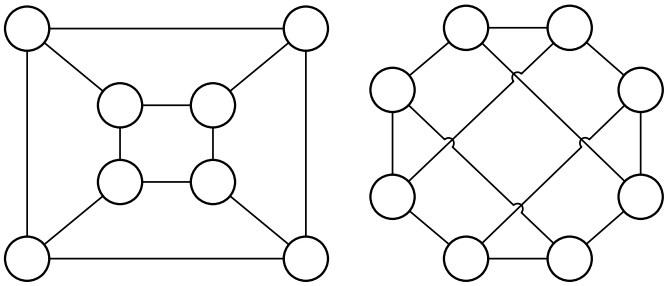


Рис. 4.1. Разные способы визуализации одного и того же графа

В управляющем графе, восстановленном из бинарного кода, сохраняется информация о разбиении на базовые блоки, начальной вершине (входе) и непустом множестве конечных вершин (выходах), а также информация о возможных путях передачи управления между базовыми блоками. Естественным визуальным представлением управляющего графа является его изображение в виде диаграммы потоков управления (блок-схемы). Исходя из данного соображения определим следующие изобразительные соглашения для визуализации узлов такого графа (рис. 4.2), где:

а) **Блок действия.** Представляет собой узел, передача управления из которого осуществляется только в одном направлении.

б) **Логический блок (блок условия).** Соответствует условному оператору в высокоуровневых языках, а в графе – узлу расхождения потока управления.

в) **Граница цикла.** Состоит из двух частей, обозначающих начало и конец операций, выполняемых внутри цикла.

г) **Блок начало-конец (пуск-остановка).** Отображает вход и выход в подпрограмму.

а) блок действия б) логический блок в) границы цикла г) начало-конец

Рис. 4.2. Типы элементов.

Основной задачей данной части работы является разработка алгоритма визуализации управляющего графа с критериями визуализации, принятыми для изображения блок-схем.

## 4.2. Метод поуровнего изображения графов

Алгоритмы визуализации графов разрабатываются с начала 60-х годов [93]. Классическими в данной области считаются работы Eades P. [94], Kamada T. и Kawai S. [95] В данных работах рассматриваются универсальные алгоритмы визуализации графов. Так как управляющий граф является направленным, рассмотрим способы визуализации такого класса графов.

Метод поуровнего изображения направленных графов, предложенный в работе [94], является основным для изображения данного класса графов. Как правило этот метод состоит из 4 шагов:

1. **Распределение вершин по уровням.** Каждой вершине присваивается ее ранг. При этом все дуги могут следовать только от меньшего ранга к большему. Между вершинами одного ранга не может быть дуг. Для распределения рангов вершин могут использоваться различные методы, в простейшем случае в качестве ранга может быть использована длина пути до вершины при обходе в глубину.

2. **Определение порядка вершин на уровнях.** Вершины упорядочиваются внутри уровня таких образом, чтобы минимизировать количество

пересечений дуг. Самым распространенным способом решения этой задачи является «метод медиан» [96].

3. **Определение координат вершин на уровне.** На каждом уровне каждой вершине присваиваются координаты таким образом, чтобы граф соответствовал определённым для него эстетическим критериям.

4. **Проведение дуг.** Обычно, эту задачу выделяют в отдельный этап только в том случае, когда дуги изображаются не в виде прямых.

## 4.3. Визуализация графов потоков управления

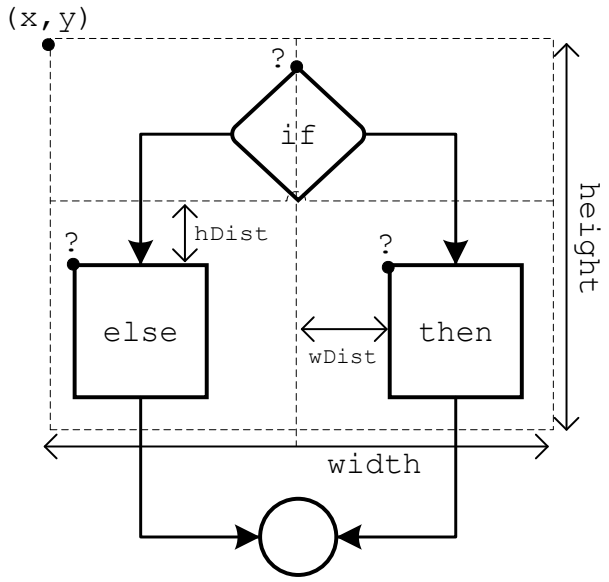



Рис. 4.3. Шаблон If-Then-Else

### 4.3.1. Алгоритм структурирования

### 4.3.2. Процесс раскладки

Процесс раскладки происходит рекурсивно сверху вниз, начиная с региона верхнего уровня. Для этого региона задаются начальные координа-

**Алгоритм 9:** Алгоритм структурирования

**Исходные параметры**: G, D, P

**Результат**: Абстрактный узел, содержащий в себе иерархию вложенных регионов

```
для каждого v из D в обратном порядке выполнять
    для каждого p ∈ (v) выполнять
        если p pidom v тогда
            S ← (v) \ p
            C ← КлассифицироватьРегион(S)
            если C ≠ неопределенный тогда
                НаложитьШаблон(C, S)
            конец условия
            иначе
                ПрименитьИерархическийРаскладчик(S)
            конец условия
            Модифицировать(G, D, P)
        конец условия
    конец цикла
конец цикла
```

ты. Далее, если у региона есть шаблон, применяются правила отображения, определенные в нем. Приведем пример для шаблона `if-then-else` (рис. 4.4).

Определим правила вычисления координат для узлов `if, then, else`:

- `if.x = x + abs(width - if.width) / 2;`
- `then.x = x + wDist / 2`
- `then.y = y + if.height + hDist`
- `else.x = x - else.width - wDist / 2`

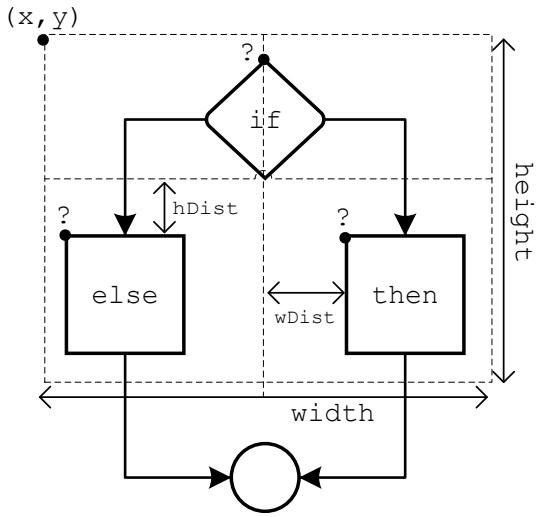



Рис. 4.4. Шаблон отображения if-then-else

- `else.y = y + if.height + hDist;`

Где `wDist` – вертикальное расстояние между узлами, `hDist` – горизонтальное. Данные параметры задаются вручную исходя из эстетических соображений.

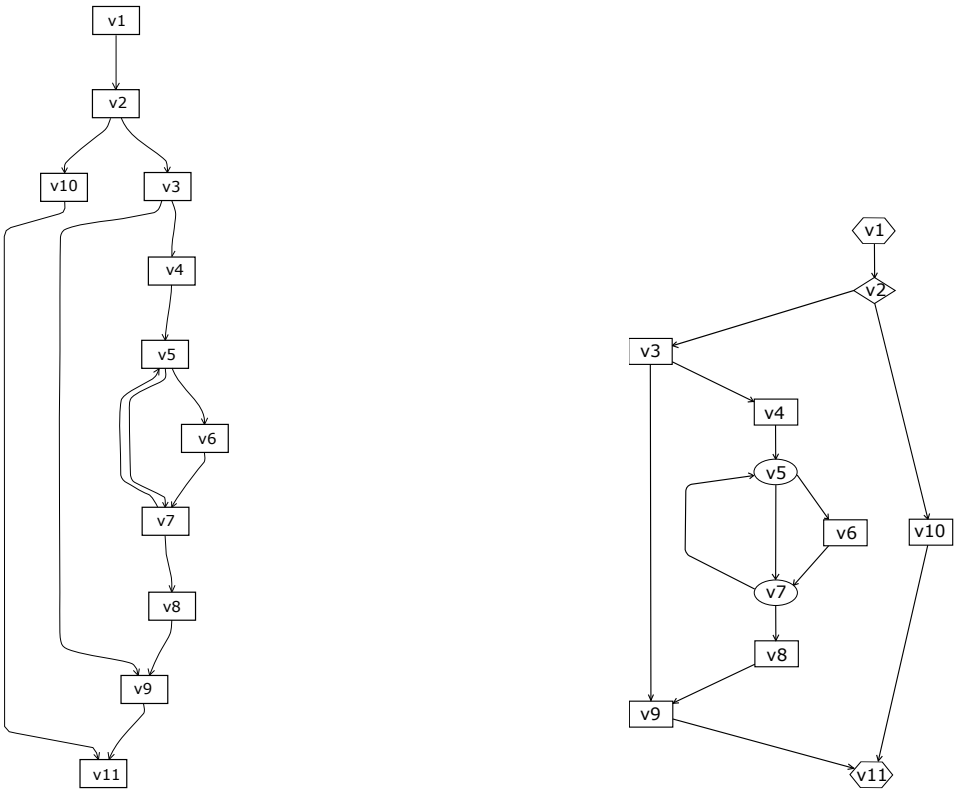
Для каждого шаблона аналогичным образом определены правила визуализации. Для неопределенных регионов используется иерархический раскладчик. Таким образом процесс раскладки сводится к последовательному применению правил отображения для вложенных регионов.

## 4.4. Реализация алгоритмов визуализации и их тестирование

Алгоритм реализован на объектно-ориентированном языке Java. На этом языке разработана библиотека JGraphX [97] и система визуализации иерархических графовых моделей VisualGraph, которая её использует. В JGraphX поддержаны популярные алгоритмы визуализации графов. Библиотека предоставляет возможности визуализации узлов и дуг графа, позволяет задавать форму и цвет узла, а также позволяет рисовать дуги с помощью задания узлов и угловых точек.

В текущей реализации частично поддерживаются изобразительные соглашения определенные для узлов в разделе 4.1. Для классификации узлов в регионах ветвления необходимо, чтобы граф был атрибутивным и содержал в себе ассемблерный код базовых блоков (линейных участков кода). Также для этого необходимо проанализировать семантику программы, представленной в виде управляющего графа. С другой стороны, анализ на основе дерева доминирующих вершин позволяет выделять циклы в графе, что позволяет специальным образом изображать обратные дуги.

