РОССИЙСКАЯ АКАДЕМИЯ НАУК
СИБИРСКОЕ ОТДЕЛЕНИЕ
ИНСТИТУТ ДИНАМИКИ СИСТЕМ И ТЕОРИИ УПРАВЛЕНИЯ

На правах рукописи
УДК: 681.3.06

Хмельнов Алексей Евгеньевич

Язык FlexT для спецификации бинарных форматов данных.

05.13.11 - Математическое и программное обеспечение
вычислительных машин, комплексов, систем и сетей

Диссертация на соискание учёной степени
кандидата технических наук.

Научный руководитель
чл.-корр. РАН С.Н. Васильев

Иркутск - 2000

АННОТАЦИЯ

диссертации А.Е. Хмельнова

Язык FlexT для спецификации бинарных форматов данных.

Разработан язык спецификации бинарных форматов данных FlexT.

Реализован ряд программных систем с использованием интерпретатора языка FlexT: программа просмотра файлов различных форматов BinView, её Интернет-версия WWWBinView и дизассемблер 32-разрядных исполняемых файлов Windows - PE Explorer.

Показана применимость языка FlexT для описания широкого круга бинарных форматов.

СОДЕРЖАНИЕ

Введение

1. Актуальность задачи описания форматов бинарных данных.

Современная цивилизация с каждым годом всё больше зависит от компьютеров. В свою очередь, основной интеллектуальный багаж вычислительной техники сосредоточен не столько в аппаратуре, сколько в том огромном количестве информации, которое накоплено в электронном виде за несколько десятилетий развития этой области знаний. Те усилия, которые потребовались для устранения проблемы 2000 года, показывают, насколько ещё плохо контролируемой является эта информация: работа жизненно важных процессов может зависеть от программ, исходные тексты которых отсутствуют и работоспособность которых приходится принимать на веру. То же относится и к другим видам данных: очень часто файл в некотором формате является "чёрным ящиком" - последовательностью неизвестно для чего предназначенных байтов, заглянуть в который можно лишь при помощи специально для этого предназначенной программы, если таковая вообще существует. При этом оказывается невозможным проконтролировать его содержимое другими способами, например, если специализированная программа отказывается работать с некоторым файлом. Таким образом, очень актуальной сейчас является проблема автоматизации обработки данных различных форматов, включая и исполняемые.

Для того, чтобы отображать содержимое файлов различных форматов, могут использоваться специализированные программы, которые вызываются из универсальной программы просмотра (пример: QuickView в Windows). Для обработки данных одного вида, например, растровых графических файлов, в некоторых программах используются подключаемые динамические библиотеки, предоставляющие определённый набор функций, которые могут быть написаны независимыми производителями для заранее неизвестных форматов файлов (пример: plug-ins в Adobe Photoshop). Также могут существовать готовые статические библиотеки (*.lib) или исходные тексты для работы с некоторыми форматами. Недостатком всех этих подходов является то, что они фиксируют способ обработки данных и скрывают информацию о структуре поддерживаемых форматов, смешивая её в коде со вспомогательными и специализированными операциями. Хотелось бы иметь возможность представления информации о форматах данных в таком виде, который допускает её многоцелевое использование и который бы содержал как можно меньше несущественной, т.е. не относящейся к способу представления данных, информации. Данная работа показывает, что для этих целей может использоваться язык спецификации форматов данных.

Автору неизвестны успешные решения задачи описания форматов бинарных данных. Более того, не приходилось встречать и постановки

подобных задач с предлагаемой в данной работе степенью универсальности. Наиболее близкой по постановке является работа **[1]**. Там автор поставил себе цель описать формат DWG (AutoCAD), а, может быть, и какие-то другие при помощи специального вида контекстно-свободных грамматик, в которых терминальными символами являются примитивные типы данных. К сожалению, данная работа, судя по всему, не увенчалась успехом и сейчас ссылка **[1]** устарела, а следы этой работы можно найти под именем BFF на сайте **[2]**.

Следует заметить, что для описания способов кодирования бинарных данных возможностей грамматик, предназначенных для формализации описания синтаксиса языков программирования, явно недостаточно. Это объясняется тем, что в текстовых файлах не используется возможность адресации, а, значит, отсутствуют аналоги таких понятий, как "указатель" или "размер", без использования которых нельзя представить практически ни один формат. При помощи формальных грамматик нельзя даже потребовать, чтобы число фактических параметров в вызове функции совпало с числом её формальных параметров. Таким образом, если представлять себе язык для описания форматов бинарных данных, как результат расширения возможностей формальных грамматик, то такое расширение должно быть очень значительным.

По пути такого расширения пошли авторы проекта **[3]**, которые являются признанными авторитетами в вопросах универсальной (независимой от компилятора) декомпиляции. В проекте **[3]** ставится задача автоматического перевода исполняемых файлов с одной платформы на другую. Для решения этой задачи необходимо уметь считывать во внутреннее представление исполняемые файлы различных форматов. На более старой версии этой страницы находилась ссылка на работу **[1]**, как на единственную известную авторам попытку спецификации форматов бинарных файлов. Эта ссылка сопровождалась замечанием, что "эта грамматика сильно зависит от формата файлов AutoCAD`a и в настоящее время непригодна для решения задачи загрузки файлов". В текущей версии страницы **[3]** имеется ссылка на собственную работу авторов этой страницы, также названную BFF **[4]**, которая является попыткой развития идей из **[1]** применительно к поставленной задаче. С использованием спецификации формата файла автоматически генерируется код для считывания в память и запоминания адресов некоторых частей файла, таких, как его заголовок или загружаемый образ памяти. После этого в сгенерированном коде что-то необходимо исправить, а остальное считывание необходимо дописать вручную, т.е. используемые описания не являются полными. Если выразить возможности предложенного в проекте **[3]** языка в терминах языка FlexT, рассматриваемого в настоящей диссертации, то можно сказать, что там в какой-то степени реализованы всего два конструктора типов: "запись" и "массив", а также поддерживаются динамические адреса переменных, т.е. этих возможностей явно недостаточно для полного описания даже самого простого формата данных.

В заключение этого по необходимости краткого обзора предшествующих работ следует сказать несколько слов о "конвергенции" терминологии и ложных предшественниках. Дело в том, что в данной работе используются такие термины, как "элемент данных", "интерпретация", "абстракция" и т.д., которые могут напомнить невнимательному читателю терминологию проблематики моделирования и описания хранимых данных в технологии баз данных, например, языка CODASYL [5], что может существенно помешать пониманию данной работы. По отношению к таким задачам язык FlexT рассматривает вопросы гораздо более низкого уровня представления информации, чем уровень моделей данных в СУБД, которые не опускаются до описания физического представления данных. Кроме того, при описании моделей данных СУБД рассматриваются только вопросы хранения информации в СУБД, а не способы представления произвольных видов данных. "Конвергенция" терминологии происходит потому, что между описываемыми сущностями возникают похожие отношения (например, как ещё можно назвать составную часть более сложных данных, как не элементом данных?).

2. Пример: разбор файла в формате DBF.

Для иллюстрации сути предлагаемого подхода рассмотрим следующий простейший пример. На Рис. 1 представлено содержимое небольшого файла в формате DBF **[6], [7]** так, как оно отображается при просмотре в клоне Norton Commander`a (при нажатии F3, F4).

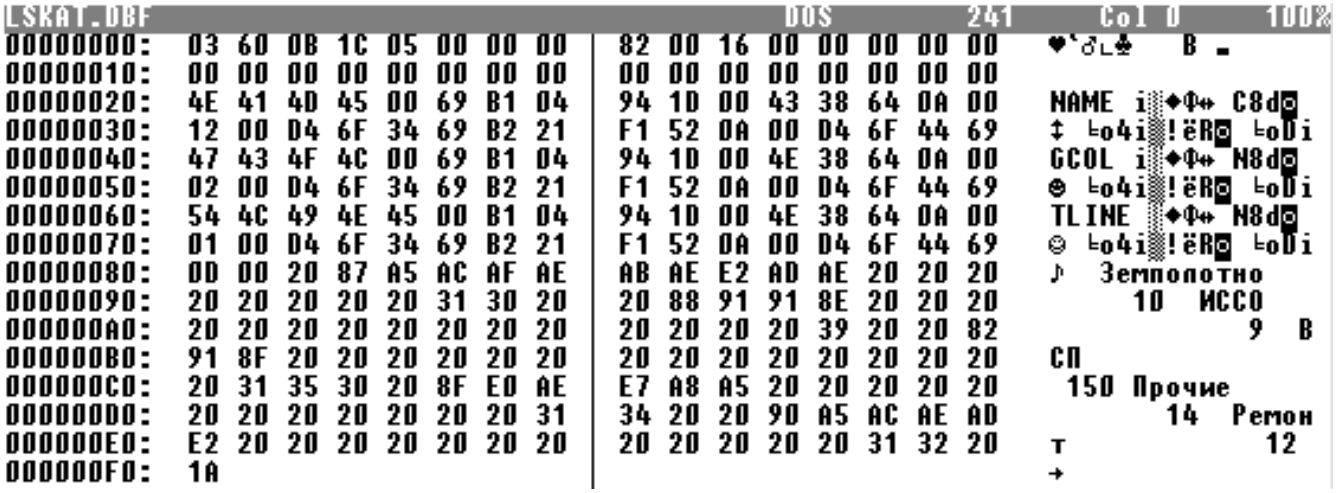


Рис. 1. Дамп файла LSKAT.DBF при просмотре в FAR`e.

В этом файле закодирована информация о таблице, представленная на Рис. 2.

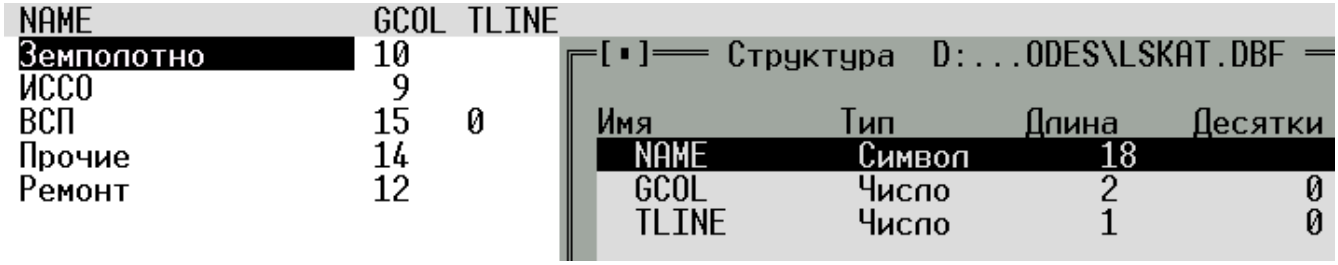


Рис. 2. Информация, содержащаяся в файле LSKAT.DBF.

Именно с этим уровнем абстракции имеет дело подавляющее большинство из тех, кто использует данные в формате DBF.

Если, например, спросить любого программиста о том, почему на Рис. 1 в байте со смещением 0xA записано значение 0x16, то, даже если он вчера написал программу для чтения данных из формата DBF, вряд ли он сможет дать ответ на этот вопрос, не заглядывая в документацию.

На Рис. 3 представлено описание на FlexT формата DBF, а на Рис. 4 - результат разбора файла LSKAT.DBF согласно этому описанию программой BinView. В результате разбора файла, с одной стороны, как и на Рис. 1, было отображено всё его содержимое, а, с другой стороны, в полученном представлении легко читается информация, представленная на Рис. 2. Таким образом, данные файла становится прозрачными, доступными для понимания человеком.

При наличии спецификаций, написанных на FlexT, подобные результаты можно получить и для гораздо более сложных, чем DBF, форматов.

```
type
  TBinDate array[3] of Byte //Дата в двоичном формате(ГГММДД)
  TDBF3FldKind enum Char (
    fkChar='C', fkNumeric = 'N', fkLog = 'L',
    fkDate = 'D', fkMemo = 'M'
  )

  TDBF3FldDsc struc
    array[11] of Char,0; Name //Имя - ASCIIZ строка
    TDBF3FldKind hType
    ulong DataP     //like Delphi Tag
    Byte Len        //Длина поля в байтах
    Byte DecNum     //Число знаков после точки
    Word MUsrRsrv1  //Зарезерв. для многопольз. систем
    Byte WorkID     //ID рабочей области
    Word MUsrRsrv2  //Зарезерв. для многопольз. систем
    Byte SetFldData //Используется командой SET FIELDS
    array[8] of Byte Reserved
  ends

  PDataArray ^TDataArray near

  TDBF3Hdr struc
    Byte Ver //$02 - dBase II, $03 - dBase III
             //$83 - dBase III с Memo-полями
    TBinDate LastChangeDate
    ulong RecCnt      //Число записей
    PDataArray HdrLen //Длина заголовка в байтах
    Word RecLen       //Длина записи в байтах
    (array[20]of Byte) Reserved
  ends

  TDBF3HdrWithFields struc
    TDBF3Hdr H
    array[(@.H.HdrLen-@.H:Size-1) div 32] of
      TDBF3FldDsc Fields
  ends

data
  0x0000 TDBF3HdrWithFields Hdr

type
  TFieldData array[Hdr.Fields[#].Len]of Char
  TFieldsData array of TFieldData :[@:Size=Hdr.H.RecLen-1]
  TRecData struc
    Char F
    TFieldsData D
  ends
  TDataArray array[Hdr.H.RecCnt] of TRecData
```

Рис. 3. Описание на FlexT формата DBF.

```
0000:Hdr: TDBF3HdrWithFields = (
  H:(Ver:03; LastChangeDate: (0:60,1:0B,2:1C); RecCnt:00000005;
    HdrLen:0082; RecLen:0016;
    Reserved: (0:00,1:00,2:00,3:00,4:00,5:00,6:00,7:00,8:00,9:00,
      10:00,11:00,12:00,13:00,14:00,15:00,16:00,17:00,18:00,
      19:00));
  Fields: (
    0:(Name:'NAME'; hType:fkChar{'C'}; DataP:000A6438; Len:12;
      DecNum:00; MUsrRsrv1:6FD4; WorkID:34; MUsrRsrv2:B269;
      SetFldData:21;
      Reserved: (0:F1,1:52,2:0A,3:00,4:D4,5:6F,6:44,7:69)),
    1:(Name:'GCOL'; hType:fkNumeric{'N'}; DataP:000A6438;
      Len:02; DecNum:00; MUsrRsrv1:6FD4; WorkID:34;
      MUsrRsrv2:B269; SetFldData:21;
      Reserved: (0:F1,1:52,2:0A,3:00,4:D4,5:6F,6:44,7:69)),
    2:(Name:'TLINE'; hType:fkNumeric{'N'}; DataP:000A6438;
      Len:01; DecNum:00; MUsrRsrv1:6FD4; WorkID:34;
      MUsrRsrv2:B269; SetFldData:21;
      Reserved: (0:F1,1:52,2:0A,3:00,4:D4,5:6F,6:44,7:69))))
    0080:0D 00 ¦..¦
0082:Hdr.H.HdrLen^: TDataArray =  (
  0:(F:' '; D: (0:'Земполотно         ',1:'10',2:' ')),
  1:(F:' '; D: (0:'ИССО               ',1:' 9',2:' ')),
  2:(F:' '; D: (0:'ВСП                ',1:'15',2:'0')),
  3:(F:' '; D: (0:'Прочие             ',1:'14',2:' ')),
  4:(F:' '; D: (0:'Ремонт             ',1:'12',2:' ')))
    00F0:1A ¦.¦
```

Рис. 4. Результат разбора файла LSKAT.DBF программой BinView.

3. Цели и структура работы.

Целью работы является разработка универсального (с некоторыми ограничениями) декларативного языка для описания бинарных (т.е. нетекстовых) форматов данных, а также реализация программной системы, использующей описания форматов данных на этом языке для отображения содержимого файлов различных форматов в пригодном для восприятия человеком виде.

Структура работы. Диссертационная работа состоит из введения, четырёх глав, заключения, литературы и приложения.

В **первой главе** - приведена классификация подходов, используемых для облегчения обмена данными между различными программными системами, обзор некоторых существующих методов спецификации и обработки данных различных форматов, введена классификация языков спецификации форматов данных по их возможностям, введено понятие идентификации типов данных, определены главные цели разработки языка FlexT, даны определения динамических и статических типов данных и сформулирован подход, позволяющий избежать проблемы, связанные с использованием статических типов данных для описания форматов хранения информации.

Вторая глава описывает основные принципы языка FlexT, обосновывает необходимость введения базовых конструкторов типов данных, вводит

нотацию, используемую при описании синтаксиса языка, раскрывает общие особенности используемого механизма определения типов, описывает вид используемых в языке выражений, определяет общую структуру программы на языке FlexT, рассматривает основные конструкторы типов данных и приводит примеры их использования, приводит список предопределённых типов языка FlexT, а также рассматривает ограничения предлагаемого подхода.

В **третьей главе** описаны некоторые детали реализации языка FlexT: механизм вызова типов, структуры данных, используемые для представления информации о типах данных и переменных, методы работы с блоками памяти, обработка типов данных с битовым размещением, процесс исполнения программы FlexT и автоматического определения формата файла в программах просмотра, технология размеченных потоков вывода, способ отображения информации о перемещаемых адресах в шестнадцатеричном дампе, алгоритм построения дерева решений для обработки перечислений термов.

В **четвёртой главе** рассматриваются вопросы использования языка FlexT. Описывается работа программ, в состав которых входит интерпретатор языка: разбор файлов в программе BinView, удалённая обработка файлов в программе WWWBinView, Просмотр 32-разрядных файлов Windows в программе PE Explorer. Далее рассматриваются примеры описаний форматов: файла класса Java-машины и кодирования машинных команд процессора Z-80 (Intel-8080). После этого рассматривается использование описаний RTTI Delphi при дизассемблировании программ, написанных на Delphi, а также опыт реконструкции 32-разрядных объектных файлов Delphi (DCU).

В заключении приводится перечень результатов, выносимых на защиту.

4. Научная новизна, практическая значимость и апробация полученных результатов.

Научная новизна. Предложен новый язык для спецификации форматов данных FlexT (**Flex**ible **T**ypes), который позволяет описать широкий набор форматов данных. Реализованы программные системы, использующие эти описания для отображения соответствующих им данных.

Практическая значимость полученных результатов. Наличие спецификации формата данных на языке FlexT позволяет автоматически разобрать содержимое данных, соответствующих этому формату, и представить закодированную в нём информацию в пригодном для восприятия человеком виде. Решение этой задачи является необходимой предпосылкой для решения широкого круга других задач, связанных с автоматизацией разработки программ, таких как автоматическая генерация кода для работы с данными по спецификации их представления или спецификация семантики блоков исполняемого кода (т.к. для спецификации семантики исполняемого кода может потребоваться описание форматов данных, с которыми он работает).

При переводе спецификаций форматов, написанных на естественном языке на язык FlexT, в большинстве из них были обнаружены ошибки. Таким

образом, язык FlexT является средством точной формальной спецификации форматов данных, с возможностью непосредственной проверки этих спецификаций на реальных данных. Переход от описания для человека к описанию для машины, которое можно сразу же проверить на настоящих данных, приводит к существенному повышению достоверности информации, и в этом состоит один из наиболее важных результатов применения рассматриваемого языка.

Каждый формат данных можно считать "искусственным языком", предназначенным для представления некоторого определённого вида информации, например, исполняемых файлов или изображений, а программы, способные обрабатывать такие форматы, - носителями этих языков. Описание формата данных на языке FlexT можно рассматривать как некоторый словарь искусственного языка, который позволяет "наладить общение", т.е. сделать понятной закодированную информацию как для человека, так и, в перспективе, для других программ. Таким образом, данная работа может также рассматриваться, как исследование лингвистики искусственных языков.

Апробация работы. Результаты диссертационной работы докладывались на конференциях по информатике: III Международный симпозиум "Интеллектуальные системы" (ИНТЕЛС'98, Псков, 1998), The 3rd IMACS International Multiconference on: Circuits, Systems, Communications and Computers (CSCC'99), а также на семинарах ИДСТУ по информатике.

Публикации. Основные результаты диссертации опубликованы в 4 печатных работах (**[8]**, **[9]**, **[10]**, **[11]**).

Главы

1. Общие принципы предлагаемого подхода

1.1. Проблемы обработки данных и способы их решения.

Для хранения и передачи информации было разработано огромное количество форматов её представления, причём, количество форматов, используемых для представления данных одного класса, может исчисляться десятками, если не сотнями. В качестве примера такого класса данных можно назвать класс файлов растровой графики.

В результате значительную часть усилий при составлении программ для работы с данными определённого класса приходится тратить на написание кода для считывания информации из различных форматов этого класса во внутреннее представление и, если это необходимо, записи изменённых данных обратно на диск в требуемом формате. В принципе, при наличии программ-конвертеров, достаточно реализовать обработку хотя бы одного формата, но при таком подходе усложняется работа пользователя, требуется распространять дополнительную программу-конвертер, да и обработку хотя бы одного формата всё равно придётся написать.