Пример работы алгоритма и сравнение его с результатом раскладки, полученным при помощи иерархического раскладчика приведены на рис. 4.5: а) иерархический, б) структурный.




а) Иерархический раскладчик б) Структурный раскладчик

Рис. 4.5. Результат раскладки управляющего графа.

## 4.5. Выводы

Предложен новый подход к раскладке графов потоков управления на плоскости с использованием методов структурного анализа. На основе разработанных методов реализован структурный раскладчик атрибутивных графов потоков управления. Произведено тестирование структурного раскладчика на тестах SPEC CPU2000 [98]:

- 197.parser – синтаксический разбор для естественного языка;
- 252.eon – трассировка лучей.

В результате около 76% графов удалось структурировать полностью (не содержат «неопределенных» регионов). Около 96% всех выделенных регионов являются структурными.

Основными преимуществами данного раскладчика являются:

- Простота изменения правил визуализации графа посредством задания правил отображения шаблонов.
- Единообразное отображение подграфов, соответствующих одним и тем же операторам в высокоуровневых языках программирования.
- Возможность выделять специальным образом узлы и дуги графа, используя семантику, полученную в процессе структурирования графа.

Результаты тестирования показали, что данный раскладчик применим для графов с небольшим количеством вершин (не более 100). Для остальных графов достаточно велика вероятность появления неопределённого региона, который не соответствует изобразительным и эстетическим соглашениям принятым для отображения управляющих графов.

В качестве решения данной проблемы и дальнейшего развития темы исследования рассматриваются два варианта:

1. Дополнение базы распознаваемых шаблонов. Предлагается собрать статистику по всем неопределенным регионам и выявить среди них наиболее распространенные.

2. Приведение неопределенных регионов к определенным посредством поиска и удаления дуг в графе. Для этого необходимо в неопределенном регионе найти дуги, при удалении которых он станет определенным. В этом случае появляется нетривиальная задача визуализации удаленных дуг.

# Заключение

В диссертации исследована актуальная научная проблема – декомпиляция объектного кода Delphi. В рамках работы получены следующие результаты:

1. Создан декомпилятор объектных файлов Delphi скомпилированных под платформу .NET, позволяющий восстанавливать программы из низкоуровнего языка CIL в программы на языке Delphi. Применение данного программного средства позволяет сократить время решения задач, связанных с унаследованным программным обеспечением, а также повысить надежность и безопасность информационных систем.

2. Разработаны и предложены методы декомпиляции объектного кода Delphi, скомпилированного под платформу .NET, позволяющие восстанавливать код на языке низкого уровня CIL в язык высокого уровня Delphi.

3. Разработан оригинальный метод визуализации управляющего графа на плоскости. Основной особенностью данного метода является возможность использования изобразительных соглашений, принятых при рисовании блок-схем. Данный метод обладает более высоким уровнем абстракции по сравнению с принятыми для раскладки уграфов, что позволяет быстрее выделять наиболее интересные участки в графе для их более детального анализа.

Разработанные в рамках диссертационной работы методы и инструментальное средство позволяют значительно повысить эффективность, снизить трудозатраты и сократить сроки решения задач, связанных с анализом исполняемого кода. В частности, разработанное инструментальное средство может существенно снизить трудозатраты при решении задач связанных с поддерж-

кой унаследованного ПО, а так же при решении задач, связанных с информационной безопасностью.

# Список сокращений и условных обозначений

| | | |
|---|---|---|
| ЯВУ | – | язык высокого уровня |
| СЕП | – | статическое единичное присваивание |
| ЭВМ | – | электронно-вычислительная машина |
| ТТ-регион | – | двух терминальный регион |
| SSA | – | Static Single Assignment (СЕП) |
| CILIR | – | CIL Intermediate Representation |
| PE | – | Portable Executable |
| LLVM | – | Low Level Virtual Machine |
| PIT | – | Parameter Identification and Tracking |

# Словарь терминов

**pidom** — непосредственный постодминатор.

**idom** — непосредственный доминатор.

**pdom** — постдоминатор.

**dom** — доминатор.

# Список литературы


1. Halstead M. H. Machine-independent computer programming. Spartan Books, 1962.
2. Sassaman W. A. A computer program to translate machine language into fortran // Proceedings of the April 26-28, 1966, Spring joint computer conference / ACM. 1966. P. 235–239.
3. Hollander C. R. Decompilation of object programs.: Tech. rep.: STANFORD UNIV CALIF STANFORD ELECTRONICS LABS, 1973.
4. Friedman F. L. Decompilation and the transfer of mini-computer operating systems. 1974.
5. Schneider V., Winiger G. Translation grammars for compilation and decompilation // BIT Numerical Mathematics. 1974. Vol. 14, no. 1. P. 78–86.
6. Workman D. A. Language Design Using Decompilation.: Tech. rep.: UNIVERSITY OF CENTRAL FLORIDA ORLANDO DEPT OF COMPUTER SCIENCE, 1979.
7. Cifuentes C. Reverse Com pilation Techniques: Ph. D. thesis / QUEENSLAND UNIVERSITY OF TECHNOLOGY. 1994.
8. Mycroft A. Type-based decompilation // Lecture notes in computer science. 1999. P. 208–223.
9. Van Emmerik M. J. Static single assignment for decompilation: Ph. D. thesis / The University of Queensland. 2007.
10. Иванников В. П., Белеванцев А. А., Бородин А. Е. и др. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды Института системного программирования РАН. 2014. Т. 26, № 1.
11. Гайсарян С. С., Иванников В. П., Аветисян А. И. Анализ и трансформация программ // URL: http://www. ict. edu. ru/ft/005642/62319e1-st06. pdf. 2016.
12. Аветисян А. И., Бородин А. Е. Механизмы расширения системы стати-

ческого анализа Svace детекторами новых видов уязвимостей и критических ошибок // Труды Института системного программирования РАН. 2011. Т. 21.

13. Тихонов А. Ю., Аветисян А. И., Падарян В. А. Извлечение алгоритма из бинарного кода на основе динамического анализа // Труды XVII общероссийской научно-технической конференции Методы и технические средства обеспечения безопасности информации. 2008. С. 109.
14. Падарян В. А., Гетьман А. И., Соловьев М. А. Программная среда для динамического анализа бинарного кода // Труды Института системного программирования РАН. 2009. Т. 16.
15. Аветисян А. И. Двухэтапная компиляция для оптимизации и развертывания программ на языках общего назначения // Труды Института системного программирования РАН. 2012. Т. 22.
16. Аветисян А. И., Гетьман А. И. Восстановление структуры бинарных данных по трассам программ // Труды Института системного программирования РАН. 2012. Т. 22.
17. Гетьман А. И., Маркин Ю. В., Падарян В. А. и др. Восстановление формата данных // Труды Института системного программирования РАН. 2010. Т. 19.
18. Падарян В. А., Гетьман А. И., Соловьев М. А. и др. Методы и программные средства, поддерживающие комбинированный анализ бинарного кода // Труды Института системного программирования РАН. 2014. Т. 26, № 1.
19. Трошина Е. Н. Исследование и разработка методов декомпиляции программ: Кандидатская диссертация / Московский государственный университет имени М. В. Ломоносова. 2009.
20. Михайлов А. А. Анализ графа потоков управления в задаче декомпиляции подпрограмм объектных файлов dcuil // Вестник Новосибирского государственного университета. Серия: Информационные технологии.

2014. Т. 12, № 2.

21. Михайлов А. А. Промежуточное представление подпрограмм в задаче декомпиляции объектных файлов dcuil // Вестник Бурятского государственного университета. 2014. № 9-3.
22. Хмельнов А. Е., Бычков И. В., Михайлов А. А. Декларативный язык FlexT—инструмент анализа и документирования бинарных форматов данных // Труды института системного программирования РАН. 2016. Т. 28, № 5. С. 239–268.
23. Mikhailov A., Hmelnov A., Cherkashin E. et al. Control flow graph visualization in compiled software engineering // Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2016 39th International Convention on / IEEE. 2016. P. 1313–1317.
24. Михайлов А. А. Анализ объектных файлов Delphi с использованием спецификации семантики машинных команд // Прикладная дискретная математика. 2012. Т. 5. С. 108–110.
25. Михайлов А. А. Анализ потоков данных подпрограмм в объектных файлах dcu // Материалы конференции «Малые Винеровские чтения 2013». № 23 – 28. 2013.
26. Михайлов А. А. Анализ потоков данных подпрограмм в объектных файлах DCU // Тезисы конференции «ЛЯПУНОВСКИЕ ЧТЕНИЯ — 2012». № 24. 2012.
27. Михайлов А. А. Анализ потоков данных подпрограмм объектных файлов Delphi // Труды XVIII Байкальской Всероссийской конференции "Информационные и математические технологии в науке и управлении". Т. 2. 2013.
28. Михайлов А. А. Алгоритм анализа потоков данных подпрограмм объектных файлов DCU // Тезисы II Российско-Монгольской конференции молодых ученых — 2013. № 43. 2013.
29. Михайлов А. А. Визуализация управляющего графа // Тезисы доклада

III Российско-монгольской конференции молодых ученых по математическому моделированию, вычислительно-информационным технологиям и управлению Иркутск (Россия) - Ханх (Монголия). № 59. 2015.

30. Михайлов А. А., Хмельнов А. Е. Анализ программного кода в объектных файлах Delphi, скомпилированных под платформу .NET // Труды конференции «Языки программирования и компиляторы – 2017». № 202 – 204. 2017.
31. Михайлов А. А. Анализ программного кода объектных файлов Delphi с использованием спецификации семантики машинных команд. 2012. URL: http://conf.nsc.ru/ym2012/ru/reportview/139230 (дата обращения: 2015-01-19).
32. Hmelnov A. E., Mikhaylov A. A., Burlakov A. S. Delphi .NET object file decompiler // In Proc. of the 5th International Workshop on Computer Science and Engineering — Russia, Moscow: Bauman Moscow State Technical University. No. 202 – 208. 2015.
33. Burlakov A. S., Mikhaylov A. A. The Computer Architecture and Hardware Descriptive Language // In Proc. of the 5th International Workshop on Computer Science and Engineering — Russia, Moscow: Bauman Moscow State Technical University. No. 148 – 154. 2015.
34. Михайлов А. А., Хмельнов А. Е. DCUIL2PAS - декомпилятор объектных модулей Delphi, скомпилированных по платформу .NET (файлов *.DCUIL). 2014. Свидетельство о государственной регистрации программ для ЭВМ № 2014617137 М.: Федеральная служба по интеллектуальной собственности, патентам и товарным знакам.
35. Михайлов А. А., Хмельнов А. Е. Модуль структурной раскладки графов потоков управления на плоскости для программы визуализации графов Visual Graph. 2016. Свидетельство о государственной регистрации программ для ЭВМ № 2014617137 М.: Федеральная служба по интеллектуальной собственности, патентам и товарным знакам.