Существование библиотек для работы с каждым форматом, как правило, не избавляет от переписывания бóльшей части их кода, поскольку каждая из таких библиотек может считывать данные лишь в своё внутреннее представление, которое, как правило, отличается от используемого в разрабатываемой программе. Таким образом, программист вынужден в очередной раз читать описание формата или код библиотеки для работы с ним и, в который уже раз, выписывать операторы открытия файла, проверки его существования, проверки соответствия формату, считывания блока данных из файла и т.д. и т.д. При этом он, в сущности, с некоторыми вариациями повторяет ту же работу, которую проделывали до него тысячи его предшественников, не создавая при этом ничего принципиально нового, поскольку вся необходимая информация уже содержалась в описании формата или в исходных текстах библиотеки для работы с ним - просто она была представлена в неявном виде: записана на естественном языке или разбросана по коду, написанному на определённом языке программирования *для определённого способа работы с этими данными*.

Самым существенным недостатком такого положения является даже не то, что при этом тратится время на повторение уже много раз проделанной работы, но то, что при этом в программу могут быть внесены ошибки, как результат невнимательности или неправильного понимания спецификации.

Какие же подходы можно применить для улучшения ситуации?

1) Универсальный обменный формат.

В предметных областях, в которых ещё не устоялся традиционный набор достаточно универсальных обменных форматов, усилия часто сосредотачиваются на разработке такого формата. В качестве примера можно привести область ГИС и попытку внедрения там формата SDTS. Предполагалось, что каждое приложение ГИС должно было поддержать универсальный обменный формат, после чего проблема обмена данными будет решена. На примере SDTS можно видеть, что такой формат получается очень сложным, поскольку он должен поддерживать все возможности других существующих форматов. Записать информацию в этот формат достаточно легко, поскольку при этом используется его подмножество, соответствующее реализованным в данной системе возможностям, но прочитать оттуда данные гораздо сложнее, поскольку они могут быть записаны в соответствии с не реализованными в данной системе возможностями другой системы.

Ещё более серьёзной сложностью является невозможность заранее предсказать развитие данной предметной области, что может привести к необходимости пересмотра даже самого универсального на данный момент обменного формата. Также существенным недостатком обменных форматов является то, что они, как правило, являются очень громоздкими и сильно уступают в эффективности форматам конкретных систем, порождая файлы очень большого объёма. Хорошим примером для иллюстрации этого факта является формат PostScript: приходилось встречать такие файлы, объём которых превосходит объем полученного из них файла печати с разрешением 300 dpi для лазерного принтера (не PostScript'овского).

2) Самодокументированные форматы.

Чтобы решить проблему ограниченности возможностей обменного формата, предлагается включать в файл всю необходимую для его интерпретации метаинформацию. При этом программа, которая читает такой файл, должна уметь использовать метаинформацию для считывания и интерпретации (перевода во внутреннее представление) той части данных из файла, для работы с которой она предназначена. Этот подход пропагандируется, например, для обмена данными физических экспериментов.

Недостатками самодокументированных форматов является то, что требуются конвертеры для перевода уже существующих файлов в этот формат, а также тот факт, что метаинформация дублируется в каждом файле, даже если все файлы содержат данные одинаковой структуры.

В принципе, данный подход можно считать особым подклассом универсальных обменных форматов. Тот же PostScript можно отнести и в этот подкласс: он содержит заголовок с описанием функций и шрифтов, используемых при описании собственно текста, причём, этот заголовок занимает значительную часть файла (а иногда и бóльшую часть) и, как правило, дублируется без изменений во всех созданных одним текстовым редактором файлах.

3) Языки описания форматов данных.

От упомянутых недостатков файлов, содержащих в себе метаинформацию, можно естественным образом избавиться, выделив эту информацию в отдельный файл. При этом, чтобы программа, использующая такой подход, смогла работать с неизвестным ранее форматом, будет достаточно предоставить ей описание этого нового формата, как представления данных того класса, для работы с которым программа предназначена.

Весь вопрос здесь состоит в разработке такого языка спецификации форматов данных, который поддерживал бы работу с рассматриваемым классом форматов файлов, причём, желательно, чтобы этот класс был достаточно широким - тогда конкретная программа сможет использовать для себя его конкретное подмножество. В то же время, при этом следует учитывать, что чем более универсальным будет такой язык, тем он будет сложнее - здесь надо суметь вовремя остановиться, чтобы полученный язык имел право считаться языком спецификации, а не просто очередным вариантом языка программирования.

1.2. Спецификация интерпретации и спецификация изменения.

По возможностям работы с описываемыми данными можно различать два уровня языков спецификации данных: *спецификация интерпретации* и *спецификация изменения*.

1.2.1. Язык спецификации интерпретации.

Язык спецификации интерпретации должен позволять *легко* описывать *интерпретацию* произвольного (в рамках определённых ограничений) класса данных.

Под интерпретацией здесь понимается не преобразование всех данных в некоторый конкретный формат, а наличие в спецификации информации о способах извлечения из описанных данных значений свойств рассматриваемого класса данных. Например, спецификация формата файла растровой графики должна позволять извлечь из такого файла информацию о размерах изображения (ширина, высота) и цвете каждого пиксела, расположенного в границах изображения. Для повышения эффективности спецификации она может быть дана в терминах менее абстрактного класса данных, для которого уже существует спецификация, соотносящая его с более абстрактным. Например, более эффективное описание данных растровой графики может давать размеры, глубину (число бит на пиксел), таблицу цветов и функцию получения индекса цвета по координатам пиксела, ещё более эффективное описание вместо последней функции может возвращать всю строку растра и т.д.

1.2.2. Язык спецификации изменения.

Язык спецификации изменения должен, кроме определения функций-наблюдателей для считывания информации, давать определения конструкторов, позволяющих создавать по заданным свойствам новые экземпляры данных. При этом приходится учитывать дополнительные детали, например, распределение памяти, порядок порождения элементов данных, соглашения по их выравниванию, способ заполнения пропусков, и т.д.

В идеальном случае язык спецификации изменения должен предоставлять всю необходимую информацию для перевода данных из одного формата в любой другой описанный формат, в том числе и во внутреннее представление некоторой программы. Сам язык описания не обязан быть очень эффективным по скорости интерпретации. Для повышения эффективности может быть реализована возможность генерации по спецификации данных кода для работы с ними. Так, для чтения во внутреннее представление из некоторого формата, по спецификации этого формата и спецификации внутреннего представления могла бы быть сгенерирована процедура чтения, которая за счёт оптимизации не порождает промежуточного представления данных. Данный уровень языков спецификации далее не будет рассматриваться - он приводится здесь в качестве возможного направления продолжения исследований в рассматриваемой области.

1.3. Требования к языку спецификации.

В книге **[12]** отмечается, что "понятие спецификации относительно, типичная спецификация, отличается от типичной программы бóльшей понятностью, бóльшей адекватностью используемых в ней средств описываемой задаче, более высоким уровнем". Кроме того, там же упомянуто следующее соображение, которое имеет прямое отношение к языку FlexT: если язык спецификации как-то *реализован*, то он является также и языком программирования. Таким образом, понятие языка спецификации является скорее неформальным, поэтому его нельзя точно определить, а можно лишь уточнять при помощи требований к такому языку.

Слова "легко", "простота" и т.п. часто употребляется при определении понятия "спецификация". Подразумевается, что запись некоторой информации на языке спецификации должна быть проще, чем на обычном языке программирования. Более существенным здесь, по мнению автора, является не субъективное понятие простоты, а отсутствие в спецификации избыточной информации. Т.е. надо стремиться к тому, чтобы в описании данных приходилось давать как можно меньше информации, являющейся следствием ранее заданной (система, читающая описание, должна воспринимать информацию не как студент, которому всё нужно объяснять дважды и без которого, поэтому, проще вообще обойтись, а как грамотный специалист, который всё понимает с полуслова). Также желательно, чтобы не приходилось вдаваться в излишние подробности, например, если на самом деле не важно в каком порядке надо обрабатывать элементы массива, то код, который содержит

цикл `for i:=0 to N-1 do`, можно считать избыточным. Именно поэтому язык спецификации должен быть как можно более декларативным: он должен описывать то, *как размещаются данные, а не то, как их надо читать*.

1.4. Используемые термины.

Для устранения проблем с пониманием оставшейся части данной работы следует определиться с используемой в ней терминологией. Некоторые из определяемых здесь понятий будут рассмотрены в дальнейшем более подробно.

Типом данных будем называть полную информацию о способе интерпретации непрерывной области памяти. Слово "полная" в этом определении подчёркивает, что все сведения, которые необходимо иметь о некоторой области памяти для того, чтобы правильно разделить её на составные части, определить её размер и т.д., содержатся в её типе данных.

Под *элементом данных* будем понимать непрерывную область памяти с известной интерпретацией. Таким образом, элемент данных характеризуется своим адресом и типом, а, например, его размер уже определяется по типу данных. Элемент данных может быть *составным*, т.е. в его области памяти могут выделяться элементы данных меньшего размера. Наличие и типы этих составляющих зависят от типа элемента данных.

Переменной будем называть *глобальный* элемент данных, т.е. такой элемент данных, который не входит в состав другого элемента. Этот термин может быть не совсем удачен, так как в контексте данной работы "переменная" не меняет своего содержимого, однако этот термин может стать более подходящим при реализации спецификаций изменения. Если некоторый элемент данных является составляющей другого элемента, будем называть его *локальным*.

Слово "*динамический*" будем использовать для обозначения зависимости свойств типа, константы и т.д. от конкретных данных, т.е. эти свойства могут принимать разные значения на разных элементах данных одного типа или на разных файлах одного формата. Соответственно, *статическими* будем называть свойства, не зависящие от конкретных данных.

1.5. Идентификация типов данных, как базовая интерпретация данных.

В качестве базовой интерпретации, пригодной для описания любого формата данных, можно предложить такую интерпретацию, которая выделяет все существующие элементы данных, т.е. определяет их адреса и приписывает им типы - выполняет *идентификацию типов данных*. Такая интерпретация является базовой и минимальной, поскольку при использовании некоторой области данных в интерпретации более высокого уровня обязательно придется определить, к какому типу она относится.

В качестве класса данных, в который они отображаются при идентификации типов, используется набор взаимосвязанных *элементов*

данных. Каждый элемент данных характеризуется своим размещением (*адресом* и *размером*) и *типом*, и может содержать ссылки на другие элементы данных, а также необходимую для полного описания других элементов данных информацию. Размер элемента данных определяется его типом. Элементы данных могут быть составными - в этом случае внутри них можно выделить элементы меньшего размера.

Элемент данных можно сравнить с термом, а идентификацию типов данных - с эрбрановой интерпретацией в первопорядковой логике **[13]**.

Интерпретация типов данных может быть *неполной*. В этом случае она содержит элементы данных, которым не сопоставлен тип или, точнее, в этом случае можно считать, что им сопоставлен примитивный тип - *сырые данные* (см. п. 2.5.10).

1.6. Динамические и статические типы данных.

Под статическими типами данных здесь мы будем понимать аналоги большинства традиционных типов данных, реализованных в процедурных языках программирования. Их отличительной особенностью является то, что размер элемента данных такого типа, а также набор и внутреннее размещение составляющих его элементов определены в момент компиляции и не зависят от конкретных данных.

В процедурных языках программирования, по крайней мере, в тех, которые реально используются в настоящее время, составные типы данных могут содержать только статические составляющие.

Размер элемента данных динамического типа и внутреннее размещение его составляющих могут зависеть от конкретных данных. Далее слово *"динамический"* будет использоваться именно в этом смысле. Примером динамических типов в традиционных процедурных языках являются строковые константы, как паскалевские, так и ASCIIZ: чтобы определить занимаемый ими размер нужно в первом случае прочитать первый байт, а во втором - просмотреть всю строку в поисках нулевого символа.

Тип	Статический	Динамический
Размер	Фиксирован	Может зависеть от данных
Смещение составляющих	Фиксировано	Может зависеть от данных
Назначение	Тип переменных	Тип констант

Примеры на Паскале	`TName = array[0..31] of Char;`
	`Tid = string[32]`
		`const CP: PChar = 'Hello';`
		`var S: String; ...`
		`S := 'Hello';`

Таблица 1. Сравнение динамических и статических типов данных.

Понятно, что динамические типы данных непригодны в качестве типов переменных, по крайней мере, если этот тип изменяемый (в терминах **[14]**),

поскольку присваивание новых значений элементам таких данных может означать изменение размера, а такие операции ни один компилятор не сможет эффективно поддержать. Для полей типов данных, содержащих те же строки или записи с вариантами, сразу выделяется максимально необходимый размер памяти. Примером поддержки компилятором динамического перераспределения памяти, занимаемой значением переменной, являются huge-строки 32-разрядных версий **Delphi**, память под которые автоматически запрашивается в куче, при этом сами переменные такого типа фактически являются указателями и всегда занимают 4 байта.

В то же время, если рассматривать статические данные, которые программа может только читать, то здесь иногда используются весьма изощрённые способы их кодировки, например, при помощи ассемблерных макрокоманд.

В качестве примера статических данных в коде программы можно привести RTTI Delphi - способ представления метаинформации о типах данных (фрагмент файла **TypeInfo.pas [15]**):

```
PTypeInfo = ^TTypeInfo;
TTypeInfo = record
  Kind: TTypeKind;
  Name: ShortString;
 {TypeData: TTypeData}
end;
PTypeData = ^TTypeData;
TTypeData = packed record
  case TTypeKind of
```

Комментарий в записи `TTypeInfo` подсказывает, что поле `TypeData` идёт непосредственно после последнего символа строкового поля `Name`, размер которого зависит от длины имени конкретного типа. Поскольку смещение поля `TypeData` зависит от конкретных данных - имени типа, такую структуру нельзя непосредственно представить на Паскале - приходится писать специальный код для доступа к этому полю (фрагмент того же файла):

```
function GetTypeData(TypeInfo: PTypeInfo): PTypeData;
  assembler;
asm
  { ->  EAX Pointer to type info }
  { <-  EAX Pointer to type data }
  {it's really just to skip the kind and the name }
  XOR  EDX,EDX
  MOV  DL,[EAX].TTypeInfo.Name.Byte[0]
  LEA  EAX,[EAX].TTypeInfo.Name[EDX+1]
end;
```

Подобные трудности испытывают все авторы книг по форматам файлов данных при попытке описать эти данные, например, на языке **C**, как в **[16]**.

Причина всех этих проблем - *в использовании для спецификации форматов данных языка, предназначенного для описания типов переменных*. В то же время, если отвлечься от необходимости придерживаться ограничений на типы

переменных, связанных с возможностью присваивания новых значений, то можно естественным образом поддержать в языке описание достаточно сложных зависимостей между элементами данных.

Для сравнения приведём описание того же типа данных `TTypeInfo` на языке FlexT:

```
TTypeInfo struc
  TTypeKind Kind
  Str Name
  TTypeData TypeData
ends
```

1.7. Использование механизма определения типов данных.

Таким образом, в качестве основного элемента языка спецификации форматов данных мы будем рассматривать механизм определения типов. В процедурных языках программирования механизм определения типов можно считать отдельным вложенным языком, т.к. от определений типов зависит, например, семантика и синтаксис процедур для работы с данными этих типов, но не наоборот, т.е. если выбросить из программы весь исполняемый код, то это не помешает понять, каким образом в ней размещаются данные. Даже в объектно-ориентированных языках структуры данных, используемые для представления объектов, никак не зависят от кода методов этих объектов.

Если в процедурных языках программирования основная информация программы содержится в кодах процедур, то при описании способов хранения информации основная нагрузка ложится именно на механизм определения типов. Рассмотрим, какими возможностями должен обладать такой механизм.

1.8. Необходимые конструкции.

При идентификации типов данных в разбираемой памяти должны быть выделены переменные, т.е. глобальные элементы данных, которые, в свою очередь, могут состоять из более мелких элементов данных. Каждый элемент данных характеризуется своим адресом, размером и интерпретацией (типом). Составные элементы данных разбиваются на более мелкие элементы. Для определения правильной интерпретации одних элементов данных может использоваться информация, закодированная в других элементах данных. В таком случае будем говорить о существовании *зависимости* определяемых элементов данных от элементов данных, предоставляющих такую информацию.

Всю информацию, которую должна предоставить спецификация идентификации типов данных о каждом элементе данных, можно разбить на набор утверждений о том:

а) на какие составляющие, каких типов этот элемент разбивается;

б) где находятся эти составляющие (каковы их адреса);

в) сколько места они занимают.

При этом из того, что в рассматриваемых данных содержится некоторый элемент составного типа, выводится информация о типах и размещении его составляющих (из которой, в свою очередь, может выводиться информация о размере этого составного элемента).

Любые данные, относящиеся к некоторому формату, являются результатом исполнения некоторого *алгоритма* записи информации. Распространение методологии структурного программирования сделало широко известным тот факт, что любой алгоритм может быть представлен при помощи стандартных структур управления: последовательность операций, условный оператор, цикл. Это утверждение относится и к алгоритму записи.

Результатам вызова процедур вывода из различных управляющих конструкций можно сопоставить базовые конструкторы типов данных:

- последовательности с фиксированным числом команд вывода соответствует запись с фиксированным числом полей (п. 2.5.6);
- условному оператору соответствуют конструкторы: варианта (п. 2.5.7), проверки (п. 2.5.8) и перечисления термов (п. 2.5.4.3). Сопоставление трёх различных конструкторов условному оператору объясняется тем, что для определения выбранного типа может использоваться как внешняя информация (вариант), так и анализ структуры разбираемых данных (проверка и перечисление термов);
- вызову команд вывода в цикле соответствует массив (п. 2.5.9).

На самом деле, существование алгоритма записи не гарантирует существования соответствующего ему алгоритма чтения. Например, пусть для массива A=(1,12,123) был выполнен оператор цикла:

```
for i:=Low(A) to High(A) do Write(A[i]);
```

т.е. были напечатаны десятичные представления чисел *без разделителей*: `112123`. При этом была утрачена существенная информация о сохраняемых данных, что препятствует их считыванию из полученного представления. Поэтому такой способ записи информации не может считаться форматом данных. Приведённый пример может показаться надуманным, но практически то же самое происходит, когда редактор связей записывает в образ памяти программы блоки кода и блоки данных, заботясь лишь о корректности перекрёстных ссылок. Таким образом, для сохранения возможности считывания в данных любого формата всегда должна быть как-то закодирована информация, необходимая для полного определения конструкторов типов всех имеющихся элементов данных.

Поэтому, кроме механизма описания структуры отдельного элемента данных, нужны механизмы для описания зависимостей между различными элементами. Наиболее важным случаем такой зависимости является указатель - элемент данных, в котором закодирована информация об адресе и типе другого элемента данных. Также следует учесть, что для правильной интерпретации одних элементов данных может потребоваться информация, содержащаяся в других элементах. Например, запись может содержать поля *Счётчик* и *Таблица*, где поле *Таблица* является массивом, число элементов которого

записано в поле *Счётчик*. Для описания подобных случаев используется механизм параметризации типов данных.

Сейчас невозможность использования языка FlexT для полного описания некоторых форматов объясняется не тем, что в языке не реализован какой-то необходимый конструктор типов, а тем, что пока отсутствует механизм извлечения информации, закодированной слишком сложным способом и необходимой для полного определения свойств некоторых типов (см. п. 2.8).

2. Основные элементы языка FlexT

Теперь рассмотрим более подробно особенности предлагаемого механизма определения типов.

2.1. Описание синтаксиса.

При описании синтаксиса конструкций языка будем пользоваться следующими обозначениями:

Обозначение	Описание
`<...>`	конструкция с именем `...`
`<Expr>`	выражение
`<Int Expr>`	целочисленное выражение
`<Addr Expr>`	адресное выражение
`'...'`	ключевые слова и символы берутся в кавычки
`[ ... ]`	необязательная часть
`{...|...}`	различные варианты продолжения
`{...|...|}`	различные варианты продолжения, могут отсутствовать
`{...,'...'}*`	последовательность из 0 или более одинаковых конструкций `...` с разделителем `'...'`
`{...,'...'}+`	последовательность из 1 или более одинаковых конструкций `...` с разделителем `'...'`
`nl`	переход на новую строку

Таблица 2. Обозначения, используемые при описании синтаксиса.

Также следует упомянуть, что в языке FlexT перевод строки является разделителем, если он происходит в месте возможного окончания конструкции и то, что вложенные определения типов можно брать в круглые скобки.

Язык FlexT является нечувствительным к размеру символов (прописные/строчные - case insensitive).

2.2. Общие особенности.

2.2.1. Составляющие переменного размера.

Язык поддерживает поля записей и элементы массивов переменного размера, т.е. размер составляющих составного типа может зависеть от данных конкретного экземпляра этого типа.

Пример из описания формата таблиц **Paradox [17]**, **[18]**:

```
TPxHeader struc pas
 .................................
  numFields: int;
 .................................
  FldNameTbl: array[@.numFields] of pchar;
  fieldNumbers: array[@.numFields] of int;
```

```
    sortOrderID: pchar
  ends
```

Для облегчения понимания примера здесь, забегая вперёд, необходимо пояснить, что `pchar` - это встроенный тип данных, обозначающий ASCIIZ строку. Т.е. в приведённом примере записано, что в состав записи `TPxHeader` входит поле `FldNameTbl`, являющееся массивом, число элементов которого определяется содержащимся в конкретных данных значением поля этой же записи `numFields`, т.е. размер поля `FldNameTbl` зависит от конкретных данных уже потому, что число элементов этого массива не фиксировано. Кроме того, сами элементы массива являются ASCIIZ строками, т.е. их размеры также определяются динамически. Также динамическими являются типы полей `fieldNumbers` и `sortOrderID`.