36. Трошина Е. Н., Чернов А. В. Восстановление типов данных в задаче декомпилирования в язык C // Прикладная информатика. 2009. № 6.
37. Aho A. V. Compilers: Principles, Techniques and Tools (for Anna University), 2/e. Pearson Education India, 2003.
38. Stroustrup B. The design and evolution of C++. Pearson Education India, 1994.
39. Вирт Н. Построение компиляторов [Электронный ресурс]/Никлаус Вирт; пер. с англ. ЕВ Борисов, ЛН Чернышов // М.: ДМК Пресс. 2010.
40. Ершов А. П. Организация АЛЬФА-транслятора // в сб."АЛЬФА—система автоматизации программирования Сиб. отд. изд-ва “Наука”–Новосибирск. 1967.
41. Бабецкий Г. И., Бежанова М. М., Волошин Ю. М. и др. Система автоматизации программирования АЛЬФА // Журнал вычислительной математики и математической физики. 1965. Т. 5, № 2. С. 317–325.
42. Huskey H. D., Halstead M., McArthur R. NELIAC—dialect of ALGOL // Communications of the ACM. 1960. Vol. 3, no. 8. P. 463–468.
43. Cifuentes C. Structuring decompiled graphs // Compiler Construction / Springer. 1996. P. 91–105.
44. The Boomerang Decompiler Project. 2006. URL: http://boomerang.sourceforge.net/ (дата обращения: 2015-01-19).
45. Трошина Е. Н., Чернов А. В. Инструментальная среда восстановления исходного кода программы-декомпилятор TyDec // Прикладная информатика. 2010. № 4.
46. SmartDec PUSHING NATIVE CODE DECOMPILATION TO THE NEXT LEVEL. 2015. URL: http://decompilation.info/ (дата обращения: 2015-01-19).
47. Turing A. M. On computable numbers, with an application to the Entscheidungsproblem // Proceedings of the London mathematical society. 1937. Vol. 2, no. 1. P. 230–265.

48. Jain T., Agrawal T. The haswell microarchitecture-4th generation processor // International Journal of Computer Science and Information Technologies. 2013. Vol. 4, no. 3. P. 477–480.
49. Jha A. Multi-register gather instruction. 2011. — 23. US Patent App. 13/995,437.
50. McCartney S. ENIAC: The triumphs and tragedies of the world's first computer. Walker & Company, 1999.
51. Guilfanov I. IDA fast library identification and recognition technology (FLIRT Technology): In-depth. 2012.
52. ILSpy .NET Decompiler. 2017. URL: http://ilspy.net/ (дата обращения: 2017-06-18).
53. .NET Reflector. 2017. URL: http://www.red-gate.com/products/dotnet-development/reflector/ (дата обращения: 2017-06-18).
54. Lattner C., Adve V. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation // Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California: 2004. — Mar.
55. Спецификация формата DCU на языке FlexT. 2017. URL: http://geos.icc.ru:8080/scripts/WWWBinV.dll/ShowR?DCU32.rfi (дата обращения: 2017-06-17).
56. Hmelnov A., Vassilyev S. Data description language FlexT: flexible types for description of static data // Proceedings of CSCC. Vol. 99. P. 1371–1376.
57. Хмельнов А. Е. DCU32INT - Программа для разбора юнитов Delphi. 2017. URL: http://hmelnov.icc.ru/DCU/index.ru.html (дата обращения: 2017-06-17).
58. Necula G. C., McPeak S., Rahul S. P. et al. CIL: Intermediate language and tools for analysis and transformation of C programs // International Conference on Compiler Construction / Springer. 2002. P. 213–228.
59. Серебряков В. А., Галочкин М. П. Основы конструирования компилято-

ров. Эдиториал УРСС М., 2001.

60. Ахо А., Сети Р., Ульман Дж. Компиляторы: принципы, технологии и инструментарий.–2-е издание // М.:«Вильямс. 2008. Т. 1184.
61. Muchnick S. S. Advanced compiler design implementation. Morgan Kaufmann, 1997.
62. Fokin A., Derevenetc E., Chernov A. et al. SmartDec: Approaching C++ decompilation // 2011 18th Working Conference on Reverse Engineering / IEEE. 2011. P. 347–356.
63. Ancona D., Lagorio G. Static Single Information Form for Abstract Compilation. // IFIP TCS / Springer. Vol. 2012. 2012. P. 10–27.
64. Евстигнеев В. А., Касьянов В. Н. Сводимые графы и граф-модели в программировании. Изд-во ИДМИ Новосибирск, 1999.
65. Ахо А, Дж Ульман. Теория синтаксического анализа, перевода и компиляции. Компиляция. 1978.
66. Lengauer T., Tarjan R. E. A fast algorithm for finding dominators in a flowgraph // ACM Transactions on Programming Languages and Systems (TOPLAS). 1979. Vol. 1, no. 1. P. 121–141.
67. Кнут Д. Э., Козаченко Ю. В. Искусство программирования: Сортировка и поиск. Издательский дом Вильямс, 2000. Т. 3.
68. Cooper K. D., Harvey T. J., Kennedy K. A simple, fast dominance algorithm // Software Practice & Experience. 2001. Vol. 4, no. 1-10. P. 1–8.
69. Baker B. S. An algorithm for structuring flowgraphs // Journal of the ACM (JACM). 1977. Vol. 24, no. 1. P. 98–120.
70. Williams M. H., Ossher H. Conversion of unstructured flow diagrams to structured form // The Computer Journal. 1978. Vol. 21, no. 2. P. 161–167.
71. Oulsnam G. Unravelling unstructured programs // The Computer Journal. 1982. Vol. 25, no. 3. P. 379–387.
72. Williams M. H., Chen G. Restructuring pascal programs containing goto statements // The Computer Journal. 1985. Vol. 28, no. 2. P. 134–137.

73. Ammarguellat Z. A control-flow normalization algorithm and its complexity // IEEE transactions on software engineering. 1992. Vol. 18, no. 3. P. 237–251.
74. Erosa A. M., Hendren L. J. Taming control flow: A structured approach to eliminating goto statements // Computer Languages, 1994., Proceedings of the 1994 International Conference on / IEEE. 1994. P. 229–240.
75. Knuth D. E., Floyd R. W. Notes on avoiding “go to” statements // Information processing letters. 1971. Vol. 1, no. 1. P. 23–31.
76. Williams M. H. Generating structured flow diagrams: the nature of unstructuredness // The Computer Journal. 1977. Vol. 20, no. 1. P. 45–50.
77. Lichtblau U. Decompilation of control structures by means of graph transformations // Mathematical Foundations of Software Development. 1985. P. 284–297.
78. Allen F. E. Control flow analysis // ACM Sigplan Notices / ACM. Vol. 5. 1970. P. 1–19.
79. Aho A. V., Sethi R., Ullman J. D. Compilers, Principles, Techniques. Addison wesley, 1986.
80. Cifuentes C., Simon D., Fraboulet A. Assembly to high-level language translation // Software Maintenance, 1998. Proceedings., International Conference on / IEEE. 1998. P. 228–237.
81. Johnson R., Pearson D., Pingali K. The program structure tree: Computing control regions in linear time // ACM SigPlan Notices / ACM. Vol. 29. 1994. P. 171–185.
82. Allen F. E., Cocke J. A program data flow analysis procedure // Communications of the ACM. 1976. Vol. 19, no. 3. P. 137.
83. Cross platform o. s. N. f. Mono is a software platform designed to allow developers to easily create cross platform applications part of the .NET Foundation. 2017. URL: http://www.mono-project.com/ (дата обращения: 2017-06-17).

84. Williams R. N. An extremely fast Ziv-Lempel data compression algorithm // Data Compression Conference, 1991. DCC'91. / IEEE. 1991. P. 362–371.
85. Fröhlich M., Werner M. Demonstration of the interactive graph visualization system da Vinci // International Symposium on Graph Drawing / Springer. 1994. P. 266–269.
86. Sander G. Graph layout through the VCG tool // International Symposium on Graph Drawing / Springer. 1994. P. 194–205.
87. Himsolt M. The Graphlet system (system demonstration) // International Symposium on Graph Drawing / Springer. 1996. P. 233–240.
88. Lauer H., Ettrich M., Soukup K. GraVis—system demonstration // International Symposium on Graph Drawing / Springer. 1997. P. 344–349.
89. Bridgeman S., Garg A., Tamassia R. A graph drawing and translation service on the WWW // International Symposium on Graph Drawing / Springer. 1996. P. 45–52.
90. Gansner E. R., North S. C. An open graph visualization system and its applications to software engineering // Software Practice and Experience. 2000. Vol. 30, no. 11. P. 1203–1233.
91. Золотухин Т. А. Визуализация графов при помощи программного средства Visual Graph // Информатика в науке и образовании. 2012. С. 135–148.
92. Касьянов В. Н., Касьянова Е. В. Визуализация информации на основе графовых моделей // Актуальные проблемы механики, математики, инфор-матики: сб. тез. науч.-практ. конф.(Пермь, 30 октября–1 ноября 2012 г.)/гл. ред. В. И. Яковлев; Перм. гос. нац. ис-след. ун-т.–Пермь, 2012.–195 с. 2011. С. 141.
93. Tutte W. T. How to draw a graph // Proc. London Math. Soc. 1963. Vol. 13, no. 3. P. 743–768.
94. Eades P., Xuemin L. How to draw a directed graph // Visual Languages, 1989., IEEE Workshop on / IEEE. 1989. P. 13–17.

95. Kamada T., Kawai S. An algorithm for drawing general undirected graphs // Information processing letters. 1989. Vol. 31, no. 1. P. 7–15.
96. Eades P., Wormald N. C. The median heuristic for drawing 2-layered networks. University of Queensland, Department of Computer Science, 1986.
97. JGraphX. JGraphX is a Java Swing diagramming (graph visualisation) library licensed under the BSD license. 2017. URL: https://github.com/jgraph/jgraphx (дата обращения: 2017-06-18).
98. SPEC CPU2000 – тесты вычислительной производительности центрального процессора. 2017. URL: https://www.spec.org/cpu2000/ (дата обращения: 2017-06-17).