Если теперь разобрать согласно приведённой спецификации следующий фрагмент данных из файла таблицы Paradox:

```
0020:   04 00                                           ¦..... . П...¦z.¦

00E0:         23 00 4E 41 4D 45 00 47 43 4F 4C 00 54    ¦   #.NAME.GCOL.T¦
00F0:4C 49 4E 45 00 01 00 02 00 03 00 04 00 61 6E 63    ¦LINE.........anc¦
0100:79 72 72 00                                        ¦yrr.            ¦
```

то будет получен следующий результат:

```
0000:Hdr: TPxHeader = (
.................................
 numFields:4;
.................................
 FldNameTbl: (0:'#',1:'NAME',2:'GCOL',3:'TLINE');
 fieldNumbers: (0:1,1:2,2:3,3:4); sortOrderID:'ancyrr')
```

2.2.2. Параметры и свойства типов.

Тип данных может иметь ряд свойств, набор которых зависит от конструктора этого типа, например, размер и число элементов у массива, или номер случая у варианта. Каждый тип имеет свойство `Size` (`Размер`). Свойства могут задаваться конкретным значением при определении типа либо некоторым выражением, которое вычисляет значение данного свойства через значения и/или свойства вложенных элементов данных сложного типа, а также, может быть, через значения параметров типа.

Некоторые правила для вычисления свойства `Размер` могут автоматически добавляться компилятором, если они не указаны явно. Это касается, например, случая, когда известен размер всей записи и всех её полей, кроме последнего (см. п. 2.5.6).

Параметры в объявлении типа представляют ту информацию, которую необходимо указать дополнительно при использовании (вызове) данного типа. По умолчанию, параметр типа является целочисленным.

Пример объявления и использования типов с параметрами (фрагмент описания формата заголовка RPM **[19]**):

```
  THdrData(Kind,Cnt) case THdrValType(@:Kind) of
    CHAR: array[@:Cnt] of Char
```

```
    INT8: array[@:Cnt] of Byte
    INT16: array[@:Cnt] of ushort
    INT32: array[@:Cnt] of ulint
    INT64: array[@:Cnt] of ulint64
    BIN: raw[@:Cnt]
    STRING, STRING_ARRAY,
    RPM_I18NSTRING_TYPE: array[@:Cnt] of pchar
  endc
  PHdrData(Base,Kind,Cnt) ^THdrData(@:Kind,
    @:Cnt) NIL- near=ulint, REF=@:Base+@;
```

В этом примере записано, что тип `THdrData`, который является вариантом, принимает два параметра: `Kind` и `Cnt`. В параметре `Kind` передаётся значение для выбора типа массива, а в параметре `Cnt` - число элементов массива. Тип `PHdrData` является указателем на `THdrData` и принимает три параметра: `Kind` и `Cnt` для передачи в `THdrData` и `Base` для использования при вычислении адреса, на который ссылается указатель.

Если в объявлении типа не были заданы необходимые свойства, то для них автоматически создаются одноимённые параметры. Например, для типа

```
TArray1 array of byte
```

будут созданы два параметра: `Count` и `Size`, на которые можно впоследствии ссылаться по имени, чтобы полностью определить данный тип. Однако, более правильно, а в данном примере и более эффективно (так как для полного определения типа массива надо знать только одно из двух его свойств), определять параметры типа явно. Автоматическое создание параметров предназначено, в первую очередь, для безымянных типов, задаваемых определением на месте использования. Пример:

```
  TRec1(TblSz) struc
    int id
    array of int Tbl
  ends:[@.Tbl:Size=@:TblSz]
```

Здесь в блоке утверждений (п. 2.2.4.1) автоматически созданному параметру `Size` у типа поля `Tbl` присваивается значение, переданное в параметре `TblSz` содержащего типа `TRec1`.

2.2.3. Квалификаторы, используемые для ссылок на свойства и составляющие в выражениях.

Выражения в языке, в основном, вычисляются в контексте некоторой переменной, относящейся к определяемому типу. Ссылка на эту переменную в выражении обозначается `'@'`. Для ссылки на составляющие сложных типов могут использоваться различные *квалификаторы*, вид которых зависит от вида конструктора типа переменной. Пусть `V` - переменная, тогда

- `V[<Int Expr>]` - ссылка на элемент №`<Int Expr>`, если `V` является массивом;

- `V.<Имя поля>` - ссылка на поле `<Имя поля>`, если `V` является записью или перечислением термов;
- `V.<Значение>` - ссылка на случай, соответствующий значению `<Значение>`, если `V` является вариантом;
- `V.<Имя случая>` - ссылка на случай `<Имя случая>`, если `V` является проверкой.

Квалифицированные выражения также рассматриваются как переменные при обработке выражений. Более подробно квалификаторы описаны в п. 2.5 "Основные конструкторы типов.".

Для ссылки на параметр или свойство переменной используется квалификатор `<Переменная> ':' <Имя>`.

Выражение `'@'` ссылается на переменную текущего определяемого типа, т.е. того типа, в определении которого это выражение употребляется. Рассмотрим несколько примеров, поясняющих определение текущего типа:

1) `TByteArray(Cnt) array[@:Cnt] of Byte`
   Здесь выражение `@:Cnt` вычисляется в контексте типа `TByteArray`, который определяется при помощи конструктора вызова типа.

2)
```
TArrayPair(Cnt) struc
  array[@:Cnt] of Byte ID
  array[@:Cnt] of Str Name
ends
```
   Здесь оба выражения `@:Cnt` вычисляются в контексте типа `TArrayPair`, который определяется при помощи конструктора записи, т.е. в данном случае для выражения в квадратных скобках, задающего число элементов массива, текущим типом является содержащая этот массив запись, а не сам массив.

3) `TMatrix(W,H) array[@:H] of (array[@:W] of int)`
   Это определение вызовет ошибку компиляции, так как текущим определяемым типом для выражения `@:W` является тип `array[@:H] of ...`, который не имеет параметра `W`, а не тип `TMatrix`, у которого такой параметр есть. Чтобы исправить это определение надо заменить `@:W` на `@@:W`, т.е. использовать ссылку на родительский тип (см. далее).

Другие квалификаторы, которые могут употребляться в выражениях:

`<Переменная>'@'` - ссылка на родительский тип, т.е. на тип, в контексте которого данный тип *определяется*. Пример:
```
T1 struc
  int Cnt1
  int Cnt2
```

```
    array[@.Cnt1] of (array[@@.Cnt2] of word) Tbl
  ends
```

Так как выражения для параметра конструктора вычисляются в контексте вызова этого конструктора, значение свойства `Count` конструктора главного массива `Tbl` вычисляется в контексте родительского типа `T1`, а для вложенного массива - в контексте главного массива `Tbl`, поэтому требуется написать `@@.Cnt2`, чтобы сослаться на поле `Cnt2` родительского типа `T1`.

<Переменная>':' '@' - ссылка на содержащую переменную, т.е. на переменную, в которую *вложена* рассматриваемая переменная. Т.к. некоторый тип может использоваться в разных составных типах, для значения этого выражения известным является только адрес, но не тип, и, при использовании этой конструкции часто необходим оператор приведения к типу с проверкой: <Переменная> 'AS' <Тип>, предоставляющий недостающую информацию.

Приведём пример кода, использующего ссылку на содержащую переменную (см приложение 3.2.2):

```
  jmpofs2(OpcOfs) num-(2):displ=(HEX(@+@:OpcOfs,4))
  jmpofs4(OpcOfs) num-(4):displ=(HEX(@+@:OpcOfs,8))
  ..................................................
  TOp struc
    TOpKind K
    case @.K of
  ..................................
      jsr: jmpofs2(&@@-&@@:@)
      goto_w,jsr_w: jmpofs4(&@@-&@@:@)
  ..................................
    endc Arg
  ends

  TOpSeries array of TOp
```

Здесь типы `jmpofs*` представляют адрес для передачи управления, который задан смещением от начала текущей инструкции. Адрес начала инструкции (тип `TOp`) вычисляется выражением `&@@`. Для того, чтобы определить смещение от начала массива кодов (тип `TOpSeries`, *содержащий* тип `TOp`), используется выражение `&@@:@` для вычисления адреса этого массива.

<Переменная> ':' '#' или '#' (краткая запись для @:#)- номер вложенного элемента данных в содержащем его составном элементе данных, например, индекс элемента массива. Пример (из описания формата DBF, см. Рис. 3):

```
  TFieldData array[Hdr.Fields[#].Len]of Char
  TFieldsData array of TFieldData
```

В этом фрагменте записано, что число элементов массива TFieldData, который является представлением значения поля записи, определяется выражением Hdr.Fields[#].Len, т.е. для его вычисления надо взять значение поля Len из элемента массива Hdr.Fields с тем же номером, что и номер данного элемента TFieldData в массиве TFieldsData.

2.2.4. Дополнительные блоки определений типов.

За любым определением типа, кроме переименования, могут следовать несколько блоков с дополнительной информацией, перед которыми стоит знак ':' - признак продолжения определения (этот признак должен стоять на той же строке, иначе определение будет считаться законченным). Рассмотрим виды дополнительных блоков.

2.2.4.1. Блок утверждений.

Признаком блока утверждений является стоящая за ':' скобка '['. Этот блок содержит в скобках '[',']' и через ',' ряд утверждений о значениях свойств определяемого типа и параметров типов входящих в него элементов. Формат утверждения:

<Элемент данных> ':' <Параметр или свойство> '=' <Expr>

где <Элементом данных> может быть:

- '@' - переменная текущего определяемого типа;
- '@.'<Имя поля> - ссылка на тип поля записи, если @ - запись;
- '@[]' - ссылка на тип элемента массива, если @ - массив;
- '@^' - ссылка на базовый тип указателя, если @ - указатель.

Пример (продолжение примера с форматом RPM из п. 2.2.2):

```
TSHeaderNDXItem(Base) struc
  TSignTagType tag
  THdrValType type
  PHdrData(Base=@:Base,Kind=@.type) offset
  ulint count
ends:[@.offset:Cnt=@.count]
```

Здесь в записи TSHeaderNDXItem происходит обращение к типу PHdrData, причём, параметры Base и Kind передаются по имени непосредственно при вызове, а параметр Cnt задаётся в блоке утверждений, т.к. к моменту объявления поля offset, поле count ещё не определено.

Другой пример (продолжение примера с DBF из п. 2.2.3):

```
TFieldsData array of TFieldData :[@:Size=Hdr.H.RecLen-1]
```

показывает, что для использования блока утверждений могут быть и более серьёзные основания, чем отсутствие поддержки в языке использования имён до их объявления (это ограничение впоследствии планируется снять). Здесь записано, что размер массива TFieldsData с элементами переменной длины указан в поле Hdr.H.RecLen. Блок утверждений используется здесь потому, что синтаксис конструктора массива предусматривает указание числа элементов, а не размера массива.

2.2.4.2. Блок условий корректности.

Признаком данного блока является ключевое слово `assert`. Синтаксис:

```
'assert' '[' {<Int Expr>, ','}+ ']'
```

Каждое из выражений `<Int Expr>` интерпретируется, как логическое выражение. Все эти выражения должны успешно вычисляться и быть истинными, чтобы элемент данных определяемого типа считался корректным. Условия корректности используются для выбора варианта содержимого в типе "Проверка" (TRY, п. 2.5.8). В дальнейшем предполагается расширить область применения этих условий.

В выражениях можно проверить корректность переменной `X` при помощи ссылки на её свойство `assert`: "`X:assert`".

2.2.4.3. Блок отображения.

Информация, определяемая в данном блоке, не влияет на способ интерпретации данных, а затрагивает лишь способ их отображения. Признаком данного блока является ключевое слово `displ`. Синтаксис:

```
'displ' '=' '(' {<Команда отображения>, ','}+ ')'
```

В настоящее время определены следующие команды отображения (все используемые в них выражения, если не оговорено иначе, оцениваются в контексте определяемого типа):

- **Вывод строки** . Синтаксис: `<строковая константа>`.
- **Перевод строки**. Синтаксис: `'NL' [<Int Expr>]`. Если в этой команде присутствует выражение, то оно задаёт величину, на которую смещается левая граница вывода текста, иначе эта граница не смещается. Изменение левой границы текста действует только в рамках текущего типа.
- **Вывод целого числа в десятичном представлении**. Синтаксис: `'INT' '(' <Int Expr> ')'`.
- **Вывод целого числа в шестнадцатеричном представлении**. Синтаксис: `'HEX' '(' <Int Expr> [',' <Число знаков>] ')'`. Значение выражения `<Int Expr>` отображается в шестнадцатеричном представлении. Если задано `<Число знаков>`, то при необходимости результат дополняется спереди нулями до этого числа знаков. В шестнадцатеричное представление значения не включается префикс `'0x'` - только сами шестнадцатеричные цифры.
- **Отображение массива**. Как следует из названия, эта команда применима только к массивам. Синтаксис:

  ```
  'SHOWARRAY' '(' <Addr Expr> ','
    '(' {<Команда отображения>, ','}+ ')' ')'
  ```

  Выражение `<Addr Expr>` должно возвращать ссылку на массив, а в следующем параметре команды в скобках задаётся последовательность команд отображения, которая выполняется для каждого элемента массива, при этом все выражения этой последовательности оцениваются в контексте отображаемого элемента массива.

- **Отображение элемента данных**. Если команда отображения не является одной из вышеупомянутых, то она должна являться адресным выражением, которое задаёт элемент данных для отображения. Элемент данных отображается согласно его блоку отображения, если этот блок задан, и если этот элемент данных не является определяемым (`'@'`), иначе он отображается согласно правилам конструктора типа. Синтаксис: `<Addr Expr>`. Если первая лексема выражения совпадает с одной из команд, то можно просто взять выражение в круглые скобки, чтобы предотвратить конфликт имён (первой лексемой станет скобка).

Так как наиболее сложной из вышеупомянутых команд является команда отображения массива, приведём примеры её использования:

1. Представление чисел в формате BCD (Binary-Coded Decimal) из описания формата таблиц Clarion **[20]**.

```
TBCD array of byte:displ=(ShowArray(@,(@)))
```

   Данное определение приводит к тому, что все байты массива отображаются слитно, без запятых и номеров элементов массива.

2. Последовательность команд Java-машины (приложение 3.2.2):

```
TOpSeries array of TOp: displ=('(',
  ShowArray(@,(NL,HEX(&@-&@:@,4),': ',@)),NL,')')
```

   В этом определении записано, что для отображения последовательности команд сначала выводится открывающая круглая скобка; затем для каждой команды: перевод строки, 4 знака шестнадцатеричного представление смещения от начала массива, строка `': '`, представление команды; и, наконец, после вывода всех команд - перевод строки и закрывающая круглая скобка.

2.2.4.4. Блок именования.

Информация, определяемая в данном блоке, может использоваться для автоматической генерации имён переменных, которые не были определены явно, а были найдены по ссылкам из указателей. В этом случае при отсутствии такой информации переменной присваивается имя в соответствии с тем квалификатором, который первым ссылается на неё в процессе разбора (т.е. обнаружение этой ссылки приводит к добавлению данной переменной). Ссылочный квалификатор может быть достаточно длинным, даже после применения сжатия повторений (см. п. 3.8.1), кроме того, он не содержит информации о назначении переменной, поэтому иногда бывает удобным дать имя переменной в соответствии с её содержимым.

Признаком данного блока является ключевое слово `autoname`. Синтаксис:

```
'autoname' '=' '(' {<Команда отображения>, ','}+ ')'
```

совпадает с синтаксисом блока отображения со следующими исключениями:

- команда NL игнорируется;
- для получения текста элемента данных используется не блок отображения, а блок именования, если таковой имеется, иначе - процедура приведения

значения к строке, а не процедура отображения (см. описания методов в п. 3.3.1).

Пример использования блока именования (заголовок RPM **[19]**):

```
TSHeaderNDXItem(Base) struc
  TSignTagType tag
 ..................
ends:[@.offset:Cnt=@.count]:autoname=(@.tag)
```

2.2.5. Вызовы типов.

Вызов типа с подстановкой фактических параметров на место формальных рассматривается как специальный конструктор типа.

При вызове типа параметры могут указываться либо позиционно, либо по имени (как при вызове методов интерфейсов COM).

Пример из файла FB09.RFI (Описание формата отладочной информации фирмы Borland **[21]** для программы PE Explorer):

```
TBaseSrcFilesTbl(Cnt,Base) array[@:Cnt] of
   PBaseSrcFileInfo(@:Base)
```

Здесь параметр в тип PBaseSrcFileInfo передаётся позиционно.

Другой пример из того же файла:

```
PDebugSubSectionData(Kind,Sz) ^
TDebugSubSectionData(@:Kind,@:Sz) - lfaFB09Base FIXUP
  OFF;
TDebugSubSection(Sz) struc
  TDebugSubSectionType subsection
  Word iMod
  PDebugSubSectionData(Kind=@.subsection) lfo
  ulong cb
  raw[] rest
ends:[@.lfo:Sz=@.cb]
```

Здесь при описании поля `lfo` записи `TDebugSubSection` используется передача фактических параметров по имени, причём параметр `Kind` передаётся непосредственно в вызове типа `PDebugSubSectionData`, а значение параметра `Sz` задаётся в блоке утверждений.

Надо различать вызов типа и *переименование типа*. В следующем примере:

```
type
  WordRename word
  WordCall   word()
```

первое определение просто сопоставляет типу `word` новое имя `WordRename`. При этом не создаётся новый тип, и поэтому, например, типу `WordRename` нельзя сопоставить свой способ отображения (блок displ). Второе определение приводит к созданию нового типа - вызова типа `word`. На это указывает наличие круглых скобок - пустого списка фактических параметров. При этом новый тип уже может, например, иметь свой способ отображения:

```
WordCall word():displ=(INT(2*@))
```

2.3. Выражения.

В различных частях программы для описания зависимостей между элементами данных могут использоваться выражения. При наличии в выражении ссылок на переменные или динамические константы будем называть их *динамическими,* так как значения таких выражений могут зависеть от конкретных данных. Соответственно, при отсутствии в выражении динамических составляющих, будем называть их *статическими*.

По виду возвращаемого результата будем различать *целочисленные* и *адресные* выражения. Результат вычисления целочисленного выражения, как это и следует из названия, является целым числом. В настоящее время в языке поддерживается 32-разрядная целочисленная арифметика. Впоследствии планируется расширить возможности интерпретатора, чтобы включить в него поддержку целых чисел произвольного размера (код для работы с такими числами автором уже написан). Адресные выражения возвращают ссылку на элемент данных, т.е. на переменную или её составную часть. Вид возвращаемого результата определяется местом употребления выражения, причём, одно и то же выражение может интерпретироваться как адресное и как целочисленное в зависимости от контекста. При описании конструкций языка будем писать `<Int Expr>` для обозначения целочисленного выражения и `<Addr Expr>` для обозначения адресного выражения.

2.3.1. Поддерживаемые операции.

В текущей версии FlexT в выражениях могут использоваться следующие операции (перечислены в порядке увеличения приоритета):

- `exc`
- `when`
- `as`
- `=, <, <=, <>, >=, >`
- `MIN, MAX`
- `-, +, or, xor`
- `shl, shr`
- `*, DIV, MOD, and`
- `- (унарный), &, not, IsFixup`

По типам операции можно разделить на:

- арифметические: `-, +, *, DIV, MOD, - (унарный),MIN,MAX`; символ `'/'` зарезервирован для операции деления плавающих и/или рациональных чисел;
- побитовые/логические: `or, and, not, xor`;
- арифметические сдвиги: `shl,shr`;
- операции сравнения: `=,<,<=,<>,>=,>`;
- специальные: `&, as, IsFixup, exc, when`.

2.3.2. Семантика операций.

Операции сравнения возвращают целое число: 1, если условие истинно и 0, если оно ложно. Для того, чтобы к этим значениям можно было применять побитовые операции в качестве логических, в тех частях кода программы, где значение выражения проверяется на истинность, учитывается лишь значение младшего бита.

В остальном семантика всех операций, кроме специальных, полностью совпадает с семантикой соответствующих операций языка Pascal, поэтому здесь мы подробнее рассмотрим только операции последней группы:

`&` – операция взятия адреса. Является унарной, аргумент - адресное выражение. Использование: `&<Addr Expr>`. Возвращает целое число - смещение результата вычисления адресного выражения в текущем адресном пространстве.

`as` - операция приведения к типу. Использование: `<Addr Expr> as <Имя типа>`. При вычислении эта операция *проверяет*, что `<Addr Expr>` действительно относится к указанному типу и, если это не так, то происходит ошибка вычисления. Эта операция нужна, например, для указания типа ссылки на содержащую переменную (`<Переменная>:@`, см. 2.2.3).

`IsFixup` - проверка указателя на то, что его адрес упомянут в таблице перемещений. Использование: `IsFixup <Addr Expr>`. Такая поверка оказывается полезной, например, для определения размера таблицы виртуальных методов.