# Список иллюстративного материала

# Приложение А

# Пример декомпиляции функции вычисления факториала

Листинг А.1. Исходный код функции на языке CIL

```
1
2 function  Fact (n: Borland.Delphi.System.Integer):
3  Borland.Delphi.System.Integer;
4 var
5  Result: Borland.Delphi.System.Integer;
6  i: Borland.Delphi.System.Integer;
7   : Borland.Delphi.System.Integer;
8 begin [Flags:3013,MaxStack:2,CodeSz:24,LocalVarSigTok:0]
9 // -- Basic Block #0 -- Incoming 0 -- // -- Outgoing 0
10 // -- Basic Block #1 -- Incoming 0 -- // -- Outgoing 2
11  00: .      |02                | ldarg.0
12  01: .      |17                | ldc.i4.1
13  02: /.     |2F 04             | bge.s   \$8
14 // -- Basic Block #2 -- Incoming 1 -- // -- Outgoing 1
15  04: .      |16                | ldc.i4.0
16  05: .      |0C                | stloc.2
17  06: +.     |2B 1A             | br.s \$22
18 // -- Basic Block #3 -- Incoming 1 -- // -- Outgoing 2
19  08: .      |17                | ldc.i4.1
20  09: .      |0C                | stloc.2
21  0A: .      |17                | ldc.i4.1
22  0B: .      |02                | ldarg.0
23  0C: .      |0A                | stloc.0
24  0D: .      |0B                | stloc.1
25  0E: .      |06                | ldloc.0
26  0F: .      |07                | ldloc.1
27  10: 2.     |32 10             | blt.s \$22
```

```
28 // -- Basic Block #4 -- Incoming 1 -- // -- Outgoing 1
29  12: .      |06                 | ldloc.0
30  13: .      |17                 | ldc.i4.1
31  14: X      |58                 | add Pop:
32  15: .      |0A                 | stloc.0
33 // -- Basic Block #5 -- Incoming 2 -- // -- Outgoing 2
34  16: .      |08                 | ldloc.2
35  17: .      |07                 | ldloc.1
36  18: Z      |5A                 | mul Pop:
37  19: .      |0C                 | stloc.2
38  1A: .      |07                 | ldloc.1
39  1B: .      |17                 | ldc.i4.1
40  1C: X      |58                 | add
41  1D: .      |0B                 | stloc.1
42  1E: .      |07                 | ldloc.1
43  1F: .      |06                 | ldloc.0
44  20: 3ϕ     |33 F4              | bne.un.s \$16
45 // -- Basic Block #6 -- Incoming 3 -- // -- Outgoing 0
46  22: .      |08                 | ldloc.2
47  23: *      |2A                 | ret
48 end;
```

Листинг А.2. Результат декомпиляции

```
1 function  Fact (n: Borland.Delphi.System.Integer):
2  Borland.Delphi.System.Integer;
3 var
4  Result: Borland.Delphi.System.Integer;
5  i: Borland.Delphi.System.Integer;
6   : Borland.Delphi.System.Integer;
7 begin [Flags:3013,MaxStack:2,CodeSz:24,LocalVarSigTok:0]
8  if (n < 1) then
9    Result := 0;
10 else begin
11   Result := 1;
12   LocVar := n;
13   i := 1;
14   if (LocVar >= i) then begin
15     LocVar := LocVar + 1;
16     repeat
17       Result := Result * i;
18       i := i + 1;
19     until (i <> LocVar);
20 end;
21 end;
```

# Приложение Б

# Процедура дизассемблирования CIL кода

```
1
2 function ProcessCommand(Action: TCmdAction; IP: Pointer): boolean;
3 var
4   opC, opC1: Byte;
5   F,Sz, Cnt: integer;
6   PCmdTbl: PCmdInfoTbl;
7   DP: Pointer;
8   CmdTblHi, D: integer;
9 begin
10  Result := false;
11  CodePtr := PrevCodePtr;
12  PCmdTbl := @OneByteOpCode;
13  CmdTblHi := High(OneByteOpCode);
14  if not ReadCodeByte(opC) then
15    Exit;
16  if opC>CmdTblHi then
17    Exit;
18  if opC <> \$fe then
19    ByteCode := OneByteOpCode[opC]
20  else begin
21    ReadCodeByte(opC1);
22    ByteCode := TwoBytesOpCode[opC1];
23  end;
24  DP := CodePtr;
25  Sz := 0;
26    case ByteCode.GetOperandType of
27      ShortInlineI ,ShortInlineBrTarget, ShortInlineVar,
28      ShortInlineArg: begin
29        SkipCode(1);
30        if ByteCode.GetOperandType = ShortInlineI then
```

```
31          ByteCode.I4:= ShortInt(DP^);
32        end;
33        InlineVar, InlineArg: begin
34          SkipCode(2);
35        end;
36        InlineBrTarget, ShortInlineR, InlineI: begin
37          SkipCode(4);
38          if ByteCode.GetOperandType = InlineI then
39            ByteCode.I4:= Integer(DP^);
40        end;
41        InlineI8, InlineR:
42          SkipCode(8);
43        InlineSwitch: begin
44          if not ReadCodeInt(Sz) then
45            Exit;
46          Cnt:= Sz;
47          ByteCode.ArgCnt:= Cnt;
48          while Cnt-1>0 do begin
49            Inc(CodePtr,SizeOf(integer));
50            ByteCode.SArgs[Cnt-1]:= (CodePtr-CodeBase)+LongInt(DP^);
51            Dec(Cnt);
52          end ;
53          //SkipCode(Sz*SizeOf(integer))
54        end;
55        InlineString, InlineSig, InlineTok, InlineType,
56        InlineMethod, InlineField:begin
57          SkipCode(4);
58        end;
59      end;
60    if CodePtr>CodeEnd then
61      Exit; //Error
62    Action(ByteCode,DP,IP);
63    Result := true;
64  end ;
```

# Приложение B

# Процедура инициализации опкодов CIL

```
1 procedure TCILOpCodes.InitOpCodes;
2 begin
3  Newarr :=  TCILOpCode.Create(92966399, 436407299);
4  Nop :=  TCILOpCode.Create(83886335, 318768389);
5  opBreak := TCILOpCode.Create(16843263, 318768389);
6  Ldarg_0 := TCILOpCode.Create(84017919, 335545601);
7  Ldarg_1 := TCILOpCode.Create(84083711, 335545601);
8  Ldarg_2 := TCILOpCode.Create(84149503, 335545601);
9  Ldarg_3 := TCILOpCode.Create(84215295, 335545601);
10 Ldloc_0 := TCILOpCode.Create(84281087, 335545601);
11 Ldloc_1 := TCILOpCode.Create(84346879, 335545601);
12 Ldloc_2 := TCILOpCode.Create(84412671, 335545601);
13 Ldloc_3 := TCILOpCode.Create(84478463, 335545601);
14 Stloc_0 := TCILOpCode.Create(84544255, 318833921);
15 Stloc_1 := TCILOpCode.Create(84610047, 318833921);
16 Stloc_2 := TCILOpCode.Create(84675839, 318833921);
17 Stloc_3 := TCILOpCode.Create(84741631, 318833921);
18 Ldarg_S := TCILOpCode.Create(84807423, 335549185);
19 Ldarga_S := TCILOpCode.Create(84873215, 369103617);
20 Starg_S := TCILOpCode.Create(84939007, 318837505);
21 Ldloc_S := TCILOpCode.Create(85004799, 335548929);
22 Ldloca_S := TCILOpCode.Create(85070591, 369103361);
23 Stloc_S := TCILOpCode.Create(85136383, 318837249);
24 Ldnull := TCILOpCode.Create(85202175, 436208901);
25 Ldc_I4_M1 := TCILOpCode.Create(85267967, 369100033);
26 Ldc_I4_0 := TCILOpCode.Create(85333759, 369100033);
27 Ldc_I4_1 := TCILOpCode.Create(85399551, 369100033);
28 Ldc_I4_2 := TCILOpCode.Create(85465343, 369100033);
29 Ldc_I4_3 := TCILOpCode.Create(85531135, 369100033);
30 Ldc_I4_4 := TCILOpCode.Create(85596927, 369100033);
```

```
31  Ldc_I4_5 := TCILOpCode.Create(85662719, 369100033);
32  Ldc_I4_6 := TCILOpCode.Create(85728511, 369100033);
33  Ldc_I4_7 := TCILOpCode.Create(85794303, 369100033);
34  Ldc_I4_8 := TCILOpCode.Create(85860095, 369100033);
35  Ldc_I4_S := TCILOpCode.Create(85925887, 369102849);
36  Ldc_I4 := TCILOpCode.Create(85991679, 369099269);
37  Ldc_I8 := TCILOpCode.Create(86057471, 385876741);
38  Ldc_R4 := TCILOpCode.Create(86123263, 402657541);
39  Ldc_R8 := TCILOpCode.Create(86189055, 419432197);
40  Dup := TCILOpCode.Create(86255103, 352388357);
41  Pop := TCILOpCode.Create(86320895, 318833925);
42  Jmp := TCILOpCode.Create(36055039, 318768133);
43  opCall := TCILOpCode.Create(36120831, 471532549);
44  Calli := TCILOpCode.Create(36186623, 471533573);
45  Ret := TCILOpCode.Create(120138495, 320537861);
46  Br_S := TCILOpCode.Create(2763775, 318770945);
47  Brfalse_S := TCILOpCode.Create(53161215, 318967553);
48  Brtrue_S := TCILOpCode.Create(53227007, 318967553);
49  Beq_S := TCILOpCode.Create(53292799, 318902017);
50  Bge_S := TCILOpCode.Create(53358591, 318902017);
51  Bgt_S := TCILOpCode.Create(53424383, 318902017);
52  Ble_S := TCILOpCode.Create(53490175, 318902017);
53  Blt_S := TCILOpCode.Create(53555967, 318902017);
54  Bne_Un_S := TCILOpCode.Create(53621759, 318902017);
55  Bge_Un_S := TCILOpCode.Create(53687551, 318902017);
56  Bgt_Un_S := TCILOpCode.Create(53753343, 318902017);
57  Ble_Un_S := TCILOpCode.Create(53819135, 318902017);
58  Blt_Un_S := TCILOpCode.Create(53884927, 318902017);
59  Br := TCILOpCode.Create(3619071, 318767109);
60  Brfalse := TCILOpCode.Create(54016511, 318963717);
61  Brtrue := TCILOpCode.Create(54082303, 318963717);
62  Beq := TCILOpCode.Create(54148095, 318898177);
63  Bge := TCILOpCode.Create(54213887, 318898177);
64  Bgt := TCILOpCode.Create(54279679, 318898177);
65  Ble := TCILOpCode.Create(54345471, 318898177);
```

```
66  Blt := TCILOpCode.Create(54411263, 318898177);
67  Bne_Un := TCILOpCode.Create(54477055, 318898177);
68  Bge_Un := TCILOpCode.Create(54542847, 318898177);
69  Bgt_Un := TCILOpCode.Create(54608639, 318898177);
70  Ble_Un := TCILOpCode.Create(54674431, 318898177);
71  Blt_Un := TCILOpCode.Create(54740223, 318898177);
72  Switch := TCILOpCode.Create(54806015, 318966277);
73  Ldind_I1 := TCILOpCode.Create(88426239, 369296645);
74  Ldind_U1 := TCILOpCode.Create(88492031, 369296645);
75  Ldind_I2 := TCILOpCode.Create(88557823, 369296645);
76  Ldind_U2 := TCILOpCode.Create(88623615, 369296645);
77  Ldind_I4 := TCILOpCode.Create(88689407, 369296645);
78  Ldind_U4 := TCILOpCode.Create(88755199, 369296645);
79  Ldind_I8 := TCILOpCode.Create(88820991, 386073861);
80  Ldind_I  := TCILOpCode.Create(88886783, 369296645);
81  Ldind_R4 := TCILOpCode.Create(88952575, 402851077);
82  Ldind_R8 := TCILOpCode.Create(89018367, 419628293);
83  Ldind_Ref := TCILOpCode.Create(89084159, 436405509);
84  Stind_Ref := TCILOpCode.Create(89149951, 319096069);
85  Stind_I1 := TCILOpCode.Create(89215743, 319096069);
86  Stind_I2 := TCILOpCode.Create(89281535, 319096069);
87  Stind_I4 := TCILOpCode.Create(89347327, 319096069);
88  Stind_I8 := TCILOpCode.Create(89413119, 319161605);
89  Stind_R4 := TCILOpCode.Create(89478911, 319292677);
90  Stind_R8 := TCILOpCode.Create(89544703, 319358213);
91  Add := TCILOpCode.Create(89610495, 335676677);
92  opSub := TCILOpCode.Create(89676287, 335676677);
93  Mul := TCILOpCode.Create(89742079, 335676677);
94  opDiv := TCILOpCode.Create(89807871, 335676677);
95  Div_Un := TCILOpCode.Create(89873663, 335676677);
96  opRem := TCILOpCode.Create(89939455, 335676677);
97  Rem_Un := TCILOpCode.Create(90005247, 335676677);
98  opAnd := TCILOpCode.Create(90071039, 335676677);
99  opOr := TCILOpCode.Create(90136831, 335676677);
100 opXor := TCILOpCode.Create(90202623, 335676677);
```

```
101 opShl := TCILOpCode.Create(90268415, 335676677);
102 opShr := TCILOpCode.Create(90334207, 335676677);
103 Shr_Un := TCILOpCode.Create(90399999, 335676677);
104 Neg := TCILOpCode.Create(90465791, 335611141);
105 opNot := TCILOpCode.Create(90531583, 335611141);
106 Conv_I1 := TCILOpCode.Create(90597375, 369165573);
107 Conv_I2 := TCILOpCode.Create(90663167, 369165573);
108 Conv_I4 := TCILOpCode.Create(90728959, 369165573);
109 Conv_I8 := TCILOpCode.Create(90794751, 385942789);
110 Conv_R4 := TCILOpCode.Create(90860543, 402720005);
111 Conv_R8 := TCILOpCode.Create(90926335, 419497221);
112 Conv_U4 := TCILOpCode.Create(90992127, 369165573);
113 Conv_U8 := TCILOpCode.Create(91057919, 385942789);
114 Callvirt := TCILOpCode.Create(40792063, 471532547);
115 Cpobj := TCILOpCode.Create(91189503, 319097859);
116 Ldobj := TCILOpCode.Create(91255295, 335744003);
117 Ldstr := TCILOpCode.Create(91321087, 436209923);
118 obj := TCILOpCode.Create(41055231, 437978115);
119 Castclass := TCILOpCode.Create(91452671, 436866051);
120 Isinst := TCILOpCode.Create(91518463, 369757187);
121 Conv_R_Un := TCILOpCode.Create(91584255, 419497221);
122 Unbox := TCILOpCode.Create(91650559, 369757189);
123 opThrow := TCILOpCode.Create(142047999, 319423747);
124 Ldfld := TCILOpCode.Create(91782143, 336199939);
125 Ldflda := TCILOpCode.Create(91847935, 369754371);
126 Stfld := TCILOpCode.Create(91913727, 319488259);
127 Ldsfld := TCILOpCode.Create(91979519, 335544579);
128 Ldsflda := TCILOpCode.Create(92045311, 369099011);
129 Stsfld := TCILOpCode.Create(92111103, 318832899);
130 Stobj := TCILOpCode.Create(92176895, 319032323);
131 Conv_Ovf_I1_Un := TCILOpCode.Create(92242687, 369165573);
132 Conv_Ovf_I2_Un := TCILOpCode.Create(92308479, 369165573);
133 Conv_Ovf_I4_Un := TCILOpCode.Create(92374271, 369165573);
134 Conv_Ovf_I8_Un := TCILOpCode.Create(92440063, 385942789);
135 Conv_Ovf_U1_Un := TCILOpCode.Create(92505855, 369165573);
```

```
136 Conv_Ovf_U2_Un := TCILOpCode.Create(92571647, 369165573);
137 Conv_Ovf_U4_Un := TCILOpCode.Create(92637439, 369165573);
138 Conv_Ovf_U8_Un := TCILOpCode.Create(92703231, 385942789);
139 Conv_Ovf_I_Un := TCILOpCode.Create(92769023, 369165573);
140 Conv_Ovf_U_Un := TCILOpCode.Create(92834815, 369165573);
141 Box := TCILOpCode.Create(92900607, 436276229);
142 arr := TCILOpCode.Create(92966399, 436407299);
143 Ldlen := TCILOpCode.Create(93032191, 369755395);
144 Ldelema := TCILOpCode.Create(93097983, 369888259);
145 Ldelem_I1 := TCILOpCode.Create(93163775, 369886467);
146 Ldelem_U1 := TCILOpCode.Create(93229567, 369886467);
147 Ldelem_I2 := TCILOpCode.Create(93295359, 369886467);
148 Ldelem_U2 := TCILOpCode.Create(93361151, 369886467);
149 Ldelem_I4 := TCILOpCode.Create(93426943, 369886467);
150 Ldelem_U4 := TCILOpCode.Create(93492735, 369886467);
151 Ldelem_I8 := TCILOpCode.Create(93558527, 386663683);
152 Ldelem_I := TCILOpCode.Create(93624319, 369886467);
153 Ldelem_R4 := TCILOpCode.Create(93690111, 403440899);
154 Ldelem_R8 := TCILOpCode.Create(93755903, 420218115);
155 Ldelem_Ref := TCILOpCode.Create(93821695, 436995331);
156 Stelem_I := TCILOpCode.Create(93887487, 319620355);
157 Stelem_I1 := TCILOpCode.Create(93953279, 319620355);
158 Stelem_I2 := TCILOpCode.Create(94019071, 319620355);
159 Stelem_I4 := TCILOpCode.Create(94084863, 319620355);
160 Stelem_I8 := TCILOpCode.Create(94150655, 319685891);
161 Stelem_R4 := TCILOpCode.Create(94216447, 319751427);
162 Stelem_R8 := TCILOpCode.Create(94282239, 319816963);
163 Stelem_Ref := TCILOpCode.Create(94348031, 319882499);
164 Ldelem_Any := TCILOpCode.Create(94413823, 336333827);
165 Stelem_Any := TCILOpCode.Create(94479615, 319884291);
166 Unbox_Any := TCILOpCode.Create(94545407, 336202755);
167 Conv_Ovf_I1 := TCILOpCode.Create(94614527, 369165573);
168 Conv_Ovf_U1 := TCILOpCode.Create(94680319, 369165573);
169 Conv_Ovf_I2 := TCILOpCode.Create(94746111, 369165573);
170 Conv_Ovf_U2 := TCILOpCode.Create(94811903, 369165573);
```

```
171  Conv_Ovf_I4 := TCILOpCode.Create(94877695, 369165573);
172  Conv_Ovf_U4 := TCILOpCode.Create(94943487, 369165573);
173  Conv_Ovf_I8 := TCILOpCode.Create(95009279, 385942789);
174  Conv_Ovf_U8 := TCILOpCode.Create(95075071, 385942789);
175  Refanyval := TCILOpCode.Create(95142655, 369167365);
176  Ckfinite := TCILOpCode.Create(95208447, 419497221);
177  Mkrefany := TCILOpCode.Create(95274751, 335744005);
178  Ldtoken := TCILOpCode.Create(95342847, 369101573);
179  Conv_U2 := TCILOpCode.Create(95408639, 369165573);
180  Conv_U1 := TCILOpCode.Create(95474431, 369165573);
181  Conv_I := TCILOpCode.Create(95540223, 369165573);
182  Conv_Ovf_I := TCILOpCode.Create(95606015, 369165573);
183  Conv_Ovf_U := TCILOpCode.Create(95671807, 369165573);
184  Add_Ovf := TCILOpCode.Create(95737599, 335676677);
185  Add_Ovf_Un := TCILOpCode.Create(95803391, 335676677);
186  Mul_Ovf := TCILOpCode.Create(95869183, 335676677);
187  Mul_Ovf_Un := TCILOpCode.Create(95934975, 335676677);
188  Sub_Ovf := TCILOpCode.Create(96000767, 335676677);
189  Sub_Ovf_Un := TCILOpCode.Create(96066559, 335676677);
190  Endfinally := TCILOpCode.Create(129686783, 318768389);
191  Leave := TCILOpCode.Create(12312063, 319946757);
192  Leave_S := TCILOpCode.Create(12377855, 319950593);
193  Stind_I := TCILOpCode.Create(96329727, 319096069);
194  Conv_U := TCILOpCode.Create(96395519, 369165573);
195  Arglist := TCILOpCode.Create(96403710, 369100037);
196  Ceq := TCILOpCode.Create(96469502, 369231109);
197  Cgt := TCILOpCode.Create(96535294, 369231109);
198  Cgt_Un := TCILOpCode.Create(96601086, 369231109);
199  Clt := TCILOpCode.Create(96666878, 369231109);
200  Clt_Un := TCILOpCode.Create(96732670, 369231109);
201  Ldftn := TCILOpCode.Create(96798462, 369099781);
202  Ldvirtftn := TCILOpCode.Create(96864254, 369755141);
203  Ldarg := TCILOpCode.Create(96930302, 335547909);
204  Ldarga := TCILOpCode.Create(96996094, 369102341);
205  Starg := TCILOpCode.Create(97061886, 318836229);
```

```
206  Ldloc := TCILOpCode.Create(97127678, 335547653);
207  Ldloca := TCILOpCode.Create(97193470, 369102085);
208  Stloc := TCILOpCode.Create(97259262, 318835973);
209  Localloc := TCILOpCode.Create(97325054, 369296645);
210  Endfilter := TCILOpCode.Create(130945534, 318964997);
211  Unaligned := TCILOpCode.Create(80679678, 318771204);
212  Volatile := TCILOpCode.Create(80745470, 318768388);
213  Tail := TCILOpCode.Create(80811262, 318768388);
214  Initobj := TCILOpCode.Create(97654270, 318966787);
215  Constrained := TCILOpCode.Create(97720062, 318770180);
216  Cpblk := TCILOpCode.Create(97785854, 319227141);
217  Initblk := TCILOpCode.Create(97851646, 319227141);
218  No := TCILOpCode.Create(97917438, 318771204);
219  Rethrow := TCILOpCode.Create(148314878, 318768387);
220  Sizeof := TCILOpCode.Create(98049278, 369101829);
221  Refanytype := TCILOpCode.Create(98115070, 369165573);
222  opReadonly := TCILOpCode.Create(98180862, 318768388);
223  Newobj := TCILOpCode.Create(41055231, 437978115);
224  Throw := TCILOpCode.Create(142047999, 319423747);
225 end;
```