`exc` - обработка ошибок вычисления выражений. Использование: `<Expr 1> exc <Expr 2>`. Если в процессе вычисления `<Expr 1>` ошибок не происходит, то значением всего выражения является значение `<Expr 1>`, иначе - значение `<Expr 2>`. При этом, если при вычислении `<Expr 2>` также произойдёт ошибка, то и всё выражение вычисляется с ошибкой. Фактически, существует два варианта данной операции: `<Int Expr> exc <Int Expr>` для целочисленных выражений и `<Addr Expr> exc <Addr Expr>` для адресных выражений.

`when` - задаёт условие, при выполнении которого возможно вычисление выражения. Использование: `<Expr> when <Int Expr>`. Если выражение `<Int Expr>` является ложным или при его вычислении произошла ошибка, то происходит ошибка вычисления всего выражения, иначе значением всего выражения является значение `<Expr>`. Как и операция `exc`, операция `when` имеет два варианта: `<Int Expr> when <Int Expr>` для целочисленных выражений и `<Addr Expr> when <Int Expr>` для адресных выражений.

2.3.3. Обработка ошибок в выражениях.

При вычислении выражений учитывается возможность возникновения ошибок и, кроме "штатной" последовательности вычислений, могут возникать

и "нештатные", например, деление на ноль для операции деления или обращение к невыбранному случаю варианта для квалификатора ссылки на случай варианта. Эти ошибки могут перехватываться операцией `exc`, что позволяет получать выражения, вычисление которых не вызывает ошибок. В отличие от механизма обработки исключительных ситуаций, используемого в современных системах программирования, например, в Delphi, здесь никак не передаётся информация о причине ошибки, т.к. необходимость в этом не была выявлена. То, что более сложный механизм не потребовался, объясняется тем, что обработчик определённого вида ошибок можно разместить в выражении непосредственно после возможного источника этих ошибок.

Комбинации операции `exc` и `when` могут использоваться для записи условных выражений. Пример (из описания формата BMP **[16]**, **[22]**):

```
TOS2Pal(BC) array[((1 shl @:BC)when(@:BC<=8))exc 0] of
    TRGBTriple
```

Здесь записано, что палитра присутствует в файле и при этом количество её элементов определяется числом бит на пиксел (параметр `BC`), только если это число меньше или равно `8`.

2.4. Структура программы.

В текущей реализации интерпретатора FlexT программа компилируется *после* загрузки разбираемых данных. Это создаёт некоторые сложности при попытке использования спецификаций для других целей, что связано с необходимостью существенного изменения методов интерпретации. С другой стороны, при таком подходе упрощается написание спецификаций за счёт возможности использования динамических выражений, например, в директивах условной компиляции (см. п. 2.4.4), а именно максимальная простота написания спецификаций и является целью создания языка спецификации.

2.4.1. Блоки определений.

Программа на FlexT состоит из нескольких видов блоков определений, в которых определения разделяются переводом строки. Основные блоки определений:

1) Блок констант.

```
'CONST' nl
  {<Имя константы> '=' <Int Expr> ';', nl}*
```

В выражениях для вычисления значений констант могут использоваться переменные из блока данных, поэтому некоторые из констант могут быть динамическими, т.е. зависящими от конкретных данных.

2) Блок типов.

```
'TYPE' ['BIT'] nl
  {<Имя типа> <Определение типа>, nl}*
```

содержит определения типов. Признак `BIT` указывает, что все определяемые в этом блоке типы будут иметь битовое, а не байтовое размещение (см. п. 3.6).

3) Блок данных.

```
'DATA' [<Блок>] nl
  {<Int Expr> [';'] <Определение типа> <Имя>, nl}*
```

содержит определения переменных описываемого формата. `<Int Expr>` задаёт адрес переменной, т.е. её адрес в текущем адресном пространстве, если имя блока не указано, или смещение в блоке памяти `<Блок>`. Так же, как и при определении констант, выражение может содержать обращения к другим переменным, так что адрес переменной может определяться динамически. `<Определение типа>` должно задавать тип без свободных параметров, т.е. должны быть определены все свойства базового конструктора этого типа и типов его составляющих, если таковые имеются. Будем называть такой тип *полностью определённым*. После выражения для адреса можно ставить признак его окончания `';'`. Это необходимо делать лишь в том случае, если начало `<Определения типа>` может быть воспринято, как продолжение выражения.

4) Блок кода.

```
'CODE' ['(' <Имя типа кода> ')'] [<Блок>] nl
  {<Int Expr> [';'] <Имя>, nl}*
```

содержит адреса и названия точек входа в блоке памяти `<Блок>` или в текущем адресном пространстве, если имя блока не указано. При указании имени типа кода память, начиная с указанных точек входа, разбирается в соответствии с этим типом данных. Тип кода должен быть задан конструктором `CODES` и быть полностью определённым, при этом он описывает кодирование машинных инструкций и признаки команд, завершающих код (таких как `RET` или `JMP`). Если тип кода не задан, то используется код процессоров Intel 80x86.

Специальное имя `'-'` означает, что указанный адрес является не точкой входа, а точкой окончания кода. Эта возможность оказывается полезной в тех случаях, когда дизассемблер не может самостоятельно правильно определить окончание последовательности команд. Это может произойти, например, в следующем случае:

```
CALL Terminate
<Данные>
```

когда процедура `Terminate` всегда вызывает функцию API для завершения работы программы. При этом встроенный дизассемблер не располагает информацией о том, что из процедуры `Terminate` не бывает возврата, и продолжает разбирать `<Данные>` как код. Заметим, что `<Данные>` могут непосредственно следовать за вызовом лишь в том случае, если компилятор учитывал особенности процедуры `Terminate`.

5) Блок именования.

```
'AUTONAME' nl
  {<Имя типа кода> <Квалификатор> [<Идентификатор>],
   nl}*
```

Этот блок сохраняется для совместимости с предшествующими версиями интерпретатора FlexT и описан здесь только потому, что он используется в некоторых примерах спецификаций, приведённых в приложениях (см., например, приложение 4.1).

По своей семантике он аналогичен включению в определение типа <Имя типа кода> одноимённого блока (см. п. 2.2.4.4) следующего содержания:

- **:autoname**=(@<Квалификатор>), если <Идентификатор> отсутствует;
- **:autoname**=('<Идентификатор>',@<Квалификатор>), если <Идентификатор> не начинается с символа '_';
- **:autoname**=(@<Квалификатор>,'<Идентификатор>'), если <Идентификатор> начинается с символа '_';

при этом, если <Квалификатор> начинается с буквы, то к нему спереди добавляется '.', т.е. если, например, <Квалификатор> состоит из одного идентификатора, то этот идентификатор будет интерпретироваться как имя поля.

2.4.2. Директивы компилятора.

Кроме блоков определений, существует ряд директив компилятора, которые обрабатываются вне контекста блоков:

1) Директива включения содержимого другого файла в текст данного файла

```
'INCLUDE' <Имя файла> nl
```

При этом любое отдельное определение (переменной, типа, и т.д.) должно содержаться целиком в одном файле. Т.е. эту директиву нельзя использовать, например, для включения частей определений.

2) Директива глобального утверждения

```
'ASSERT' <Int Expr> ';' nl
```

Задаёт логическое утверждение, которое должно выполняться, если файл действительно относится к рассматриваемому формату. Эта директива используется для автоматического определения формата файла: если в процессе чтения описания формата такое выражение оказывается ложным на разбираемых данных, то система делает вывод, что эти данные не относятся к рассматриваемому формату. В этом случае система может перейти к проверке следующего описания формата, которое может соответствовать расширению файла (см. п. 3.8).

3) Директива задания установок компилятора:

```
SET <Имя свойства> <Значение свойства> nl
```

Пока <Имя свойства> может принимать только одно значение - BYTEORDER. Свойство BYTEORDER задаёт *порядок байтов*, в котором записаны целочисленные данные (см. п. 3.7). Для BYTEORDER <Значение свойства> имеет вид:

```
'REV'|'NORM' ['=' <Int Expr>';']
```

Так как оценки автора в данном вопросе являются смещёнными в пользу порядка байтов процессоров Intel, значение NORM задаёт порядок байтов этих процессоров, известный также как LSB (Least Significant Byte first), который мы далее будем называть нормальным, а значение REV - обратный этому порядок байтов, известный также как MSB (Most Significant Byte first), который используется, например, процессорами Motorola (компьютеры Apple) или SPARC и который мы далее будем называть обратным. Если за названием порядка байтов следует знак равенства, то далее должно быть задано логическое выражение, при истинности которого выбирается названный порядок байт, а иначе - обратный ему. Это логическое выражение является динамическим, поэтому порядок байтов части данных может определяться значениями других данных, например, значением поля из заголовка файла (выдержка из описания формата ELF **[30]**):

```
set byteorder rev=(Ident.ei_encoding =
  TElfEncoding.ELFDATA2MSB);
```

Установленный порядок байтов действует на все *определяемые далее* целочисленные типы данных, а также на массивы и записи с битовым размещением (см. п. 3.6). Однако, этот порядок не действует на уже определённые типы и, в том числе, на предопределённые целочисленные типы, такие как WORD или INT. Все предопределённые целочисленные типы имеют нормальный порядок байтов. Это означает, что если, например, требуется определить целочисленный тип данных, занимающий два байта, но с обратным порядком размещения, то нельзя воспользоваться предопределённым типом INT, а необходимо объявить новый целочисленный тип при помощи конструктора num. Пример (из описания формата файла класса Java-машины **[23]**):

```
set byteorder rev
type
  u1 num +(1)
  u2 num +(2)
  u4 num +(4)
```

Область действия директивы, задающей порядок байтов, ограничивается следующей подобной директивой.

4) Директива описания формата:

```
DESCR '('<Список команд печати>')'
```

задаёт текст, который должен быть включён в листинг - результат разбора файла. <Список команд печати> имеет тот же формат, что и в блоке описания способа отображения типа (блок DISPL, см. п. 2.2.4.3). Эта информация никак не влияет на сам процесс разбора файла и может рассматриваться как специальный вид комментария.

2.4.3. Комментарии.

В программе на FlexT могут использоваться комментарии следующих трёх видов:

1) От '/*' до '*/';
2) От '//' до конца строки;
3) От '%' до конца строки.

Комментарии вида /* */ могут быть вложенными.

2.4.4. Условная компиляция.

За комментариями, начинающимися с символа '%', могут следовать директивы условной компиляции, начинающиеся с символа '$'. Эти директивы имеют следующий вид:

```
  '%$IF' <Int Expr> ';' nl
    <Текст программы>
 ['%$ELSIF' <Int Expr> ';' nl
    <Текст программы>]*
 ['%$ELSE' nl
    <Текст программы>]
  '%$END'
```

Выражение <Int Expr> задаёт логическое условие, при выполнении которого компилируется <Текст программы>, непосредственно следующий за %$IF или %$ELSIF, иначе проверяется следующее условие %$ELSIF или компилируется <Текст программы>, непосредственно следующий за %$ELSE, если у директивы имеются такие части. Как и в других конструкциях FlexT, выражения для условий могут быть динамическими. Пример (выдержка из описания формата EXE-файлов: **[16]**-NE, **[24]**,**[25]**-PE, **[26]**-LX):

```
data
  NewHdrOfs word wNewSignature //.EXE file signature
                               // ('NE','PE','LE')
%$IF wNewSignature=sgnLE;
  include LE.RFI
%$ELSIF wNewSignature=sgnNE;
  include NE.RFI
%$ELSIF wNewSignature=sgnPE;
  data
    NewHdrOfs+2 word PEz
    assert PEz=0;
  type
```

```
    TIMAGE_HEADERS forward
  data
    NewHdrOfs+4 TIMAGE_HEADERS COFFHdr
    include COFF.RFI
%$END sgnLE
```

Последовательность '%$'<Идентификатор> вызывает ошибку компиляции, если <Идентификатор> не входит в число допустимых. Также вызывает ошибку отсутствие закрывающей директивы %$END для директивы %$IF.

2.5. Основные конструкторы типов.

2.5.1. Целые числа.

Целое число является базовым типом данных, т.к. уметь читать целые числа необходимо, чтобы определять значения параметров всех остальных типов. В языке существует конструктор типов целых чисел:

```
'NUM' {'+'|'-'|} '(' <Expr> ')'      ,
```

который позволяет задать размер типа и наличие у него знака. Также имеется ряд предопределённых типов (см. п. 2.6): byte, int, long и т.д., которые просто являются предопределёнными вызовами конструктора NUM, например, тип int можно определить, как:

```
int NUM - (2)
```

На конструктор целого числа влияют текущий порядок байтов (см. SET в п. 2.4.2) и единица размещения данных (бит или байт, см. TYPE в п. 2.4.1), так же, как на результат процедуры line влияет выбранное в контексте устройства перо. Если установлено битовое размещение данных, то заданный размер целого числа измеряется в битах, иначе - в байтах.

2.5.2. Пустой тип void.

Этот тип удобно использовать в процессе изучения данных, формат которых до конца не известен. Например, можно указать расположение в памяти некоторой переменной, тип которой ещё не установлен:

```
data
  0x00456780 void SomeData
```

после чего все ссылки на эту переменную из кода будут помечены указанным именем переменной:

```
0042F6EC: XOR ECX,ECX
0042F6EE: MOV CL,BYTE PTR [EAX]
0042F6F0: MOV ECX,DWORD PTR [4*ECX+SomeData{00456780}]
0042F6F7: CALL ECX,NEAR
0042F6F9: RET NEAR
```

что может помочь установить её тип. Так, по приведённому фрагменту можно понять, что, на самом деле, эта переменная имеет тип **array of** CodePtr.

Другое возможное применение - базовый тип указателя на данные неизвестной структуры. Рассмотрим следующий пример:

```
type
  PSomeData = ^void
  TNameInfo = struc
    pchar Name
    PSomeData Info
  ends
data
  0x00456780 array[10]of TNameInfo NameInfoTbl
```

Здесь предполагается, что в ходе изучения блока данных некоторой программы была обнаружена повторяющаяся 10 раз структура: (<ASCIIZ строка>, <указатель в блок данных>). Указатель легко обнаруживается в данных при наличии таблицы перемещений (см. п. 3.11). Чтобы понять назначение обнаруженных указателей, можно написать вышеприведённый код, после чего все адреса в блоке данных, на которые ссылаются такие указатели, будут помечены:

```
  0045AFAB:F1 0A 03 4C 54 52 80 46 5D 01 0F 00 ¦ë..LTRAF]...¦
0045AFB7: NameInfoTbl[6]^.Info: void =
  0045AFB7:FE 0A 03 53 54 52 80 46 4D 01 0F 00 ¦¦..STRAFM...¦
```

что позволяет изучить данные, расположенные по этим адресам.

2.5.3. Символьные типы `char` и `wchar`.

Тип `char` служит для представления отдельных 8-битных символов в текущей кодировке. Тип `wchar` служит для представления отдельных 16-битных символов в кодировке UniCode **[28]**.

В существующей версии языка текущая кодировка задаётся глобально (см. п. 4.1, /I), в дальнейшем предполагается сделать этот механизм более гибким. Также предполагается повысить уровень абстракции при объявлении символьных типов за счёт сопоставления произвольному базовому типу выражения, возвращающего соответствующий значению этого типа код символа в текущей кодировке или в кодировке UniCode. Это может быть сделано по аналогии со свойством REF у указателя либо через механизм интерфейсов (см. п. 2.7). Такое расширение механизма определения символьных типов позволит обрабатывать и многобайтовые кодировки символов азиатских языков (multibyte character sets), которые сейчас не поддерживаются интерпретатором. Другое ограничение, которое может быть преодолено таким образом, - это то, что сейчас в языке FlexT не поддерживаются символьные типы с битовым размещением.

Значения обоих символьных типов при упоминании в выражениях могут интерпретироваться как целое число без знака соответствующего размера. При использовании обоих типов в качестве типа элемента массива, такой массив может интерпретироваться, как строка (см. п. 2.5.9).

2.5.4. Перечислимые типы.

Эти типы позволяют поставить в соответствие некоторым значениям базового типа имена, значения выражений или термы. Конструктор:

```
'ENUM' [<Имя типа>] {
    '(' {<Имя>[{'='<Int Expr>|'^'<Int Expr>}] , ','}*
')'
  |['=']<Expr> ';'
  | 'FIELDS' '(' {<Определение поля>, ','}+ ')' 'OF'
      '(' {<Имя> ['(' {<Имя поля>, ','}+ ')']
          ['=' <База терма>] , ','}* ')'
}
```

имеет три варианта:

1) Собственно перечисление.

   В круглых скобках могут перечисляться пары `<Имя>=<Int Expr>` ; если `'='<Int Expr>` опущено, то имени сопоставляется предыдущее значение плюс 1. Начальное значение равно 0.

2) Отображение выражением.

   Если круглых скобок нет или стоит знак `'='`, то указывается выражение для сопоставления значению строки.

3) Перечисление термов.

   Ключевое слово `'FIELDS'` указывает на перечисление термов, которое позволяет интерпретировать значения типа не просто как константы, а как терм с параметрами - битовыми полями. Этот тип удобно использовать для описания кодирования машинных команд в стиле **[29]**.

Во всех трёх вариантах:

- базовый тип `<Имя типа>` не должен иметь параметров;
- если `<Имя типа>` опущено, то базовым является тип `Byte`;
- если значение базового типа не удаётся отобразить согласно имеющемуся определению (значению не было сопоставлено имя в перечислении, терм - в перечислении термов, или вычисление выражения для отображения выражением привело к ошибке), то это значение отображается согласно определению базового типа. Это позволяет организовать наследование перечислений, так как базовым типом для перечисления может быть другое перечисление.

Базовый тип собственно перечисления и перечисления термов должен приводиться к целому числу. Задача первых двух вариантов перечисления сводится к управлению отображением, а перечисление термов позволяет также обращаться к полям выбранного терма в выражениях.

Рассмотрим особенности каждого из перечислимых типов.

2.5.4.1. Собственно перечисление

Используется для сопоставления имён констант некоторым значениям базового типа. Пример (из описания формата ELF **[30]**):

```
TElfClass enum byte (
```

```
    ELFCLASSNONE=0, //Invalid class
    ELFCLASS32=1, //32-bit objects
    ELFCLASS64=2 //64-bit objects
  )
```

В этом примере значениям 0, 1, 2 базового типа `byte` сопоставлены имена констант, остальные значения, если они встретятся, будут отображаться в соответствии с определением базового типа. Тип `TElfClass` можно было бы определить и без упоминания значений констант:

```
  TElfClass enum byte (
    ELFCLASSNONE, //Invalid class
    ELFCLASS32, //32-bit objects
    ELFCLASS64 //64-bit objects
  )
```

так как по умолчанию нумерация констант начинается с 0, а значение следующей константы, если оно не указано явно, равно значению предыдущей плюс 1.

Имена констант, определённых в перечислении термов, не являются глобальными, поэтому, чтобы сослаться на них в произвольном выражении, надо использовать синтаксис `<Имя типа>.<Имя константы>`, например, `TElfClass.ELFCLASS32`. Исключение составляют случаи, когда известно, что происходит чтение константы данного перечислимого типа, например, при чтении констант для выбора случаев варианта, если в качестве селектора варианта используется выражение, возвращающее значение этого перечислимого типа - в этом случае можно использовать имя константы без указания имени типа.

2.5.4.2. Отображение выражением.

Этот вариант конструктора можно было бы заменить на

```
  :displ=(<Expr> exc @)
```

в определении типа `<Имя типа>`. Этот вариант сохраняется для совместимости с имеющимися спецификациями (он был реализован раньше, чем блок определений типов **`displ`**), а также потому, что пока не реализована возможность определения типа по частям, зато реализовано переопределение перечислимого типа в варианте отображения выражением. Переопределение означает, что можно, например, написать:

```
  TEnum enum TEnum = NameTbl[@].name^;
```

При этом новое определение *заменяет* старое определение типа, поэтому это старое определение не должно быть переименованием типа (иначе оно повлияет и на базовый тип переименования). Например, если `TEnum` было определено, как

```
  TEnum word    ,
```

то предыдущий пример вызовет ошибку. Чтобы избавиться от неё, надо преобразовать определение типа в вызов типа:

```
  TEnum word()
```

Пример, использующий возможность переопределения перечислимого типа (фрагмент приложения 3.2.1):

```
type
 ......................
  TClassNDX u2()
 ......................
  TClassFile struc
   ......................
    array[ ... ]of cp_info ... C_pool
   ......................
    TClassNDX this_class
    TClassNDX super_class
   ......................
  ends
data
  0004 TClassFile Hdr
type
 ......................
  TClassNDX enum TClassNDX Hdr.C_pool[@-1].info.C_Class;
```

Здесь для отображения значения типа `TClassNDX` используются данные переменной `Hdr`, относящейся к типу `TClassFile`, а этот тип, в свою очередь, имеет составляющие, относящиеся к типу `TClassNDX`. Использование переопределения позволяет разорвать циклическую зависимость.

Надо признать, что подобные приёмы не упрощают понимание и использование языка, поэтому впоследствии предполагается реализовать возможность использования имён до их определения, как это, например, сделано в языке JavaScript **[31]**.

2.5.4.3. Перечисление термов.

Перечисление термов удобно использовать в тех случаях, когда способ интерпретации некоторых данных зависит не от значения другого элемента данных - этот случай описывается при помощи конструктора типов `CASE`, а от вида нескольких первых байтов *самих этих данных*. Более общий случай выбора интерпретации по виду разбираемых данных может быть представлен при помощи блока `TRY` в сочетании с условиями корректности типов - вариантов интерпретации (см. п. 2.5.8, пример с форматом DVI). Однако, такой подход оказывается неудобным и не слишком эффективным, когда существует много возможных вариантов интерпретации данных, как это, например, происходит при описании кодирования машинных инструкций: при попытке описать таким образом кодирование инструкций Z-80 **[32]** - далеко не самой сложной системы команд, автору хватило терпения только на первый десяток инструкций. Что касается эффективности, то, если при выборе подходящего варианта в универсальном блоке `TRY` приходится последовательно

рассматривать все возможные альтернативы, пока не встретится подходящая, то для перечисления термов существуют более быстрые методы, использующие таблицу или дерево решений (см. п. 3.12).

Синтаксис <Определения поля> в блоке FIELDS:

```
<Имя поля> [':'] <Имя типа> ['@'<Int Expr>]
      {'.' <Int Expr> | ':' <Int Expr> | }
```

Тип поля <Имя типа> должен иметь битовое размещение, он не должен иметь параметров, а его размер должен быть фиксирован. <Int Expr> после символа '@' определяет смещение поля (номер первого бита поля в данных перечисления термов), если это смещение не задано явно, то оно принимается равным 0. <Int Expr> после символа '.' определяет размер поля в битах, а <Int Expr> после символа ':' - номер последнего бита поля. Последние два значения задают размер поля в битах. Эта информация является избыточной и используется только для проверки, а также в качестве комментария. Если явно заданный размер не равен размеру типа поля, то происходит ошибка. Пример объявления полей (фрагмент приложения 3.3.1):

```
TBit3 num+(3)
TRN enum TBit3 (B,C,D,E,H,L,M,A)
.................................
TByteOpCode enum TBit8 fields (
  Nd: TRN @3.3,
  Ns: TRN @0.3,
.............................
) of (
```

Здесь тип TRN определяет соответствие номеров именам регистров, а поля Nd и Ns задают, соответственно, регистр-приёмник и регистр-источник в машинных командах.

<База терма> является строковым представлением числа, которое, кроме того, может содержать символы маски '_' вместо 0, если это представление не является десятичным. Если в <Базе терма> присутствуют символы маски, то сообщается об ошибке, если эта маска не соответствует битам, занятым полями терма. Если маска отсутствует, то она строится автоматически, согласно полям терма. Таким образом, использование символом маски также предоставляет избыточную проверочную информацию. При определении выбранного терма происходит поиск первого в порядке объявления терма, у которого значение <Базы терма> совпадает с текущим значением, без учёта битов маски. На самом деле, поиск выполняется более эффективно (см. п. 3.12), но при этом получается тот же результат, что и при линейном поиске. Пример объявления термов (продолжение предыдущего):

```
) of (
  nop =          0b00000000,
  inr(Nd) =      0b00___100,
  dcr(Nd) =      0b00___101,
  hlt =          0b01110110,
  mov(Nd,Ns) =   0b01______,
```

Не допускается объявление термов с пересекающимися полями, т.е. с полями, битовые маски которых имеют общие биты. Также сообщается об ошибке, если терм является недостижимым, т.е. полностью маскируется предшествующими термами.

Кроме кода операции и некоторых её параметров, машинные команды (или другие данные подобного вида) могут иметь и другие аргументы, закодированные в последующих байтах (конечно, если это не команды RISC-процессора). Наличие таких аргументов зависит от выбранного типа команды. Чтобы можно было описывать такую зависимость, существует разновидность конструктора CASE, в которой для выбора варианта используется имя выбранного терма в перечислении термов (см. п. 2.5.7).

На имя терма можно ссылаться в выражениях так же, как на имя константы из перечисления термов: `<Имя типа>.<Имя терма>`. Значением такого выражения является *база терма*. Кроме того, в выражениях можно ссылаться на поля термов, так же, как на поля записей, при этом происходит ошибка вычисления выражения, если у выбранного терма упомянутое поле отсутствует. Пример использования перечисления термов см. в приложении 3.3.1.

2.5.5. Множество.

Является аналогом типа "множество" (set) в языке Паскаль **[33]**, т.е. значение бита, входящего в элемент данных, относящийся к этому типу, интерпретируется, как информация о вхождении соответствующего номеру бита элемента в рассматриваемое множество. При этом элементам множества в конструкторе данного типа сопоставляются имена. Наиболее частое применение данного конструктора - описание поля, содержащего набор битовых признаков. Синтаксис конструктора множества:

```
'SET' [<Int Expr>] 'OF' ('
  {['~']<Имя>{'='<Int Expr>|'^'<Int Expr>|} , ','}* ')'
```

Выражение после ключевого слова `SET` задаёт размер множества (максимальное "число элементов"), если это выражение опущено, то размер определяется по максимальному среди упомянутых в скобках констант номеру бита (размер больше этого номера на 1). Максимальный размер множества пока ограничен 32-мя элементами, т.к. значение элемента данных этого типа рассматривается как беззнаковое целое число со специальным способом отображения. По этой же причине на данный тип влияет установленный порядок байтов.

Если перед именем элемента множества поставлен признак отрицания `'~'`, то бит, сопоставленный этому элементу, интерпретируется не как признак вхождения элемента во множество (0-не входит, 1-входит), а как признак исключения (1-не входит, 0-входит).

После знака `'='` указывается выражение, задающее номер бита для элемента `<Имя>`, а после знака `'^'` - выражение, задающее битовую маску ($2^{\text{<номер бита>}}$). Таким образом, в последнем случае значение выражения должно

быть степенью 2. Также, как в определении перечислимого типа, если выражение для определения номера бита не задано, то имени сопоставляется номер предыдущего бита плюс 1, а первому имени - 0.

Пример (из приложения 3.2.1):

```
TClassAccessFlags set 16 of (
  ACC_PUBLIC =    0,
  ACC_FINAL =     4,
  ACC_SUPER =     5,
  ACC_INTERFACE = 9,
  ACC_ABSTRACT = 10
)
```

Другой пример, использующий битовые маски, (из описания заголовков ARJ **[34]**):

```
TArjFlags set 8 of (
  OLD_SECURED ^ 0x02,
  VOLUME  ^ 0x04,//indicates presence of succ. Vol.
  PATHSYM ^ 0x10,//indicates archive name translated
  BACKUP  ^ 0x20, // indicates backup type archive
  SECURED ^ 0x40
)
```

Этот способ задания множеств был реализован для того, чтобы легче было переносить информацию из тех описаний форматов, в которых флаги заданы своими битовыми масками, а не порядковыми номерами.

Так же, как имена констант перечислимого типа, имена элементов множества являются локальными для данного типа, и, чтобы сослаться на них в выражениях, надо использовать тот же синтаксис: `<Имя типа>.<Имя элемента>`. При этом значением такой константы будет *соответствующая биту маска*, а не номер бита, т.е., например, `TClassAccessFlags.ACC_FINAL` = 2^4 = 16.

Значение элемента данных типа множество отображается как:

```
'['{<Имя вошедшего элемента>,','}* ['|'<Остаток>]']'
```

т.е. сначала через `','` перечисляются все именованные элементы, вошедшие в представленное данным значением множество, а затем, если остались неименованные биты, не равные 0, то состоящее из этих битов значение отображается после разделителя `'|'` как шестнадцатеричное число. Например, для поля `access_flags` с типом `TClassAccessFlags` значение 0x401 будет отображаться, как

`access_flags:``[ACC_PUBLIC,ACC_ABSTRACT]  ,`

а значение 0x1403 - как

`access_flags:``[ACC_PUBLIC,ACC_ABSTRACT|1002]  ,`

В битовом режиме множество занимает столько бит, каков его размер, а в байтовом размер в памяти выравнивается на границу байта.

2.5.6. Запись.

Состоит из фиксированного набора именованных полей. Может содержать поля переменного размера. Каждое следующее поле следует в памяти непосредственно после предыдущего, т.е. запись всегда является упакованной (packed record) в терминах языка Паскаль и, чтобы учесть выравнивание полей, надо использовать специально для этого предназначенный тип данных (п. 2.5.11). Синтаксис записи:

```
'STRUC' ['PAS'] NL
 { {<Определение типа> <Имя поля>, NL}*
  |{<Имя поля> ':' <Определение типа>, NL}* }
'ENDS'
```

Второй вариант синтаксиса для задания полей записи используется при наличии модификатора `PAS` после ключевого слова `STRUC`. Его удобно использовать при переводе описаний структур данных с языка Паскаль на FlexT.

При создании записи могут автоматически добавляться правила для вычисления размера всей записи по размерам её полей или размера одного из полей по размерам остальных и размеру всей записи - в зависимости от того, что является известным. Кроме того, такие правила всегда можно добавить явно - в блоке определений. Для того, чтобы размер поля можно было определить автоматически, необходимо, чтобы размеры всех предшествующих полей были *известны*, а размеры всех последующих полей - *фиксированы*. Наиболее частый случай использования этой возможности - определение размера последнего поля записи, когда известен размер всей записи. Пример (формат TTF **[35]**):

```
TEmbedBitmapLoc(Len) struc
  FIXED ver //initially defined as 0x00020000
  DWORD numSizes //Number of bitmapSizeTables
  TBitmapSizeTableArray(@.numSizes) SzTbl
  raw[] rest
ends:[@:Size=@:Len]
```

Здесь размер всей записи был задан вне её и передаётся в тип через параметр, поэтому размер поля `rest` определяется автоматически.

2.5.7. Вариант.

Тип составляющей определяется в зависимости от значения параметра варианта - выбранного случая. Поддерживаются три типа вариантов: с числовыми значениями, со строковыми значениями и с именами термов из перечисления термов. Синтаксис:

```
'CASE' <Expr> 'OF'
{  {<Интервал значений>,  ','}+  ':'  <Определение типа>,
NL}*
['ELSE' < Определение типа> NL]
'ENDC'
```

Вид `<Интервала значений>` зависит от типа варианта:

- {<Int Expr>|<Int Expr> '..' <Int Expr>} , если это - целочисленный вариант;
- <Строковая константа> , если это - строковый вариант;
- <Имя терма> , если это - вариант по перечислению термов.

Строковые варианты являются нечувствительными к размеру символов.

При ссылке на случай варианта в выражениях в качестве его идентификатора можно использовать любое значение из соответствующего списка констант, при этом возникает ошибка вычисления выражения, если на самом деле выбран другой случай (но не происходит ошибки, если выбрано другое значение из того же списка).

Так же, как и в типе данных "запись", происходит автоматическое добавление правил для определения размера типа по размерам составляющих и наоборот. А именно, если размеры типов для всех случаев известны, то и размер всего варианта является известным, при этом, если все эти размеры фиксированы и равны между собой, то и размер всего варианта является фиксированным. С другой стороны, если задан размер варианта, то это значение передаётся типу выбранного случая, если у последнего размер неизвестен.

2.5.8. Проверка.

В отличие от варианта, использует для выбора типа содержимого условия корректности (блок assert) предполагаемых типов содержимого. Синтаксис:

```
'TRY'
  {<Имя случая>':' < Определение типа>,NL}*
'ENDT'
```

Типы проверяются в порядке перечисления. <Имя случая> используется для ссылок на тип содержимого в выражениях.

Пример применения такого подхода (фрагмент описания формата DVI **[36]**):

```
TChars7 array of Char ?@>127!void;
TDVIOpCode enum uint1 (
  ....................................
)
.....................................
TDVIOp struc
  TDVIOpCode Op
  case @.Op of
.....................................
  endc Prm
ends: assert[@.Op>=0x80]

TFntNum enum uint1 @-TDVIOpCode.op_fnt_num_0; :
  assert[@>=TDVIOpCode.op_fnt_num_0,
         @<=TDVIOpCode.op_fnt_num_63]
TDVIRec try
```

```
    FN: TFntNum
    Op: TDVIOp
    S: TChars7
  endt
```

В формате DVI значения байтов меньшие, чем 128, интерпретируются как коды символов, значения в диапазоне от `TDVIOpCode.op_fnt_num_0=171` до `TDVIOpCode.op_fnt_num_63=234` интерпретируются как команды выбора ранее определённого шрифта с номером `@-171`, а остальные значения байтов - как коды команды различного назначения, за которыми могут следовать данные этих команд.

2.5.9. Массив.

Состоит из переменного числа однотипных элементов. Размер массива может ограничиваться либо заданием числа элементов, либо заданием размера, либо стоп-условия (как с ASCIIZ строкой). Синтаксис:

```
'ARRAY' [ '[' [<Int Expr>] ']' ] 'OF' <Определение типа>
   {  '?' <Int Expr> ['!'<Определение типа>]
    | ',' ['<'] <Значение> ';' | }
   ['TAKES' <Int Expr> ';']
```

Число элементов массива может быть опущено и задано впоследствии, например, в блоке утверждений содержащего данный тип сложного типа через обращение к параметру типа `Count`. С другой стороны, в этом блоке вместо числа элементов может быть задан размер массива через обращение к параметру `Size`.

Если после определения типа элемента массива стоит символ `'?'`, то далее следует логическое стоп-выражение, которое оценивается для каждого элемента массива и истинность которого указывает на последний элемент. В этом случае тип последнего элемента может отличаться от типа остальных элементов массива, например, вместо целой записи может остаться только поле с признаком окончания. Для учёта этой возможности тип последнего элемента может быть задан отдельно после символа '!'.

Для более простого случая, когда тип элемента массива является целочисленным или символьным, вместо стоп-условия можно указать стоп-значение. Стоп-значение указывается после символа `','`, и признаком последнего элемента массива является равенство значения этого элемента стоп-значению.

В некоторых форматах данных под строковое поле выделяется фиксированный размер памяти, но используется только часть этого поля. При этом конец значения может определяться стоп-значением в некотором символе, т.е. для определения конца значения надо просматривать массив от начала в поисках признака окончания. Чтобы описать этот случай на языке FlexT, надо указать размер массива, но при этом задать и стоп-значение. Пример (имя поля в формате DBF, **Рис. 3**):

```
  array[11] of Char,0; Name //Имя - ASCIIZ строка
```

Другой вариант определения размера значения - заполнение справа некоторым символом. В этом случае для определения размера значения надо просматривать массив *с конца* в поисках первого символа, который отличается от символа-заполнителя. На использование заполнения указывает символ '<' перед стоп-значением. Пример (библиотека COFF **[25]**):

```
IMAGE_ARCHIVE_MEMBER_HEADER struc pas
  Name: array[16]of char,<' ';
  Date: array[12]of char,<' ';
 ..............................
  EndHeader: array[2]of char;
ends:assert[@.EndHeader='`'#10]
```

Возможен вариант, когда один элемент массива может занимать несколько его ячеек. Для учёта этого случая используется блок числа индексов 'TAKES' в определении массива. Заданное в этом блоке выражение оценивается на каждом элементе и, если это выражение успешно вычисляется, то оно определяет количество номеров, занимаемых данным элементом, иначе считается, что элемент занимает один номер. Из-за такой интерпретации ошибок вычисления данное условие удобно записывать в виде <Сколько индексов> when <На каких элементах>. Пример использования блока числа индексов см. в приложении 3.2.1 (тип поля C_pool в записи TClassFile), пояснения см. в п. 4.4.

Если тип элемента массива является символьным (п. 2.5.9), то содержимое этого массива может интерпретироваться как строковое значение. Так, это содержимое отображается в виде строки и может использоваться для выбора случая в строковом варианте (п. 2.5.7).

Следует напомнить, что любое определение типа можно брать в круглые скобки, и за счёт этого можно избежать неоднозначностей в определении принадлежности информации о стоп-условии и дополнительных блоков определения типа при объявлении массива с элементами-массивами. Пример:

```
TMatrix(SM,SR) array of (array of
          int:[@:Size=@@:SR]):[@:Size=@:SM]
```

2.5.10. Сырые данные.

Служат для отображения блока данных, про которые ничего не известно, кроме их размера, в виде шестнадцатеричного дампа. Синтаксис:

```
'RAW' '[' [<Int Expr>] ']' ['AT' < Int Expr> ';']
```

Значение в скобках определяет размер блока и может быть опущено, чтобы, например, задать этот размер в блоке утверждений содержащего данный тип сложного типа. Второе выражение задаёт базовый адрес, т.е. адрес, смещения от которого отображаются в дампе, как адреса строк. Это позволяет, например, отобразить адреса неиспользуемых данных некоторой записи относительно начала этой записи.

2.5.11. Выравнивание.

Чтобы явно описать случай выравнивания смещения следующих данных от некоторого начального адреса на величину, кратную некоторому числу используется специальный тип данных, который отображается как сырые данные. Синтаксис:

```
'ALIGN' <Int Expr> ['AT' < Int Expr> ';']
```

значение первого выражения должно быть степенью 2, оно задаёт границу выравнивания. Второе выражение задаёт базу выравнивания: если оно опущено, то выравнивание происходит относительно начала блока памяти. Данные этого типа отображаются так же, как сырые данные. База выравнивания влияет не только на способ отображения, но и используется при определении размера блока выравнивания: концом блока является первый адрес после его начала, который имеет смещение от базы выравнивания кратное границе выравнивания.

2.5.12. Адресные пространства, адресные блоки и указатели

Некоторые составляющие компактных блоков могут быть указателями - ссылками на другие элементы данных. В простейшем случае значением указателя может быть смещение от начала файла или какая-то линейная функция от него. В более сложном случае значением указателя может быть виртуальный адрес в некотором *адресном пространстве*. В это адресное пространство могут входить один или несколько *адресных блоков*, каждый из которых характеризуется своим начальным виртуальным адресом и виртуальным размером (этот размер нужен для определения неинициализированных данных программ). Пример таких данных - 32-разрядный исполняемый файл **Windows 95/NT [25]**.

Таким образом, элемент данных может являться адресным блоком, при этом может быть указано его адресное пространство, виртуальный адрес в этом пространстве и виртуальный размер. При объявлении типа данных указателя может быть задано адресное пространство, в которое он ссылается (а также смещение и коэффициент).

Синтаксис указателя:

```
'^' <Определение типа> ['*' <Int>[',']] ['-' <Int>]
    ['HIDEREF'] ['FIXUP' {'ON'|'OFF'}]
    ['NIL' {'=' <Int> | ':' <Int Expr> | '-'}]
    {'NEAR' ['=' <Имя типа>] <Ссылка на блок> ]|
    '=' <Имя типа>} [[','] 'REF' '=' <Int Expr> ';']
```

После `'NIL'` может либо указываться конкретное значение `'=' <Int>`, соответствующее отсутствию ссылки, либо условие `':' <Int Expr>` для проверки на отсутствие ссылки, либо то, что любое значение надо рассматривать, как ссылку (`'-'`). По умолчанию, в качестве `Nil` рассматривается значение, равное 0. `'NEAR'` означает, что ссылка идёт в конкретный блок, иначе она рассматривается как виртуальный адрес в текущем

адресном пространстве. '=' <Имя типа> означает, что размер указателя равен размеру этого типа. Если адрес ссылки не совпадает с целочисленным значением базового типа, то после 'REF' можно указать выражения для вычисления адреса.

2.5.13. Предварительное объявление типа.

Возможны ситуации, когда между типами существует циклическая зависимость, например, при вычислении выражений для значений некоторых свойств типа используются обращения к полям записи, в состав которой входит поле данного типа. В таких случаях необходимо использовать объявление типа при помощи ключевого слова forward. Строго говоря, данная синтаксическая конструкция не является конструктором типа, но её удобнее рассмотреть в общем контексте механизмов определения типов. Пример циклической зависимости (из описания формата DFM **[37]**):

```
TValList forward
.....................
TValue struc pas
  T: TValueType
  D: case @.T of
    vaList: TValList
    vaInt8: byte
.....................
  endc
ends
TValList array of TValue ?@.T=TValueType.vaNull;
```

Здесь в состав записи TValue может входить массив, состоящий из таких же записей.

В случае, когда предварительно объявляемый тип имеет параметры, эти параметры должны быть указаны и в объявлении через forward, причём, оба списка параметров должны полностью совпадать. Неявное предварительное определение типа происходит при использовании имени ещё не объявленного типа в качестве базового типа указателя, однако, этот приём можно использовать только для типов без параметров. Если базовый тип имеет параметры, то нужно их определить, поэтому необходимо использовать явное предварительное определение. Пример (фрагмент описания формата TTF **[35]**):

```
TTTFtable(hDir,Len) forward
PTTFtable(hDir,Len) ^TTTFTable(@:hDir,@:Len) near=DWORD
```

2.5.14. Машинные команды.

В дизассемблерах также используется специальный тип указателя - указатель на код, который означает, что по данному смещению находится начало исполняемого кода. Сначала для спецификации формата кодирования машинных команд использовался специальный язык, но рассматриваемый подход может и непосредственно применяться для этих целей. При этом, в

отличие от широко известного подхода **[29]** к этой задаче, здесь мы рассматриваем машинную инструкцию как некоторый специальный тип данных, применяя для его описания наработанные для произвольных данных методы.