# Приложение Г

# Пример декомпиляции главной процедуры сжатия

Листинг Г.1. Исходный код функции

```
1 procedure TWinForm.btnCompress_Click(sender: System.Object;
2                                      e: System.EventArgs);
3 var
4   finfo : FileInfo;
5   finput :FileStream;
6   bwriter : BinaryWriter;
7   ms : MemoryStream;
8   fs : FileStream;
9 begin
10 finfo := FileInfo.Create(textInput.Text);
11 if (finfo.Exists) then
12 begin
13   finput := finfo.OpenRead();
14     ms := MemoryStream.Create;
15     bwriter := BinaryWriter.Create(ms);
16     LZRWCompressFileToStream(finput, bwriter);
17     if (bwriter <> nil) then
18     begin
19       fs := FileStream.Create(textOutput.Text,FileMode.Create);
20       bwriter.BaseStream.Seek(0,SeekOrigin(0));
21       (MemoryStream (bwriter.BaseStream)).WriteTo(fs);
22       fs.Close();
23       bwriter.Close();
24     end;
25     finput.Close();
26   end;
27 end;
```

Листинг Г.2. Исходный код функции на языке CIL

```
1 procedure  TWinForm.btnCompress_Click (sender: Object; e: EventArgs);
2 var
3  finfo: FileInfo;
4  finput: FileStream;
5  bwriter: BinaryWriter;
6  ms: MemoryStream;
7  fs: FileStream;
8 begin [Flags:3013,MaxStack:3,CodeSz:80,LocalVarSigTok:0]
9 // -- Basic Block #0
10 // -- Basic Block #1
11  00: .      |02                | ldarg_0
12  01: {....  |7B(05 00 00 00     | ldfld  &KA TWinForm{#$8E} .textInput{+5}
13  06: oЌ...  |6F(8D 00 00 00     | callvirt  &KA Control{#$46} .get_Text{+141}
14  0B: s....  |73(00 00 00 00     | newobj  &K6 FileInfo.Create{#$27}
15  10: .      |0A                | stloc_0
16  11: .      |06                | ldloc_0
17  12: o....  |6F(06 00 00 00     | callvirt &KA FileSystemInfo{#$31}
18                                          .get_Exists{+6}
19  17: ,f     |2C 66             | brfalse_s  $7F
20 // -- Basic Block #2
21  19: .      |06                | ldloc_0
22  1A: (....  |28(00 00 00 00     | call  &K6 FileInfo.OpenRead{#$28}
23  1F: .      |0C                | stloc_2
24  20: s....  |73(00 00 00 00     | newobj  &K6 MemoryStream.Create{#$2A}
25  25: .      |0D                | stloc_3
26  26: .      |09                | ldloc_3
27  27: s....  |73(00 00 00 00     | newobj  &K6 BinaryWriter.Create{#$2C}
28  2C: .      |0B                | stloc_1
29  2D: .      |02                | ldarg_0
30  2E: .      |08                | ldloc_2
31  2F: .      |07                | ldloc_1
32  30: (....  |28(00 00 00 00     | call  &K6
33                                  TWinForm.LZRWCompressFileToStream{#$BF}
```

```
34  35: .      |07                | ldloc_1
35  36: ,A     |2C 41             | brfalse_s  $79
36 // -- Basic Block #3
37  38: .      |02                | ldarg_0
38  39: {....  |7B(06 00 00 00     | ldfld  &KA TWinForm{#$8E} .textOutput{+6}
39  3E: oЌ...  |6F(8D 00 00 00     | callvirt  &KA Control{#$46} .get_Text{+141}
40  43: .      |18                | ldc_i4_2
41  44: s....  |73(00 00 00 00     | newobj  &K6 FileStream.Create{#$2F}
42  49: ..     |13 04             | stloc_s  $04
43  4B: .      |07                | ldloc_1
44  4C: o....  |6F(06 00 00 00     | callvirt  &KA BinaryWriter{#$21}
45                                            .get_BaseStream{+6}
46  51: .      |16                | ldc_i4_0
47  52: n      |6E                | conv_u8
48  53: .      |16                | ldc_i4_0
49  54: o....  |6F(0E 00 00 00     | callvirt  &KA Stream{#$20} .Seek{+14}
50  59: &      |26                | pop
51  5A: .      |07                | ldloc_1
52  5B: o....  |6F(06 00 00 00     | callvirt  &KA BinaryWriter{#$21}
53                                            .get_BaseStream{+6}
54  60: u....  |75(00 00 00 00     | isinst  &K7 MemoryStream{#$23}
55  65: ..     |11 04             | ldloc_s  $04
56  67: o....  |6F(1A 00 00 00     | callvirt  &KA MemoryStream{#$23}
57                                           .WriteTo{+26}
58  6C: ..     |11 04             | ldloc_s  $04
59  6E: o....  |6F(12 00 00 00     | callvirt  &KA FileStream{#$25} .Close{+18}
60  73: .      |07                | ldloc_1
61  74: o....  |6F(04 00 00 00     | callvirt  &KA BinaryWriter{#$21} .Close{+4}
62 // -- Basic Block #4
63  79: .      |08                | ldloc_2
64  7A: o....  |6F(12 00 00 00     | callvirt  &KA FileStream{#$25} .Close{+18}
65 // -- Basic Block #5
66  7F: *      |2A                | ret
67 end ;
```

Листинг Г.3. Результат декомпиляции

```
1 procedure  TWinForm.btnCompress_Click (sender: Object; e: EventArgs);
2 var
3  finfo: FileInfo;
4  finput: FileStream;
5  bwriter: BinaryWriter;
6  ms: MemoryStream;
7  fs: FileStream;
8 begin [Flags:3013,MaxStack:3,CodeSz:80,LocalVarSigTok:0]
9  finfo := FileInfo.Create(Control.get_Text(TWinForm.textInput));
10  if (FileSystemInfo.get_Exists(finfo) <> 0) then begin
11    finput := FileInfo.OpenRead(finfo);
12    ms := MemoryStream.Create();
13    bwriter := BinaryWriter.Create(ms);
14    TWinForm.LZRWCompressFileToStream(Self, finput, bwriter)
15    if (bwriter <> 0) then begin
16      fs := FileStream.Create(Control.get_Text(TWinForm.textOutput), 2);
17      MemoryStream.WriteTo(MemoryStream(BinaryWriter.get_BaseStream(
18                          bwriter)), fs)
19      FileStream.Close(fs)
20      BinaryWriter.Close(bwriter)
21    end;
22    FileStream.Close(finput)
23  end;
24 end ;
```