Для описания кодирования машинных инструкций и признаков команд, завершающих код (таких, как RET или JMP), используется специальный тип данных, подобный массиву со стоп-условием:

```
'CODES' 'OF' <Тип инструкции> '?'<Стоп-выражение> [ '!'
  <Тип последней инструкции> ] ';'
```

Таким образом, кодирование отдельной машинной команды может описываться любыми конструкциями языка FlexT, а условие для определения команд с безвозвратной передачей управления задаётся стоп-выражением в типе CODES. Это условие используется впоследствии при разборе блоков кода дизассемблером FlexT, являющимся пока достаточно примитивным, т.е. не использующим никакую другую информацию о семантике машинных команд (см. п. 3.8.2).

2.6. Предопределённые типы.

При написании спецификаций было бы достаточно использовать конструкторы, описанные в (п. 2.5), однако, некоторые часто используемые типы в языке FlexT являются предопределёнными, некоторые из них при этом реализованы более эффективно. Предопределённые типы можно разбить на следующие группы:

1) Беззнаковые целочисленные типы:
   - `byte` **`num`**`+(1)`
   - `word` **`num`**`+(2)`
   - `ulong` **`num`**`+(4)`
2) Знаковые целочисленные типы:
   - `sint` **`num`**`+(1)`
   - `int` **`num`**`+(2)`
   - `long` **`num`**`+(4)`
3) Бестиповый указатель:
   - `pointer ^`**`void`**
4) Указатели на код Intel80x86:
   - `codeofs ^TIntel80x86Codes` **`near`**`=word CODE`
   - `codeptr ^TIntel80x86Codes =ulong`

   Где TIntel80x86Codes обозначает аналог типа CODES для встроенного дизассемблера машинных команд.
5) Строковые типы:
   - `str` **`array of`** `char:[@:Size=@[0]+1]` , первый символ не отображается. Тип str является представлением паскалевских строковых констант. Этот тип более правильно было бы объявить так:

     `str` **`struc`**

```
    byte L
    array[@.L]of char S
 ends:displ=(@.S)
```

но тогда это отличалось бы от того, как это сделано в Паскале.

- `pchar array of char,0`
  Этот тип является представлением строковых констант C (ASCIIZ строки).

6) Выравнивания:

- `align2 align[2]`
- `align4 align[4]`
- `align8 align[8]`
- `align18 align[16]`

Все предопределённые типы имеют нормальный (LSB) порядок байтов и байтовое размещение. Если требуется использовать подобные типы, но, например, с другим порядком байтов, то такие типы надо объявить явно: так же, как это показано в данном пункте.

2.7. Интерфейсы.

Все элементы языка FlexT, описанные выше, были реализованы в его интерпретаторе. В этом пункте будет рассмотрен механизм интерфейсов, который, хотя и не реализован в текущей версии интерпретатора FlexT, имеет большое значение для дальнейшего развития этого языка.

Некоторый элемент данных может быть представлением какого-то более абстрактного типа данных **[14]**. Простейший пример: тип *"индекс"* в ***.obj*** - файлах **[38]**:

```
if (first_byte & 0x80)
  index_word = (first_byte & 7F) * 0x100 +
 second_byte;
else
  index_word = first_byte;
```

Т.е. для экономии места небольшие (меньше `0x80`) целые числа кодируются одним байтом, а числа побольше - двумя. Для описания такого типа данных придётся использовать запись с вариантным полем, но во всей остальной спецификации эти низкоуровневые подробности представления индексов абсолютно неинтересны и было бы лучше иметь возможность заменять все ссылки на данные типа "индекс" тем числом, которое эти данные представляют.

Для описания абстрактных типов данных в языке FlexT предполагается реализовать механизм интерфейсов. Описание типа "индекс" могло бы выглядеть следующим образом:

```
TIndex struc
  Byte B
  case @.B and 0x80 of
    0x80: Byte
```

```
    endc B0
  ends:is IUInt default (val=((@.B and 0x7F)*0x100+
      @.B0.0x80)exc @.B)
```

Здесь дополнительный блок определений типов `is` служит для установления связи между типом и интерфейсом беззнакового целого числа `IUInt`. Ключевое слово `default` означает, что при обращении в выражениях к элементу данных типа `TIndex` по умолчанию будет происходить его приведение к этому интерфейсу. В свою очередь, свойство `val` интерфейса `IUInt` также может быть объявлено как `default`, в результате чего обращение в выражениях к элементу данных типа `TIndex` будет возвращать значение свойства `val`.

Предполагается, что в определении интерфейса можно будет указать родительский интерфейс, объявить свойства данного интерфейса, а также, может быть, определить правила для вычисления значений одних свойств через другие. Правило может активизироваться, если все необходимые для его вычисления значения оказываются известными.

Интерфейсы могут использоваться для спецификации интерпретаций данных более высокого уровня, чем просто идентификация типов. С другой стороны, использование интерфейсов может помочь описывать сложные зависимости между данными при идентификации типов. Например, при помощи интерфейса объявления типа данных `IDataType` можно было бы сообщить системе, что некоторый элемент данных содержит метаинформацию о некотором, специфическом для конкретного файла, типе данных, после чего можно было бы использовать эту информацию, например, при описании находящейся в файле константы этого типа.

2.8. Ограничения предлагаемого подхода.

Рассмотрим некоторый формат упаковки данных, например **[39]**. Как правило, данные в этом формате содержат поток битов. Для того, чтобы правильно интерпретировать следующие несколько битов, требуется обратиться к некоторым структурам данных, динамически перестраиваемым в процессе чтения и зависящим от всего ранее прочитанного потока. Это надо сделать не только для того, чтобы правильно интерпретировать числовое значение, как, например, номер ранее встречавшейся последовательности байтов, но и просто для того, чтобы определить размер следующего элемента данных. Таким образом, в данном случае мы имеем дело с зависимостью неограниченного размера между элементами данных, поэтому её нельзя представить выражением фиксированного размера, а в языке FlexT в настоящее время поддерживаются только такие выражения. Процесс чтения упакованных данных всегда излагается как алгоритм, поэтому его невозможно представить чисто декларативно.

Другой пример зависимости неограниченного размера - задача разделения содержимого исполняемого файла на код и данные. Здесь число, помещённое в регистр в одном месте программы, может оказаться адресом процедуры, на

которую передаётся управление посредством косвенного вызова в другом месте программы, сколь угодно далёком от места присваивания, при этом на сохранение или разрушение значения в регистре могут влиять все команды, которые могли быть выполненными между двумя этими событиями. Для анализа таких зависимостей также потребовалось бы развивать императивную составляющую языка.

Впоследствии планируется включить в интерпретатор FlexT элементы функционального программирования, что может позволить снять некоторые из упомянутых ограничений. Предполагается использовать именно функциональное программирование, поскольку такие программы достаточно легко поддаются автоматическому анализу **[40]**.

Упаковка данных может вызывать проблемы не только из-за неограниченности размера, но и из-за эффекта, который можно назвать *двойной буферизацией*. Суть этого эффекта состоит в том, что некоторые сколь угодно сложные данные могут быть сначала записаны в буфер, после чего содержимое этого буфера может быть упаковано, причём, способ упаковки может быть очень простым, например, RLE (Run Length Encoding, **[22]**), что позволяет описать способ извлечения закодированных данных на языке FlexT. Однако, в языке отсутствует механизм для сопоставления структуры распакованным данным, поэтому, например, нельзя извлечь из них значение некоторого поля записи. Для снятия этого ограничения можно расширить механизм блоков памяти (п. 3.5) на упакованные потоки.

3. Детали реализации языка FlexT.

В этой главе рассмотрим наиболее интересные детали реализации интерпретатора языка FlexT, а также детали реализации различных программных систем, использующих этот интерпретатор.

3.1. Механизм вызова типов.

{Этот пункт исключён из электронной версии диссертации.}

3.2. Вычисление значений параметров и свойств типа по необходимости.

При передаче фактических параметров вызова типа в языке FlexT используется механизм вычисления по необходимости, известный также как механизм ленивых вычислений **[40]**. Также в ленивом режиме вычисляются значения свойств типов. Это означает, что, например, при вызове типа в запись о значении параметра просто помещается ссылка на выражение для вычисления этого значения. Само вычисление происходит лишь при необходимости получения значения.

В некоторых случаях использование ленивых вычислений позволяет избежать переполнения стека. Пример (фрагмент описания формата VXD Windows-95 **[27]**):

```
PFixupPageRecTbl forward
TFIXUP_PAGE_TABLE(Cnt) array[@:Cnt] of PFixupPageRecTbl
PFixupPageRecTbl ^TFixupPageRecTbl - FixupTblOfs nil-
  near=ulong : [@^:Sz=
    ((@:@ as TFIXUP_PAGE_TABLE)[#+1]-@) exc 0]
```

В этом фрагменте записано, что массив `TFIXUP_PAGE_TABLE` содержит указатели на блоки с типом `TFixupPageRecTbl`, блоки размещаются в памяти непосредственно друг за другом, поэтому для определения размера блока (которому соответствует значение параметра `Sz`) достаточно вычислить разность между двумя соседними указателями. Этот невинный с виду фрагмент приводил к переполнению стека до того, как были реализованы ленивые вычисления (и послужил стимулом к их реализации). Проблема состояла в том, что обращение к следующему элементу массива в выражении для `@^:Sz` текущего элемента приводило к *вычислению* выражения для `@^:Sz` в блоке утверждений этого элемента, что, в свою очередь, вызывало обращение к следующему за ним элементу и т.д. В режиме вычислений по необходимости ничего подобного не происходит, поскольку выражение для `@^:Sz` не вычисляется для следующего элемента массива.

3.3. Представление информации о типах данных.

Для представления в памяти информации об определённых в спецификации на языке FlexT типах данных в коде интерпретатора используется иерархия классов, базовым классом в которой является тип `TDataType`.

3.3.1. Тип TDataType.

{Этот пункт исключён из электронной версии диссертации.}

3.3.1.1. Основные методы.

{Этот пункт исключён из электронной версии диссертации.}

3.3.1.2. Операции со свойствами типов:

{Этот пункт исключён из электронной версии диссертации.}

3.3.1.3. Операции с параметрами типов:

{Этот пункт исключён из электронной версии диссертации.}

3.3.2. Учёт зависимостей между типами данных.

{Этот пункт исключён из электронной версии диссертации.}

3.3.3. Иерархия классов.

{Этот пункт исключён из электронной версии диссертации.}

3.4. Представление информации о переменных.

{Этот пункт исключён из электронной версии диссертации.}

3.5. Представление информации о блоках памяти.

В простейшем случае весь файл с данными некоторого формата может рассматриваться как один блок памяти, при этом для указания положения переменных в таком файле используются физические адреса, т.е. смещения от начала файла. Такими свойствами обладает большинство форматов, описанных на FlexT в настоящее время. Однако, некоторые форматы и, в частности, практически все форматы исполняемых и объектных файлов, устроены сложнее, так что их нельзя качественно описать без использования механизма вложенных блоков памяти и виртуальных адресных пространств. В частности, большинство форматов исполняемых файлов не только содержит информацию о виртуальных блоках, но и указатели в виртуальное адресное пространство из полей переменных физического адресного пространства, например, адрес точки входа в заголовке файла.

В качестве примера формата файлов, который не содержит исполняемый код, но разбивается на блоки, можно привести формат ANI (Windows animated cursors) **[41]**, составными частями которого являются блоки с данными в формате ICO (Windows icons) **[16]**.

В наиболее общем случае адресное пространство может содержать несколько виртуальных блоков, а каждый виртуальный блок может иметь в качестве носителя несколько физических блоков, каждый из которых содержит образ памяти некоторого интервала виртуальных адресов. Физический блок имеет, кроме виртуального адреса, также и физический адрес, который, в свою очередь, может быть не только файловым адресом, но и адресом в некотором другом виртуальном адресном пространстве (см. Рис. 5). С другой стороны, очень распространённым вариантом адресного пространства является гораздо более простой случай, когда некоторому интервалу физических адресов сопоставляется интервал виртуальных адресов того же размера, т.е. адресное пространство содержит один виртуальный блок, который базируется на одном физическом блоке.

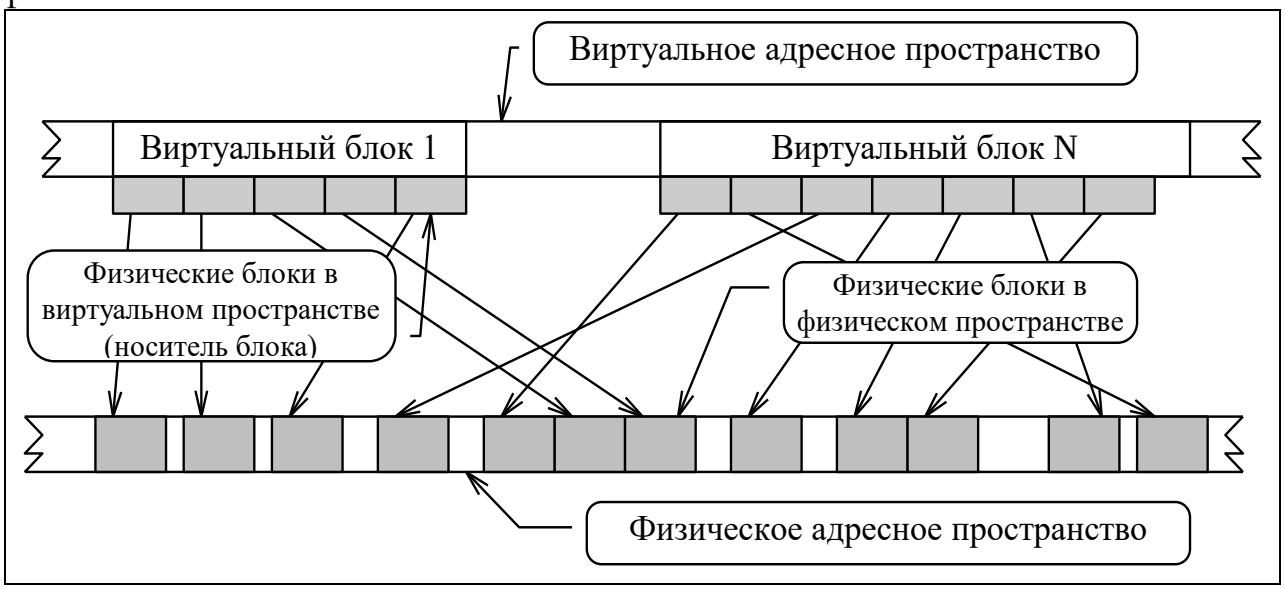


Рис. 5. Представление содержимого виртуальных блоков физическими блоками.

{Оставшаяся часть этого пункта исключёна из электронной версии диссертации.}

3.6. Обработка типов данных с битовым размещением.

{Этот пункт исключён из электронной версии диссертации.}

3.7. Влияние порядка байтов на определения типов.

На Рис. 6 показано представление в памяти 16-битного значения, соответствующего определению типа **num**+(2). При прямом порядке байтов (LSB) первым (т.е. по меньшему адресу) в память записывается младший байт двоичного представления числа, а при обратном (MSB) - первым записывается старший.

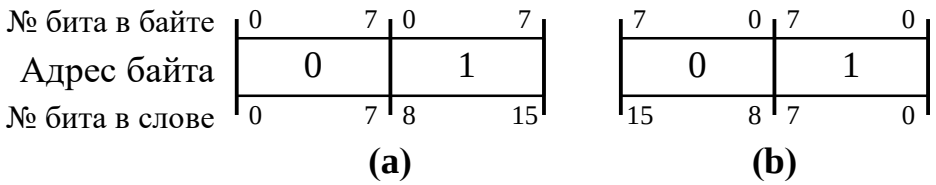


Рис. 6. Прямой и обратный порядки байтов в слове (a)-прямой, (b)-обратный.

Из рисунка видно, что в первом случае естественно представлять себе, что биты в байтах ориентированы в направлении роста адресов, а во втором - в обратном направлении, так как только при такой ориентации соседние биты (7 и 8) окажутся рядом. Таким образом, можно считать, что обратному порядку байтов соответствует и обратный порядок битов в байтах.

Кроме конструкторов целых чисел, от порядка байтов зависят также любые типы с битовым размещением. Рассмотрим следующее объявление типа:

```
type bit
  TBitRec struc
    num+(6) A
    num+(8) B
    num+(2) C
  ends
```

На Рис. 7 показано побитовое размещение в памяти значений полей этой записи в зависимости от порядка байтов.

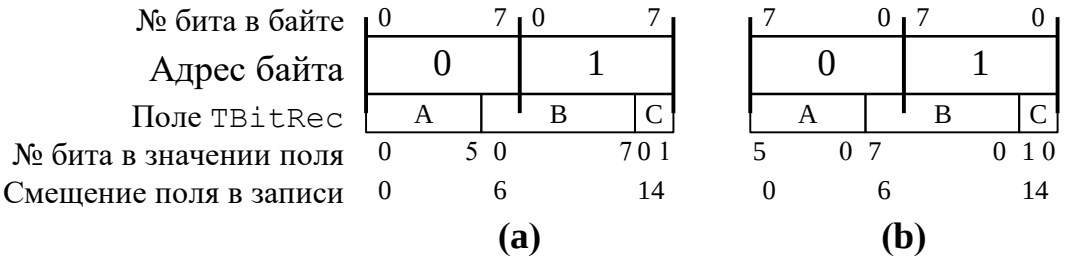


Рис. 7. Побитовое размещение полей записи TBitRec (a)-прямой порядок байтов, (b)-обратный.

Видно, что байты объединяются в поток битов в соответствии с заданным для них порядком. Битовые элементы данных всегда размещаются в этом потоке в порядке своих номеров, поэтому на рисунке смещения полей не зависят от порядка байтов. Однако, на Рис. 7(a) первым в потоке размещается младший бит значения поля, а на Рис. 7(b) - старший, что полностью соответствует порядку битов в байтах.

Оба рассмотренных способа размещения удовлетворяют следующему требованию: элементы данных с бóльшими номерами размещаются в байтах с бóльшими или равными адресами. Таким образом, если, например, объявлен тип

```
type bit
  TCntRec struc
```

```
    num+(9) Len
    array[@.Len]of num+(3) Tbl
  ends
```

то он может быть корректно интерпретирован. Иначе, т.е., если размещать элементы данных в порядке, обратном порядку адресов, то элемент данных `Len` не сможет быть найден, т.к. в памяти он должен был бы следовать *после* `Tbl`.

Однако, возможны случаи, когда при обратном порядке байтов элементы данных требуется размещать в порядке, *обратном* порядку адресов байтов. Это не противоречит сказанному в предыдущем абзаце, так как в таких форматах размер всей структуры фиксирован и известен заранее, поэтому её можно было бы разбирать и с конца (в смысле порядка адресов). Пример (информация о перемещаемых адресах формата ELF **[30]**):

```
#define ELF32_R_SYM(i) ((i)>>8)
#define ELF32_R_TYPE(i) ((unsigned char)(i))
```

Здесь сообщается, что для извлечения полей `SYM` и `TYPE` из 32-битного поля, требуется прочитать значение этого поля в соответствии с порядком байтов, а затем младшие 8 бит этого значения будут соответствовать полю `TYPE`, а старшие 24 бита - полю `SYM`. Пока для представления такой структуры на FlexT приходится писать следующий код, использующий условную компиляцию, для изменения порядка полей в записи:

```
TE_R_SYM num+(3)
%$IF Ident.ei_encoding=TElfEncoding.ELFDATA2MSB;
TE_R_INFO struc
  TE_R_SYM hSym
  TE_R_TYPE Typ
ends
%$ELSE
TE_R_INFO struc
  TE_R_TYPE Typ
  TE_R_SYM hSym
ends
%$END
```

Может быть, в дальнейшем следует явно поддержать такой способ размещения битовых полей, контролируя при этом, что размер типа определяется извне, либо фиксирован.

3.8. Процесс исполнения программы FlexT и автоматическое определение формата файла в программах просмотра.

Как уже упоминалось, в текущей версии интерпретатора FlexT программа загружается и исполняется *после* загрузки данных. При определении формата файла используется комбинированная стратегия: учитывается как расширение имени файла, так и его содержимое. Такой подход отличается от принятого в

большинстве универсальных средств просмотра файлов, например, QuickView в Windows **[42]**, где способ просмотра определяется исключительно по расширению. Определить формат файла только по расширению не всегда возможно, поскольку одному расширению может соответствовать несколько форматов, например, расширение OBJ и форматы OMF **[38]** и COFF **[25]**, а один формат может иметь несколько расширений, например, исполняемые файлы Windows могут иметь расширения: EXE, DLL, OCX и др.

По умолчанию, расширению `<Ext>` соответствует файл с описанием формата `<Ext>.rfh`. Кроме того, существует специальный файл описания расширений `REF.CFG` с информацией о связи расширений и файлов с описаниями форматов. Каждая строка этого файла имеет вид:

```
<Ext>':' {<FName>, ' '}+
```

и несёт информацию о том, что для поиска формата файла с расширением `<Ext>` надо последовательно, в порядке упоминания, проверить файлы `<FName>` с описаниями форматов. `<FName>` включает в себя расширение. Для файлов, упомянутых в `REF.CFG` рекомендуется использовать расширение `.rfi`. Таким образом, если расширение `<Ext>` было упомянуто в файле описания расширений, то происходит поиск по списку описаний форматов из этого файла, иначе - проверяется только файл `<Ext>.rfh`. Текущая версия файла `REF.CFG` показана в приложении 1.1.

Для проверки соответствия данных файла некоторому формату используется информация, содержащаяся в блоках утверждений (п. 2.4.2). Выбирается первое описание, у которого все утверждения истинны. Если ни одно из описаний не подходит, то всё содержимое файла отображается просто как шестнадцатеричный дамп.

В параметрах программы просмотра BinView можно указать путь для поиска файлов FlexT. По умолчанию, это - подкаталог REF в каталоге BinView. Поиск описаний форматов происходит в каталогах, указанных в пути поиска.

Кроме общего описания формата, некоторому файлу может соответствовать и "личное" описание, которое должно содержаться в том же каталоге, где находится этот файл, иметь то же имя и расширение `.ref`. Это описание может потребоваться для объявления дополнительных структур данных и адресов процедур, которые не являются частью описания формата и были обнаружены в результате изучения содержимого конкретного файла.

Процесс исполнения программы можно разделить на следующие шаги:

1. поиск и загрузка описания формата;
2. загрузка "личного" описания файла;
3. прослеживание указателей (поиск переменных, на которые ссылаются указатели);
4. дизассемблирование;
5. отображение результата.

Рассмотрим подробнее шаги 3 и 4.

3.8.1. Процедура прослеживания указателей.

В некоторых форматах, например, 32-битных DCU **[47]** или OMF **[38]**, всё содержимое файла представляет собой одну большую структуру данных, в рассматриваемом случае это - потоки теговых записей. Таким образом, для описания таких форматов необходимо определить тип данных этой структуры и объявить переменную с этим типом и нулевым адресом. Пример (описание OMF):

```
data
  0x0000 TObjRecStm Stream
```

Однако, чаще всего большинство содержимого файла занимают структуры данных, адресуемые через указатели из других структур данных, при этом, как правило, все цепочки ссылок начинаются в полях заголовка файла. В этом случае также достаточно явно объявить лишь одну переменную, соответствующую заголовку файла, а всё остальное содержимое файла будет разобрано в результате прослеживания указателей. Примеры таких форматов: TPU/16-битный DCU, COFF, ELF.

Процесс прослеживания указателей вызывает рекурсивную процедуру прослеживания указателей для всех переменных из всех блоков памяти, которая выполняет следующие шаги:

1. Как уже упоминалось, каждая переменная имеет флаг, который показывает, были ли прослежены указатели из неё. Если он установлен, то дальнейшая обработка прерывается, иначе флаг устанавливается и процедура продолжает работу.
2. В процессе настройки параметров и свойств типа данных происходит также установка флага `toHasPtrs`, если этот тип является указателем или имеет указатели в своём составе. Если такой флаг не установлен у типа обрабатываемого элемента данных, то сразу происходит выход из процедуры.
3. Если обрабатываемый элемент данных не является указателем, то происходит рекурсивный вызов процедуры прослеживания для всех вложенных элементов данных (используется итератор ForEachVar).
4. Иначе, если это - указатель на тип CODES, то происходит запоминание точки входа в код для процедуры дизассемблирования.
5. Иначе тип данных должен быть обычным указателем. Для этого указателя происходит поиск переменной, на которую он ссылается, и, если такой переменной ещё не существует, то она создаётся. Если была создана новая переменная, то для неё также происходит вызов процедуры прослеживания указателей.

При создании переменной ей сопоставляется имя. Это имя может порождаться при помощи информации из блока именования типа переменной (п. 2.2.4.4), если таковой имеется, иначе используется квалификатор ссылки на переменную. Поскольку порождённые имена не могут упоминаться в выражениях, они не обязаны следовать синтаксису идентификаторов и могут содержать специальные символы.

При прослеживании списочных структур ссылочные квалификаторы могут иметь многократно повторяющиеся части, например, `.Next^.Next^`. В таких случаях при именовании ссылочных переменных используется *сжатие квалификатора*, в результате которого повторяющаяся часть упоминается один раз, а после неё следует `'*'<Число повторений>`. Пример (фрагмент результата разбора файла HASH.TPU - модуля интерпретатора FlexT, блок именования типа `TNameInf` закомментирован):

```
02C2:TD_THashTable.TI.RD.RefTblOfs^.Tbl[0]^.Next*4^: TNameInf = (
  Next:0000; Kind:(K:nkProc{52}; S:pub{0}); Name:'Init';
```

Для сравнения приведём вид этого же фрагмента результата разбора, при включении блока именования типа `TNameInf`:

```
02C2:Init_Inf: TNameInf = (Next:0000;
  Kind:(K:nkProc{52}; S:pub{0}); Name:'Init';
```

Таким образом, в процессе прослеживания указателей происходит поиск в глубину. Новые переменные добавляются в таблицу переменных блока и могут быть повторно рассмотрены, так как таблица переменных упорядочена по адресам. При повторном рассмотрении переменной её обработка сразу завершается, поскольку у такой переменной установлен соответствующий флаг. При отсутствии блока именования у типа ссылочной переменной, ей сопоставляется имя в соответствии с первой обнаруженной ссылкой на эту переменную, после чего остальные ссылки уже не рассматриваются, иначе имя однозначно определяется содержимым переменной.

3.8.2. Процедура дизассемблирования.

В интерпретаторе FlexT используется примитивный дизассемблер, который способен прослеживать переходы лишь по непосредственно упомянутым в командах перехода адресам, т.е. отслеживается лишь поток команд, но не поток данных. При этом такому дизассемблеру нужно знать совсем немного о способе кодирования машинных инструкций:

1. Как определить размер, занимаемый одной машинной командой;
2. Как определить, что команда безвозвратно передаёт управление;
3. Как проследить ссылки из машинной команды (переходы на другие команды);
4. Как отобразить машинную команду.

Ответы на вопросы 1, 3 и 4 предоставляет любое определение типа данных. Ответы на вопрос 2 можно дополнительно сообщить в определении типа CODES (п. 2.5.14), который является специальным случаем массива со стоп-условием (**Ошибка! Источник ссылки не найден.** и **Ошибка! Источник ссылки не найден.**). Таким образом, на самом языке FlexT можно полностью описать способ кодирования машинных команд заранее не известного процессора, так, чтобы можно было частично дизассемблировать исполняемый код для этого процессора.

Процесс дизассемблирования продолжается до тех пор, пока остаются нерассмотренные точки входа. Первоначально эти точки могут определяться в блоке CODE программы на FlexT (п. 2.4.1), а также добавляться в результате

прослеживания указателей на код из переменных (п. 3.8.1). Дополнительные точки входа могут добавляться в процессе дизассемблирования.

В результате дизассемблирования в блоках памяти выделяются *части кода*, каждая из которых характеризуется своим типом машинных команд, адресом, размером, а также имеет таблицу ссылок. Каждая запись в таблице ссылок содержит информацию о смещении в блоке целевого адреса ссылки, об адресе, откуда происходит эта ссылка, и о её типе (ссылка из данных - `'d'`, вызов - `'p'`, переход - `'j'`, цикл - `'l'`). При отображении результатов дизассемблирования выводятся адреса происхождения ссылок, что позволяет легко проследить места использования рассматриваемой части кода, особенно в программе PE Explorer, где эти адреса являются гипертекстовыми ссылками (см. Рис. 8).

```
ProcTbl1[0], ProcTbl1[1]:
@p2945(00429EA5, 00429F67, 0042A62D), @d2948(00456940, 00456944):
  004295C4: 53                    ¦S       ¦PUSH EBX
  004295C5: 8B D8                 ¦Л+      ¦MOV EBX,EAX
  004295C7: 8A 4B 05              ¦KK.     ¦MOV CL,BYTE PTR [EBX+5]
  004295CA: B8 01 00 00 00        ¦¬....   ¦MOV EAX,00000001
```

Рис. 8. Отображение ссылок на код.

Добавление точки входа сводится к добавлению ссылки на код, если её целевой адрес попадает в одну из существующих частей кода, иначе создаётся новая часть кода нулевого размера. То, что часть кода имеет нулевой размер, является признаком того, что она ещё не была рассмотрена.

Для каждой ещё не рассмотренной части кода процесс дизассемблирования последовательно рассматривает находящиеся по её адресу машинные команды. Этот процесс прерывается либо, если команда безвозвратно передаёт управление, либо при достижении следующей непустой части кода (пустые части кода поглощаются с импортом ссылок). При достижении следующей части кода происходит её объединение с рассматриваемой частью. Также объединяются смежные части кода, разделённые командой безвозвратной передачи управления, если между ними существуют переходы. Такой подход позволяет во многих случаях добиться соответствия частей кода процедурам программы.

Для прослеживания ссылок из машинных команд, например, адресов перехода, используется процедура прослеживания ссылок из переменной (п. 3.8.1).

Результатом процедуры дизассемблирования становится только информация о частях кода, но не о каждой отдельной инструкции. Когда требуется отобразить часть кода, происходит повторное дизассемблирование, но уже без всякого анализа переходов: просто каждая машинная команда в диапазоне адресов части кода декодируется и отображается. Такой подход позволяет разбирать большие исполняемые файлы, размер которых измеряется мегабайтами.

3.9. Логическая семантика спецификаций форматов.

При разработке языка FlexT, одним из основных критериев отбора способов расширения данного языка являлось сохранение его эффективности - возможности практической работы с реальными данными большого объёма. Если текущая версия программы PE Explorer может открывать и декодировать значительную часть кода файлов, занимающих несколько мегабайт, то такое положение стоит сохранять и дальше. Поэтому любые улучшения возможностей системы должны происходить без существенного увеличения затрат по памяти или по времени.

Первоначально предполагалось использовать в качестве языка спецификаций некоторую версию Пролога **[43]**, может быть, с объектными расширениями (в духе F-логики **[44]**), и с интерфейсом к обрабатываемым данным. В рамках этого подхода можно было бы добавить в интерпретатор Пролога специальные предикаты для получения информации о данных файла (число по адресу, строка по адресу и т.д.), а все остальные знания записывать непосредственно на нём. Однако, из соображений эффективности лучше поднять уровень интерфейса к данным как можно выше, поскольку на низшем уровне не требуются многие возможности логических языков, например, поиск с возвратами. Другим преимуществом специализированных языков по сравнению с непосредственным использованием Пролога является их бóльшая ориентированность на решаемую задачу, т.е. бóльшее удобство использования. Также следует упомянуть, что в данной задаче естественным направлением рассуждений чаще является прямой вывод. Таким образом, сейчас более перспективным представляется вариант использования специализированного языка спецификаций, некоторые элементы которого могут интерпретироваться логическими методами.

Теперь перейдём собственно к логической интерпретации тех знаний, для представления которых используется язык спецификации форматов данных.

В файле описания формата данных могут содержаться следующие виды утверждений, запускающих работу машины вывода:

1. о значениях констант;
2. о характеристиках (параметрах, свойствах, интерфейсах и правилах) типов данных;
3. о существовании элементов данных;
4. может быть, сюда также добавятся впоследствии определения функций и предикатов.

Машина вывода на основе этих утверждений создаёт в памяти представление информации о константах, типах и элементах данных (сейчас информация о вложенных элементах данных порождается динамически по мере необходимости).

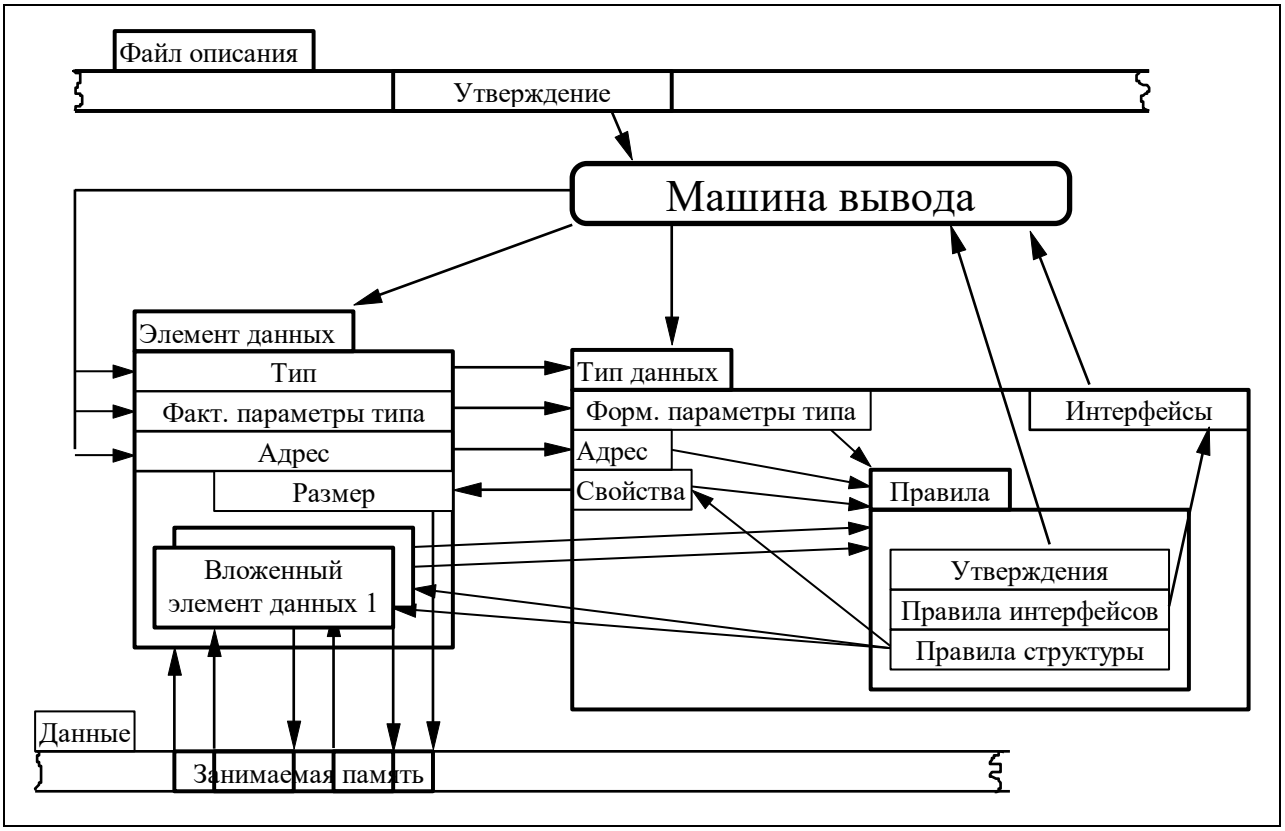


Рис. 9. Схема информационных зависимостей в интерпретаторе FlexT.

С каждым типом данных связан ряд правил, которые можно разделить на несколько видов:

1. Правила структуры выводят информацию о свойствах типов, а также о наличии и размещении вложенных элементов данных;
2. Правила интерфейсов выводят информацию о наличии интерфейсов и значениях их свойств;
3. Прочие утверждения могут содержать информацию о других элементах данных, которая по своей сути аналогична информации из файла описания.

Последний пункт требует некоторого пояснения. Примером типов, содержащих такие утверждения уже в текущей версии системы, являются указатели, поскольку обнаружение машиной вывода элемента данных типа указатель приводит к тому, что делается заключение о существовании по соответствующему значению указателя адресу элемента данных с типом и параметрами, соответствующими базовому типу указателя. Этот случай можно обобщить, учтя возможность существования метаинформации о других объектах языка. Например, из того, что машина вывода обнаружила некоторый элемент данных, может быть сделан вывод о существовании некоторого типа в некотором пространстве имён и/или о значениях некоторых характеристик этого типа.

В качестве примера метаинформации рассмотрим файл TEST.PAS (Приложение 3.1.1) и разобранный результат трансляции этого файла компилятором Borland Pascal 7.0 - файл TEST.TPU (Приложение 3.1.2). В данном файле определяются два типа данных: `TLine` и `TTypeConstRec`, и

создаётся типизированная константа ConstRec (т.е. переменная с заранее присвоенным значением):

```
const
  ConstRec: TTypeConstRec =
    (W: $A7B8; Ch: 'X'; L: $89ABCDEF; Line: 'Text Ln');
```

Под такую переменную компилятор выделяет память в специальном блоке TPU-файла, который после связывания станет составной частью сегмента данных программы. Этот блок Hdr.DebugInfoOfs^.DataInf[0] можно увидеть в самом конце файла TEST.TPU, и сейчас он отображается в виде сырых данных, в которых можно без труда опознать занесённые в ConstRec значения.

В то же время, в интервале адресов [0x0137,0x01A6) записана вся необходимая для создания типа данных TTypeConstRec информация. Например, элемент данных, названный Line_Inf, несёт в себе информацию о том, что у данной записи есть поле с именем '*Line*', это поле расположено по смещению 7 байт от начала записи, а описание типа данного поля находится в этом же файле по смещению 0x0127. Таким образом, вопрос состоит в том, чтобы "научить" машину вывода понимать такую информацию и работать с ней. Тогда из факта существования элемента данных TTypeConstRec_Inf и тех элементов данных, на которые он ссылается, машина вывода сможет сделать заключение о существовании типа данных TTypeConstRec и, например, сможет использовать этот тип, чтобы представить содержимое блока Hdr.DebugInfoOfs^.DataInf[0] не в виде сырых данных, а в более естественном виде, подобном вышеприведённому объявлению ConstRec, или, что более важно, отвечать на такие вопросы, как: "чему равно значение ConstRec.L". Для этих целей предполагается использовать механизм интерфейсов (п. 2.7).

Таким образом, при работе с описаниями форматов данных машина вывода делает заключения о существовании и свойствах одних понятий из подобной информации о других понятиях, поэтому в большинстве случаев естественным направлением рассуждений является прямой вывод с запоминанием части уже выведенной информации (о константах, переменных и типах), а, с точки зрения логической интерпретации, утверждения языка FlexT предоставляют начальную информацию для этого процесса. Однако, возможны случаи, когда требуется процесс рассуждений более сложный, чем просто прямой вывод: для интерпретации одних данных может потребоваться результат разбора других данных, до которых процесс разбора ещё не дошёл.

3.10. Использование размеченных потоков вывода.

{Этот пункт исключён из электронной версии диссертации.}

3.11. Отображение информации о перемещаемых адресах в шестнадцатеричном дампе.

Другой полезный приём, который автору не приходилось встречать в других программных системах, касается способа отображения информации о перемещаемых адресах при выводе шестнадцатеричного дампа.

Любой формат исполняемых файлов, рассчитанный на использование в современных операционных системах, должен поддерживать возможность загрузки программы в различные диапазоны адресов. При этом возникает необходимость *перемещения* ссылок на содержащиеся в программе данные и код. Как правило, это перемещение состоит в прибавлении к адресу разности между базовым адресом загрузки и базовым адресом, с которым была скомпилирована программа, хотя, могут существовать и более сложные варианты (см., например, **[30]**, **[45]**). Перемещаемыми являются большинство 32-разрядных исполняемых файлов Windows. Исключение составляет динамическая библиотека KERNEL32.DLL, поскольку она загружается первой в системные 2Gb адресов и не нуждается в перемещении, а также некоторые исполняемые файлы, сгенерированные компиляторами Microsoft, которые не экспортируют адреса.

Что касается объектных файлов, то там некоторые фрагменты данных также отмечаются как места, куда должны быть помещены ссылки на переменные или процедуры программы в процессе компоновки.

Таким образом, даже если не удалось определить, является ли некоторый интервал адресов образа программы данными некоторого определённого типа или кодом, при его выводе было бы полезно отображать информацию об имеющихся в этом интервале перемещаемых адресах, так как это может помочь определить назначение отображаемых данных. Например, если в некоторой части дампа следуют несколько перемещаемых адресов подряд, то можно предположить, что здесь расположена таблица указателей, а также сделать вывод, что эта область данных никак не может быть исполняемым кодом.

На Рис. 10 показан пример, иллюстрирующий способ отображения перемещаемых адресов 32-разрядных исполняемых файлов Windows, используемый интерпретатором FlexT в программе PE Explorer.

```
00458380:0F 0D 04 05 02 06 07 03 04 05 04 04   00458727  ¦............'ЗЕ.¦
00458390:  00458731    0045873B    00458745    00458775  ¦1ЗЕ.;ЗЕ.ЕЗЕ.uЗЕ.¦
004583A0:  0045877E    00458787    00458790    00458799  ¦~ЗЕ.ЗЗЕ.РЗЕ.ЩЗЕ.¦
004583B0:  004585AF    004585A6    004585AF  00 D9 DD DB ¦пЕЕ.жЕЕ.пЕЕ..-¦-¦
004583C0:DF 00 18 18 38 38 00 00   0045888D    00458895  ¦-...88..НИЕ.ХИЕ.¦
004583D0:  0045889D    004588A5    004588AD    004588B5  ¦ЭИЕ.еИЕ.нИЕ.¦ИЕ.¦
004583E0:  004588BD    004588C4    004588CC    004588D4  ¦-ИЕ.-ИЕ.¦ИЕ.LИЕ.¦
004583F0:  004588DC    004588E4    004588EC  00 00 00 00 ¦-ИЕ.фИЕ.ьИЕ.....¦
00458400:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ¦................¦
```

Рис. 10. Отображение перемещаемых адресов в дампе.