# Приложение Д

# Некоторые структуры данных разработанного декомпилятора

```
1  TInstruction = class(TCmd)
2  protected
3    FByteCode: TCILOpCode;
4    FExpr: TCILExpr;
5    FI4: integer;
6    FFix: Integer;
7    FFixupRec: PFixupRec;
8    FIsProc: boolean;
9    FSArgs: array[byte] of LongInt;
10   FArgCnt: integer;
11   procedure SetArg(index: integer; Value: LongInt);
12   function GetArg(index: integer): LongInt;
13 public
14   constructor Create0(AOfs: integer; ByteCode: TCILOpCode);
15   procedure AsString();
16   property ByteCode: TCILOpCode read FByteCode;
17   property Expr: TCILExpr read FExpr;
18   property I4: integer read FI4 write FI4;
19   property Fix: integer read FFix write FFix;
20   property FixupRec: PFixupRec read FFixupRec write FFixupRec;
21   property IsProc: boolean read FIsProc write FIsProc;
22   property ArgCnt: integer read FArgCnt write FArgCnt;
23   property SArgs[index: integer]: LongInt read GetArg write SetArg;
24 end ;
25
26 TPredcessors = class;
27
28 TCtrlFlowNode = class(TCmdSeq)
29 protected
```

```
30    FInCtx: TCILCtx;
31    FOutCtx: TCILCtx;
32    FInEdges: TList;
33    FIncoming: TPredcessors;
34    FOutgoing: TList;
35    FDominatorTreeChildren: TList;
36    FImmediateDominator: TCtrlFlowNode;
37    FDominanceFrontier: TCtrlFlowNode;
38    FCmdOfs : Cardinal;
39    FVisited: boolean;
40    FLinesCnt: integer;
41    FLabel: TCILLabel;
42    FLInd: integer;
43    function GetInEdgeCnt: integer;
44    function GetInEdges: TList;
45    function GetIncoming: TPredcessors;
46    function GetOutgoing: TList;
47    function GetNext: PCmdSeqRef;
48    function GetNextCond: PCmdSeqRef;
49    function GetStart: TInstruction;
50    function GetEnd: TInstruction;
51    function GetExprByOpC(var Instr: TInstruction): boolean;
52  public
53    FIndex: integer;
54    constructor Create(AStart: integer);override;
55    constructor Create0();
56    procedure Show;
57    destructor Destroy;override;
58    procedure BuildILAst;
59    function GetStr: String;
60    procedure AddInRef(CmdSeq: TCtrlFlowNode);
61    //function AddCmd(AStart,ASize: Cardinal): TCmd; override;
62    property InEdgeCnt : integer read GetInEdgeCnt;
63    function RemoveGoToExpr(Item: Pointer): Integer;
64    property InEdges : TList read GetInEdges;
```

```
65    property CmdOfs : Cardinal read FCmdOfs write FCmdOfs;
66    property Next : PCmdSeqRef read GetNext;
67    property Cond : PCmdSeqRef read GetNextCond;
68    property StartOpCode : TInstruction read GetStart;
69    property EndOpCode : TInstruction read GetEnd;
70    property Incoming : TPredcessors read GetIncoming;
71    property Outgoing : TList read GetOutgoing;
72    property DominatorTreeChildren: TList read FDominatorTreeChildren;
73    property ImmediateDominator: TCtrlFlowNode read
74                                   FImmediateDominator write FImmediateDominator;
75    property DominanceFrontier: TCtrlFlowNode read
76                                  FDominanceFrontier write FDominanceFrontier;
77    property Visited : boolean read FVisited write FVisited;
78    property InCtx: TCILCtx read FInCtx write FInCtx;
79    property OutCtx: TCILCtx read FOutCtx write FOutCtx;
80    property LinesCnt: integer read FLinesCnt write FLinesCnt;
81    property Commands: TList read FCommands;
82  end;
83
84  TMethodBody = class(TProc)
85  protected
86    FCtx: TCILCtx;
87    destructor Destroy;
88  private
89    FLInd: integer;
90    FRegularExit: TCtrlFlowNode;
91    function GetEntryPoint: TCtrlFlowNode;
92    function GetRegularExit: TCtrlFlowNode;
93    procedure SetEntryPoint(const Value: TCtrlFlowNode);
94    procedure SetRegularExit(const Value: TCtrlFlowNode);
95  public
96    constructor Create(AStart,ASize: integer; Ctx: TCILCtx);
97    destructor Free;
98    procedure SetState;
99    procedure ResetVisited;
```

```
100     procedure CreateCtrFlowEdges;
101     procedure ComputeDominance;
102     procedure FindConditions(var Body: TMethodBody);
103     procedure SimplifyConditions(var Body: TMethodBody);
104     procedure BuildAst;
105     procedure Show;
106     procedure ShowDom;
107     procedure ShowGraph;
108     procedure RemoveNode(SrcNode, TrgNode: Pointer);
109     function GetIndex(I: TCtrlFlowNode): integer;
110     property EntryPoint: TCtrlFlowNode read GetEntryPoint
111                                        write SetEntryPoint;
112     property RegularExit: TCtrlFlowNode read GetRegularExit
113                                         write SetRegularExit;
114     property Ctx: TCILCtx read FCtx write FCtx;
115   end;
```

# Приложение E

# Результат декомпиляции модуля WinForm.dcuil

```
1 unit  WinForm;
2
3 interface
4
5 uses
6  System.Runtime.CompilerServices,
7  System.Runtime.InteropServices,
8  System,
9  LZRW1_EIS,
10  bitwiseclass,
11  System.IO,
12  .System.Data,
13  System.Data,
14  .System.Windows.Forms,
15  System.Windows.Forms,
16  System.ComponentModel,
17  .mscorlib,
18  System.Collections,
19  .System.Drawing,
20  System.Drawing,
21  Borland.Delphi.System,
22  .System,
23  System;
24
25 type
26  $Unit = class (TObject)
27  public
28    class  method  $ClassInit: WinForm.$ClassInit{#$D2} ;
29  end ;
30
```

```
31  int = Integer;
32
33  TWinForm = class (Form)
34  {type}
35  private
36    Components: Container;
37    btnCompress: Button;
38    btnDeCompress: Button;
39    textInput: TextBox;
40    textOutput: TextBox;
41    lblInput: Label;
42    lblOutput: Label;
43    method  InitializeComponent: TWinForm.InitializeComponent{#$8F};
44    method  btnCompress_Click: TWinForm.btnCompress_Click{#$AD} ;
45    method  btnDeCompress_Click: TWinForm.btnDeCompress_Click{#$B6};
46    method  TWinForm_Load: TWinForm.TWinForm_Load{#$CE} ;
47  protected
48    method  Dispose: TWinForm.Dispose{#$A8}; override{;virtual @48};
49  public
50    compressor: LZRW1KHCompressor;
51    constructor  Create: TWinForm.Create{#$AB};
52  private
53    method  LZRWCompressFileToStream:
54            TWinForm.LZRWCompressFileToStream{#$BF};
55    method  LZRWDeCompressFromStream:
56            TWinForm.LZRWDeCompressFromStream{#$C6};
57  public
58    class  method  $ClassInit: TWinForm.$ClassInit{#$8A};
59  end ;
60
61 procedure  WinForm;
62
63 implementation
64
65 uses
```

```
 66  System.Globalization;
 67
 68 procedure  TWinForm.InitializeComponent;
 69 type
 70  $2 = procedure (sender: Object; e: EventArgs) of object ;
 71  $delegate1 = class (MulticastDelegate)
 72  public
 73    procedure  Create(object: Object; method: IntPtr);
 74    procedure  Invoke(sender: Object; e: EventArgs);
 75    function  BeginInvoke(sender: Object; e: EventArgs;
 76      callback: AsyncCallback; object: Object): IAsyncResult;
 77    procedure  EndInvoke(result: IAsyncResult);
 78  end ;
 79  $3 = procedure (sender: Object; e: EventArgs) of object ;
 80  $delegate2 = class (MulticastDelegate)
 81  public
 82    procedure  Create(object: Object; method: IntPtr);
 83    procedure  Invoke(sender: Object; e: EventArgs);
 84    function  BeginInvoke(sender: Object; e: EventArgs;
 85      callback: AsyncCallback; object: Object): IAsyncResult;
 86    procedure  EndInvoke(result: IAsyncResult);
 87  end ;
 88  $4 = procedure (sender: Object; e: EventArgs) of object ;
 89  $delegate3 = class (MulticastDelegate)
 90  public
 91    procedure  Create(object: Object; method: IntPtr);
 92    procedure  Invoke(sender: Object; e: EventArgs);
 93    function  BeginInvoke(sender: Object; e: EventArgs;
 94      callback: AsyncCallback; object: Object): IAsyncResult;
 95    procedure  EndInvoke(result: IAsyncResult);
 96  end ;
 97 const
 98   $WStr$val$1 = WideString('btnCompress' );
 99   $WStr$val$2 = WideString('Compress' );
100   $WStr$val$3 = WideString('btnDeCompress' );
```

```
101   $WStr$val$4 = WideString('DeCompress' );
102   $WStr$val$5 = WideString('textInput' );
103   $WStr$val$6 = WideString('' );
104   $WStr$val$7 = WideString('textOutput' );
105   $WStr$val$8 = WideString('' );
106   $WStr$val$9 = WideString('lblInput' );
107   $WStr$val$10 = WideString('Full Path of Input:' );
108   $WStr$val$11 = WideString('lblOutput' );
109   $WStr$val$12 = WideString('Full Path of Output:' );
110   $WStr$val$13 = WideString('TWinForm' );
111   $WStr$val$14 = WideString('LZRW1/KH De/Compressor by E.I. Simay');
112 begin [Flags:3013,MaxStack:3,CodeSz:334,LocalVarSigTok:0]
113   btnCompress := Button.Create();
114   btnDeCompress := Button.Create();
115   textInput := TextBox.Create();
116   textOutput := TextBox.Create();
117   lblInput := Label.Create();
118   lblOutput := Label.Create();
119   Control.SuspendLayout(Self);
120   Self.Controls.Add(lblInput);
121   Self.Controls.Add(lblOutput);
122   Self.Controls.Add(textInput);
123   Self.Controls.Add(textOutput);
124   Self.Controls.Add(btnCompress);
125   Self.Controls.Add(btnDeCompress);
126   Self.Name := 'TWinForm';
127   Self.Text := 'LZRW1/KH De/Compressor by E.I. Simay';
128   Include(Self.Load, Self.TWinForm_Load);
129   Self.ResumeLayout(False);
130
131 end ;
132
133 procedure  TWinForm.Dispose (Disposing:
134           Borland.Delphi.System.Boolean); overload ;
135 begin
```

```
136   if Disposing then begin
137     if Components <> nil then
138       Components.Dispose();
139   end;
140   inherited Dispose(Disposing);
141 end ;
142
143 constructor  TWinForm.Create;
144 begin
145   inherited Create;
146   InitializeComponent;
147   compressor := LZRW1KHCompressor.Create;
148 end ;
149
150 procedure TWinForm.btnCompress_Click (sender: Object; e: EventArgs);
151 var
152  finfo: FileInfo;
153  finput: FileStream;
154  bwriter: BinaryWriter;
155  ms: MemoryStream;
156  fs: FileStream;
157 begin
158   finfo := FileInfo.Create(textInput.Text);
159   if (finfo.Exists <> nil) then begin
160     finput := finfo.OpenRead();
161     ms := MemoryStream.Create;
162     bwriter := BinaryWriter.Create(ms);
163     LZRWCompressFileToStream(finput, bwriter);
164     if (bwriter <> nil) then begin
165       fs := FileStream.Create(textOutput.Text,FileMode.Create);
166   bwriter.BaseStream.Seek(0,SeekOrigin(0));
167   (MemoryStream (bwriter.BaseStream)).WriteTo(fs);
168   fs.Close();
169   bwriter.Close();
170     end;
```

```
171     finput.Close();
172   end;
173 end;
174 end ;
175
176 procedure  TWinForm.btnDeCompress_Click (sender: Object;
177                                          e: EventArgs);
178 var
179  finfo: FileInfo;
180  finput: FileStream;
181  breader: BinaryReader;
182  ms: MemoryStream;
183  fs: FileStream;
184 begin
185   finfo := FileInfo.Create(textInput.Text);
186   if (finfo.Exists) then begin
187     finput := FileStream.Create(textInput.Text,
188                                 FileMode.Open,FileAccess.Read);
189     breader := BinaryReader.Create(finput);
190     fs := FileStream.Create(textOutput.Text,FileMode.Create);
191     try
192       ms := LZRWDeCompressFromStream(breader);
193     if (ms <> nil) then begin
194         ms.WriteTo(fs);
195     ms.Close();
196   end;
197     except
198      fs.Close();
199      breader.Close();
200     end;
201   end;
202 end ;
203
204 procedure  TWinForm.LZRWCompressFileToStream (InStream: Stream;
205   OutStream: BinaryWriter);
```

```
206 var
207   sLength: Cardinal;
208   CompIdentifier: Cardinal;
209   DstSize: Word;
210 begin
211   try
212     CompIdentifier := UInt32(compressor.LZRWIdentifier);
213     OutStream.Write(CompIdentifier);
214     sLength := UInt32(InStream.Length);
215     OutStream.Write(sLength);
216     compressor.SourceSize := compressor.ChunkSize;
217     InStream.Seek(0,SeekOrigin(0));
218     while (compressor.SourceSize = compressor.ChunkSize) do  begin
219       compressor.SourceSize := UInt16(InStream.Read(compressor.Source,
220                                 0, compressor.ChunkSize));
221     DstSize := compressor.Compression();
222       OutStream.Write(DstSize);
223   OutStream.Write(compressor.Destination, 0, DstSize);
224     end;
225   except
226     OutStream.Close();
227     OutStream := nil;
228   end;
229 end ;
230
231 function  TWinForm.LZRWDeCompressFromStream (InStream: BinaryReader):
232   MemoryStream;
233 var
234   Result: MemoryStream;
235   OutStream: MemoryStream;
236   Identifier: Cardinal;
237   OrigSize: Cardinal;
238   DstSize: Word;
239 begin
240   OutStream := MemoryStream.Create;
```

```
241   try
242     InStream.BaseStream.Seek(0,SeekOrigin(0));
243     Identifier := InStream.ReadUInt32();
244     OrigSize := InStream.ReadUInt32();
245     DstSize := compressor.ChunkSize;
246     compressor.SourceSize := 0;
247     while (DstSize = compressor.ChunkSize) do
248       compressor.SourceSize :=  InStream.ReadUInt16();
249   InStream.Read(compressor.Source, 0, compressor.SourceSize);
250       DstSize := (compressor.Decompression());
251       OutStream.Write(compressor.Destination, 0, DstSize);
252     end;
253     OutStream.Seek(0,SeekOrigin(0));
254     Result := OutStream;
255   except
256     OutStream.Free;
257     Result := nil;
258   end;
259 end ;
260
261 procedure  TWinForm.TWinForm_Load (sender: Object; e: EventArgs);
262 begin
263 end ;
264
265 procedure  WinForm;
266 begin
267 end ;
268
269 end .
```

# Приложение Ж

# Справка о внедрении

«УТВЕРЖДАЮ»

Проректор ФГБОУ ВО «ИГУ»
по научной работе и
международной деятельности
профессор А.Ф. Шмидт

«_____» _____ 201__ г.

СПРАВКА О ВНЕДРЕНИИ

Выдана для предоставления в диссертационный Совет о том, что результаты кандидатской диссертационной работы Михайлова Андрея Анатольевича «Методы декомпиляции объектного кода Delphi» используются в учебном процессе на кафедре «Информационных технологий» ИМЭИ ИГУ в преподавании специальных курсов студентам 2-3 курсов дневного отделения.

Директор Института математики,
экономики и информатики ИГУ,
профессор, д.ф.-м.н. М.В. Фалалеев

Приложение 3

# Диплом за победу в конкурсе молодых учёных ИДСТУ СО РАН

Диплом

Федеральное государственное бюджетное учреждение науки
ИНСТИТУТ ДИНАМИКИ СИСТЕМ И ТЕОРИИ УПРАВЛЕНИЯ
имени В.М. МАТРОСОВА
Сибирского отделения Российской академии наук

НАГРАЖДАЕТ

*Михайлова*
*Андрея Анатольевича*

*за победу в конкурсе молодых ученых*
*«Результаты научной деятельности» в номинации*
*«Профессиональные достижения*
*в области информационных технологий»*
*на конференции «Ляпуновские чтения – 2015»*

Директор ИДСТУ СО РАН,
академик И.В. Бычков

30 ноября 2015 г.

Приложение И

# Диплом конкурса прикладных работ ИДСТУ СО РАН

ДИПЛОМ

Федеральное государственное бюджетное учреждение науки
ИНСТИТУТ ДИНАМИКИ СИСТЕМ И ТЕОРИИ УПРАВЛЕНИЯ
Сибирского отделения Российской академии наук

НАГРАЖДАЕТ

*Михайлова*
*Андрея Анатольевича*

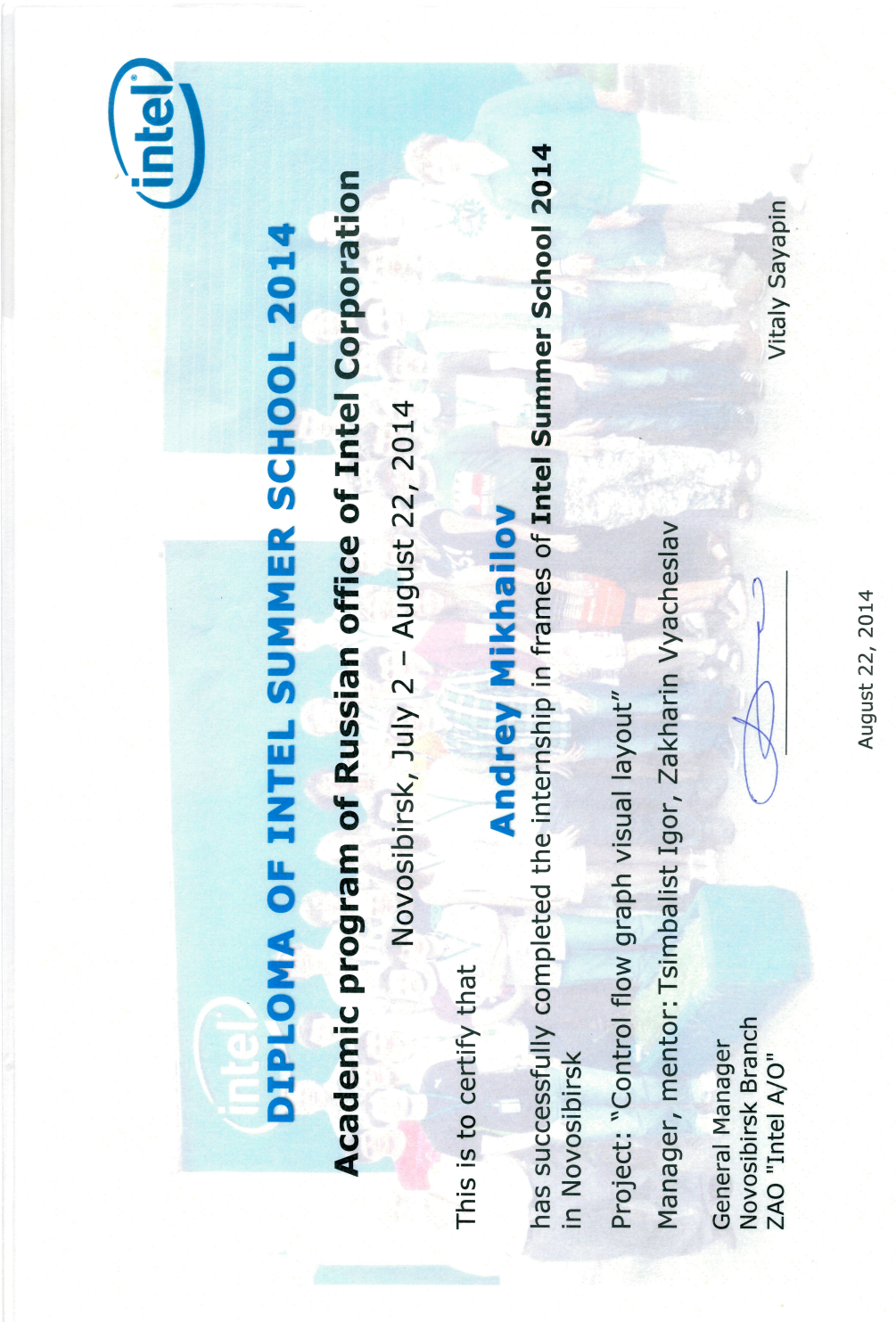
***за второе место***
***в конкурсе прикладных работ молодых ученых***
***на конференции «Ляпуновские чтения – 2014»***

Директор ИДСТУ СО РАН
академик И.В. Бычков

3 декабря 2014 г.

# Приложение К

# Диплом о прохождении стажировки



DIPLOMA OF INTEL SUMMER SCHOOL 2014

Academic program of Russian office of Intel Corporation

Novosibirsk, July 2 – August 22, 2014

This is to certify that **Andrey Mikhailov**

has successfully completed the internship in frames of **Intel Summer School 2014** in Novosibirsk

Project: "Control flow graph visual layout"

Manager, mentor: Tsimbalist Igor, Zakharin Vyacheslav

General Manager
Novosibirsk Branch
ZAO "Intel A/O"

Vitaly Sayapin

August 22, 2014

# Приложение Л

# Свидетельство о регистрации модуля структурной раскладки

РОССИЙСКАЯ ФЕДЕРАЦИЯ

**СВИДЕТЕЛЬСТВО**

**о государственной регистрации программы для ЭВМ**

**№ 2016612670**

**Модуль структурной раскладки графов потоков управления на плоскости для программы визуализации графов Visual Graph**

Правообладатель: ***Федеральное государственное бюджетное учреждение науки Институт динамики систем и теории управления имени В.М. МАТРОСОВА Сибирского отделения Российской академии наук (RU)***

Авторы: ***Хмельнов Алексей Евгеньевич (RU),***
***Михайлов Андрей Анатольевич (RU)***

Заявка № **2016610127**
Дата поступления **12 января 2016 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***03 марта 2016 г.***

*Руководитель Федеральной службы*
*по интеллектуальной собственности*

*Г.П. Ивлиев*

# Приложение М

# Свидетельство о регистрации DCUIL2PAS

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2014617137

«DCUIL2PAS - декомпилятор объектных модулей Delphi, скомпилированных под платформу .NET (файлов *.DCUIL)»

Правообладатель: *Федеральное государственное бюджетное учреждение науки Институт динамики систем и теории управления Сибирского отделения Российской академии наук (ИДСТУ СО РАН) (RU)*

Авторы: *Хмельнов Алексей Евгеньевич (RU), Михайлов Андрей Анатольевич (RU)*

Заявка № 2014614007

Дата поступления 30 апреля 2014 г.

Дата государственной регистрации в Реестре программ для ЭВМ *14 июля 2014 г.*

*Руководитель Федеральной службы по интеллектуальной собственности*

*Б.П. Симонов*