На этом рисунке соседние 4 байта, составляющие перемещаемый адрес, отображаются как одно 32-разрядное число в шестнадцатеричном

представлении, которое выводится по центру места, отведённого для отображения этих байтов. При этом перемещаемый адрес отображается именно как адрес, т.е. с кодом siDataAddr, поэтому, например, по нему можно переходить, как по гипертекстовой ссылке.

На Рис. 10 показан достаточно хороший случай, когда все перемещаемые ссылки расположены по адресам, кратным четырём и, поэтому, они не пересекают границы 16-байтных блоков, соответствующих строкам дампа. Более сложный случай показан на Рис. 11.

```
00426660:7D 18 FF FF FF 7F 0F 84 1F 0C 00 00 8B 15 0045D258¦}.   □Д....Л.ХТЕ.¦
      00426672:8B C7 81 C2 40 02 00 00 E8 D1 8C FE FF 8B ¦  Л¦БТ@...шТМ¦ Л¦
00426680:D5 E8 CA 8F FE FF 8B F8 89 6F 08 E9 FB 0B 00 00 ¦-ш¦П¦ Л°Йо.щv...¦
00426690:80 3D   0045E514  4F 75 33 B8 20 00 00 00 E8 AD ¦A=.хЕ.Ou3¬ ...шн¦
004266A0:E7 FF FF 8B F0 8B C6 E8 F4 DD FF FF 8B D0 B8 004566CC¦ч  ЛЁЛ¦шЇ¦  Л¦¬¦fЕ.¦
         004266B3:E8 64 DE FF FF 8B F8 A1   0045D258  05 ¦   шd¦  Л°бХТЕ..¦
004266C0:40 02 00 00 89 47 08 E9 BF 0B 00 00 8B 2D 0045D258¦@...ЙG.щ¬...Л-ХТЕ.¦
      004266D2:C7 44 24 08   004566D5  81 C5 80 02 00 00 ¦  ¦D$.-fЕ.Б+А...¦
004266E0:33 D2 8B 44 24 08 E8 31 DE FF FF 8B F8 89 6F 08 ¦3ТЛD$.ш1¦  Л°Йо.¦
```

Рис. 11. Отображение перемещаемых адресов, пересекающих 16-байтные блоки.

Видно, что при этом всё равно можно отобразить перемещаемый адрес, как единое целое, за счёт некоторого нарушения выравнивания строк дампа.

Заметим, что Рис. 10 является характерным для фрагментов данных, а Рис. 11 - для фрагментов кода программы.

3.12. Деревья и таблицы решений для индексации перечислений термов.

При работе с перечислениями термов необходимо уметь сопоставлять значению базового типа терм в самом этом типе данных, а также определять номер выбранного случая в варианте по перечислению термов (см. п. 2.5.7). В обоих случаях используется специальная структура данных - *индекс перечисления термов* (названная так по аналогии с индексами таблиц в базах данных). В вариантах используются собственные индексы, чтобы впоследствии можно было там поддержать учёт значений полей терма, а не только имён термов, как сейчас.

Будем называть маской полей терма i число MV_i, в котором биты, занятые полями, установлены в 1, а остальные биты равны нулю:

$MV_i = \bigvee_j MF_{ij}$, где MF_{ij} - маски полей, а V - побитовая операция "или".

Соответственно, маской базы терма M_i будем называть побитовое дополнение маски полей: $M_i = \neg MV_i$. Базу терма будем обозначать B_i.

По определению перечисления термов, для обнаружения терма, соответствующего некоторому значению базового типа X, можно проверять термы в порядке их объявления на совпадение этого значения с базой терма по маске базы терма: $X\&M_i = B_i\&M_i$ => терм найден. Однако, такой способ

проверки может оказаться слишком медленным. Для ускорения процесса можно воспользоваться какими-либо методами индексации.

Простейшим и наиболее быстродействующим способом индексации является таблица решений, которая представляет собой массив T, сопоставляющий значению базового типа X номер соответствующего ему терма за один шаг: i := T[X]. Для построения такой таблицы она заполняется специальным значением (-1), означающим отсутствие варианта, и затем для всех термов i в порядке объявления каждое значение X, которому ещё не сопоставлен вариант, проверяется на соответствие терму. При обнаружении соответствия, в таблицу помещается номер найденного терма: T[X] := i. В случае индекса для варианта по перечислению термов в таблицу вместо номера терма непосредственно помещается номер варианта. Однако, недостатком этого подхода является большой размер таблицы для больших размеров базового типа, поэтому в интерпретаторе FlexT этот способ используется только для размеров базового типа до 8 бит включительно.

Конструктор типов данных "перечисление термов" был реализован под влиянием работ **[29]**. Там для определения выбранного терма строится дерево решений, причём, на каждом шаге проверяется значение одного из полей, которое было выбрано из общего списка полей при построении дерева исходя из критерия оптимальности - максимального уменьшения энтропии. Под полями там понимаются не только поля термов, но и группы соседних битов, входящих в маски баз термов. Для того, чтобы не заставлять пользователя объявлять фиктивные поля, а также для того, чтобы в дереве решений можно было использовать более простые структуры данных (в подходе **[29]** деревья решений представляют собой вложенные варианты (`switch`) в сгенерированном коде на C), было решено использовать в качестве тестов в узлах дерева проверки значений отдельных битов, с возможностью последующего группирования одновариантных масок. При этом для построения дерева решений также используется энтропийный критерий и жадный алгоритм **[46]**. Удобным свойством такого подхода является также то, что при построении дерева используется только информация о масках и базах термов, но не информация о полях.

Узлы дерева решений представлены следующей записью:

```
PEnumTermNDXTreeRec = ^TEnumTermNDXTreeRec;
TEnumTermNDXTreeRec = record
  M,V: LongInt;
  Sub: array[0..1] of PEnumTermNDXTreeRec;
end ;
```

где `M` - маска базы терма, `V` - база терма, `Sub` может содержать либо указатель на поддерево, либо номер выбранного варианта (номера вариантов опознаются по попаданию в диапазон `0..<Число вариантов>-1`, поскольку адреса данных в куче никогда не бывают близкими к нулю). Для поиска выбранного варианта по значению `V` используется следующий код:

```
Node := NDXTree;
```

```
  while (LongInt(Node)<0)or(LongInt(Node)>Count) do
    Node   :=   Node^.Sub[Ord(Node^.M   and(Node^.V   xor
  V)<>0)];
  GetValTerm := LongInt(Node)-1;
```

Т.е. `Sub[0]` содержит ссылку на поддерево, соответствующее совпадению со значением по маске, а `Sub[1]` - различию.

Первоначально дерево строится, как иерархия тестов отдельного бита, при этом во всех узлах `M` содержит один ненулевой бит, а `V` равно нулю.

Для построения дерева решений по энтропийному критерию необходимо уметь определять "вес" каждого случая, т.е. число удовлетворяющих ему значений, при наличии ограничения - результата применения тестов на предыдущих узлах дерева. Текущее ограничение может быть представлено также как терм: соответствующими ему маской - набором уже проверенных битов и значением по маске.

Если терм не перекрывается частично ранее объявленными термами, то его вес при отсутствии ограничений определяется числом установленных битов в маске полей. Пусть $|X|$ обозначает число битов равных 1 в значении X, тогда вес терма i равен $P_i = 2^{|MV_i|}$. Два терма пересекаются, если существуют значения, соответствующие сразу обоим этим термам. Условием существования пересечения двух термов i и j является совпадение всех битов, которые не варьируются (т.е. не входят в маску полей) ни у одного из них: $B_i\&(M_i\&M_j) = B_j\&(M_i\&M_j)$. Если это условие выполняется, то вес пересечения равен $P_{ij} = 2^{|MV_i\&MV_j|}$, иначе вес пересечения равен нулю. Если на термы наложено ограничение C, то вес терма i будет равен P_{iC}, т.е. весу пересечения этого терма с термом, соответствующим ограничению.

Для запоминания информации о частичном перекрытии терма предшествующими термами, а также о термах, включённых в каждый из случаев варианта, на время построения дерева создаётся вспомогательная структура данных - таблица вариантов, которая представляет собой массив с элементами `PEnumTermNDXRec`, по одному на каждый случай.

```
PEnumTermNDXRec = ^TEnumTermNDXRec;
TEnumTermNDXRec = record
  Next: PEnumTermNDXRec;
  Sub: PEnumTermNDXRec; {Sub-blocks, Subtract}
  Mask,V: LongInt;
end ;
```

При помощи этой структуры и определяется вес каждого варианта.

В приложении 2.1 "Описание кодирования машинных команд PDP-11." содержится пример объявления 16-битного перечисления термов (т.к. PDP-11 - это серия 16-битных машин). Для этой спецификации дерево решений строится дважды: для самого перечисления термов `TWOpCode` и для варианта по нему `TOpPDP.Dat`. Таблица вариантов для `TOpPDP.Dat` приведена на Рис. 12, а таблица вариантов для `TWOpCode` - в приложении 2.2.

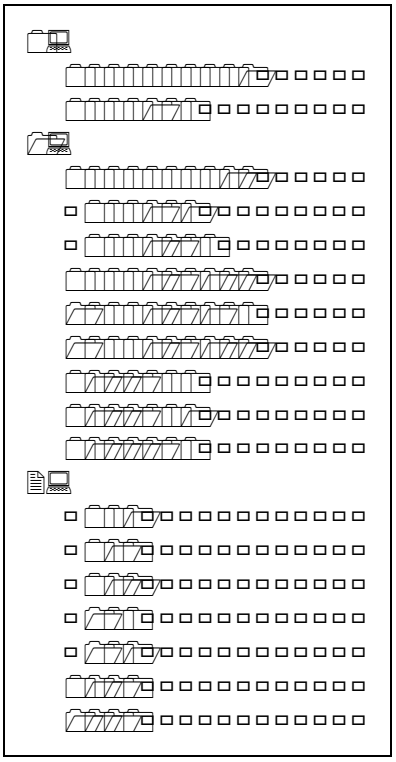


Рис. 12. Таблица вариантов для TOpPDP.Dat в спецификации системы команд PDP-11.

Заметим, что перекрытие вариантов в этой спецификации обнаружено только в одном случае: вариант №11 в приложении 2.2, соответствующий использованию команды очистки флагов CLF при отсутствии очищаемых флагов в качестве NOP. Таким образом, во всех элементах таблицы вариантов, кроме одного, поле `Sub` будет равно `Nil`.

Пусть P_{iC} обозначает теперь вес варианта i, с учётом его перекрытия другими вариантами и ограничения C. Обозначим $S_C = \sum_i P_{ic}$. Тогда энтропия набора вариантов определяется как $E_C = -\sum_i (\frac{P_{iC}}{S_C}) \ln(\frac{P_{iC}}{S_C}) = \ln(S_C) - \sum_i \frac{P_{iC} \ln(P_{iC})}{S_C}$.

Пусть C{k,v} обозначает ограничение, полученное из C добавлением условия, что бит k должен быть равен v. При поиске следующего бита для проверки выбирается тот из них, в результате тестирования которого получается минимальная средняя энтропия $E_{C\{k\}} = E_{C\{k,0\}} + E_{C\{k,1\}}$.

После построения дерева с побитовыми сравнениями его можно упростить за счёт объединения соседних ветвей, выбирающих один и тот же вариант. Деревья решений для `TOpPDP.Dat` приведены на Рис. 13: с побитовыми условиями (слева) и его сжатая версия (справа). Деревья решений, полученные для `TWOpCode`, приведены в приложении 2.3.

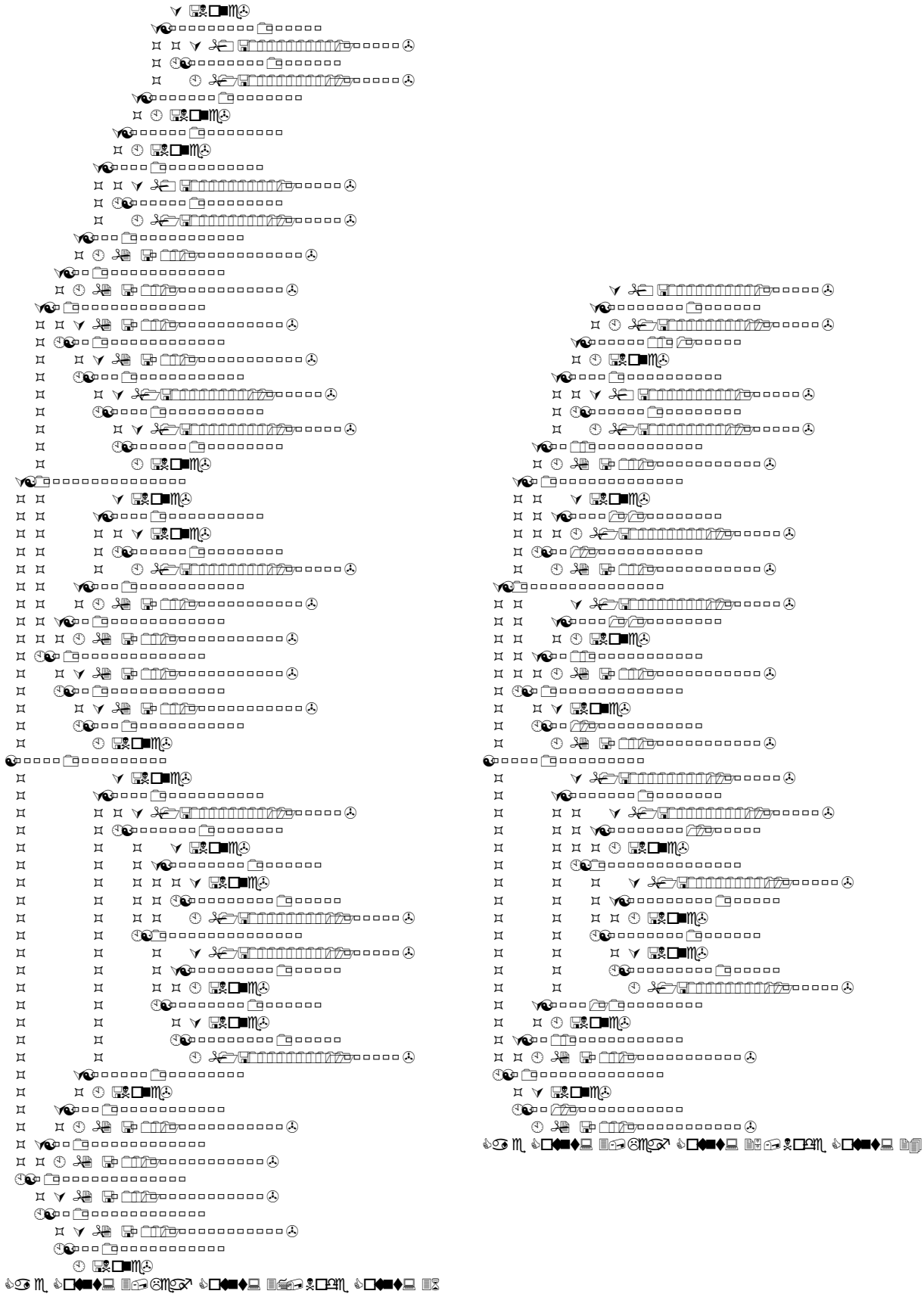


Рис. 13. Деревья решений для TOpPDP.Dat.

4. Использование языка FlexT

С использованием интерпретатора FlexT было разработано несколько программных систем:

- программа для отображения содержимого файлов произвольных форматов BinView;
- Интернет-версия программы BinView для удалённой обработки файлов - WWWBinView;
- программа для интерактивного просмотра 32-разрядных исполняемых файлов Windows (формат PE) - PE Explorer.

В этой главе будут рассмотрены особенности применения каждой из этих систем, что позволит охарактеризовать их возможности, а также примеры описания некоторых форматов данных. Описания примеров будут достаточно краткими, поскольку основная информация содержится в исходных текстах на FlexT, приведённых в приложениях.

4.1. Разбор файлов в программе BinView.

Данная программа является консольным приложением, при запуске которого в параметрах может быть задан файл для разбора и ряд других установок. За счёт этого можно, например, написать командный файл для пакетного разбора группы файлов.

Рассмотрим допустимые ключи программы BinView, при этом будут также рассмотрены и основные возможности данной программы:

Ключ	Описание
/A	разбирать весь файл(не выделять данные)
/B<Мин>-<Макс>	разбирать блок файла
/I<codepage>	целевая кодовая страница
/H, /X	вывод в шестнадцатеричных кодах (по умолчанию)
/W<Width>	ширина текста
/D+(-)[1]	включить (отменить) дамп команд
/C+(-)	включить (отменить) символьное отображение команд
/L+(-)	включить (отменить) вывод смещения команд
/O	вывод в восьмеричных кодах
/R<Path>	Путь для поиска файлов *.ref, *.rfh, и т.д.
/Z-(+)	убрать (установить) точку входа на 0
/E<ext>	разобрать, как файл с расширением <ext>
/F=[<FName>'.'] {TXT|RTF|HTM|TEX}	формат вывода результата

[1] Если после ключа стоит +(-) или -(+), то это означает, что после него можно указать один из этих символов (+ или -), причём состояние, соответствующее первому из них, установлено по умолчанию.

/P	не прослеживать указатели
/N	не показывать ссылки в HTML
/4	32-разрядный код Intel 80x86
/V	только переменные (не показывать дамп)

Таблица 3. Ключи программы BinView.

Рассмотрим возможные установки программы более подробно, объединив их в группы по назначению:

1) Определение формата файла.

/A разбирать весь файл(не выделять данные),

/B<Мин>-<Макс> разбирать блок файла в диапазоне смещений от <Мин> до <Макс>.

Если задан один из этих ключей, то весь файл или его интервал будут разобраны как последовательность машинных команд i80x86. Эта возможность оказывается полезной для дизассемблирования фрагментов кода. Была написана специальная служебная программа HEX2BIN, которая позволяет получить бинарный файл, соответствующий фрагменту шестнадцатеричного дампа. Таким образом, можно извлечь из результатов разбора фрагмент и дизассемблировать его, даже если формат не описан до уровня виртуальных блоков.

/E<ext> разобрать как файл с расширением <ext>

Этот ключ позволяет вмешаться в логику определения формата файла, которая учитывает расширение файла (см. п. 3.8). При задании этого ключа файл будет разобран так, как если бы он имел расширение <ext>. В <ext> не включается точка.

2) Способ разбора.

/4 32-разрядный код Intel 80x86

Так как в данной программе по умолчанию предполагается, что код i80x86 является 16-разрядным, необходимо указать этот ключ для разбора файла с 32-разрядным кодом.

/Z-(+) установить точку входа на 0

В некоторых файлах, содержащих исполняемый код, точка входа находится в начале этого файла. В таком случае можно использовать этот флаг. Вместо этого можно было бы создать файл с "личным" описанием (*.ref, п. 3.8).

/P не прослеживать указатели

Если установить этот флаг, то файлы, относящиеся к форматам, не использующим указатели, будут разбираться немного быстрее, так как этап прослеживания указателей (п. 3.8.1) будет пропущен.

3) Способ отображения.

Система счисления:

/H, /X вывод в шестнадцатеричных кодах (по умолчанию)
/O вывод в восьмеричных кодах
Отображение машинных команд:
/D+(-) включить (отменить) дамп команд
/C+(-) включить (отменить) символьное отображение команд
/L+(-) включить (отменить) вывод смещения команд
Весь результат:
/I<codepage> целевая кодовая страница
/W<Width> ширина текста
/N не показывать ссылки в HTML

Эта настройка может сильно уменьшить размер результата разбора при выводе в формат HTML.

/V только переменные (не показывать дамп)

По умолчанию при отображении результатов разбора те области памяти, которые не вошли в состав обнаруженных переменных и частей кода, отображаются как шестнадцатеричный дамп. Если ключ /V задан, то эти области отображаться не будут.

4) Поиск и размещение файлов.

/F=[<FName>'.']{TXT|RTF|HTM|TEX} формат вывода результата и путь файла результата.

Если этот ключ не задан, то результат выводится в стандартный вывод в виде текста.

Если задано только расширение, например, /F=RTF, то результат выводится в стандартный вывод в соответствующем расширению формате.

Если в качестве <FName> задан символ '*', то результат выводится в файл, путь которого отличается от разбираемого файла только расширением. Пример: для вызова BinView /F=*.rtf c:\test\x.dcu результат будет записан в файл c:\test\x.rtf.

Иначе результат выводится в файл с заданным после /F= путём, причём, если в этом пути содержится символ '*', то он заменяется на имя разбираемого файла. Пример: для вызова BinView /F=d:\res*.rtf c:\test\x.dcu результат будет записан в файл d:\res\x.rtf. Этот режим удобно использовать для пакетной обработки файлов, записанных на компакт-диск.

/R<Path> Путь для поиска файлов *.ref, *.rfh, *.rfi, *.cfg

4.2. Удалённая обработка файлов в программе WWWBinView.

Для демонстрации возможностей интерпретатора FlexT была реализована Интернет-версия программы BinView - WWWBinView. На Рис. 14 показана

форма для выбора файла и задания параметров для разбора. Выбранный файл передаётся на сервер, разбирается и результат разбора возвращается пользователю. Переданный файл кэшируется, поэтому его можно разобрать повторно с другими установками.

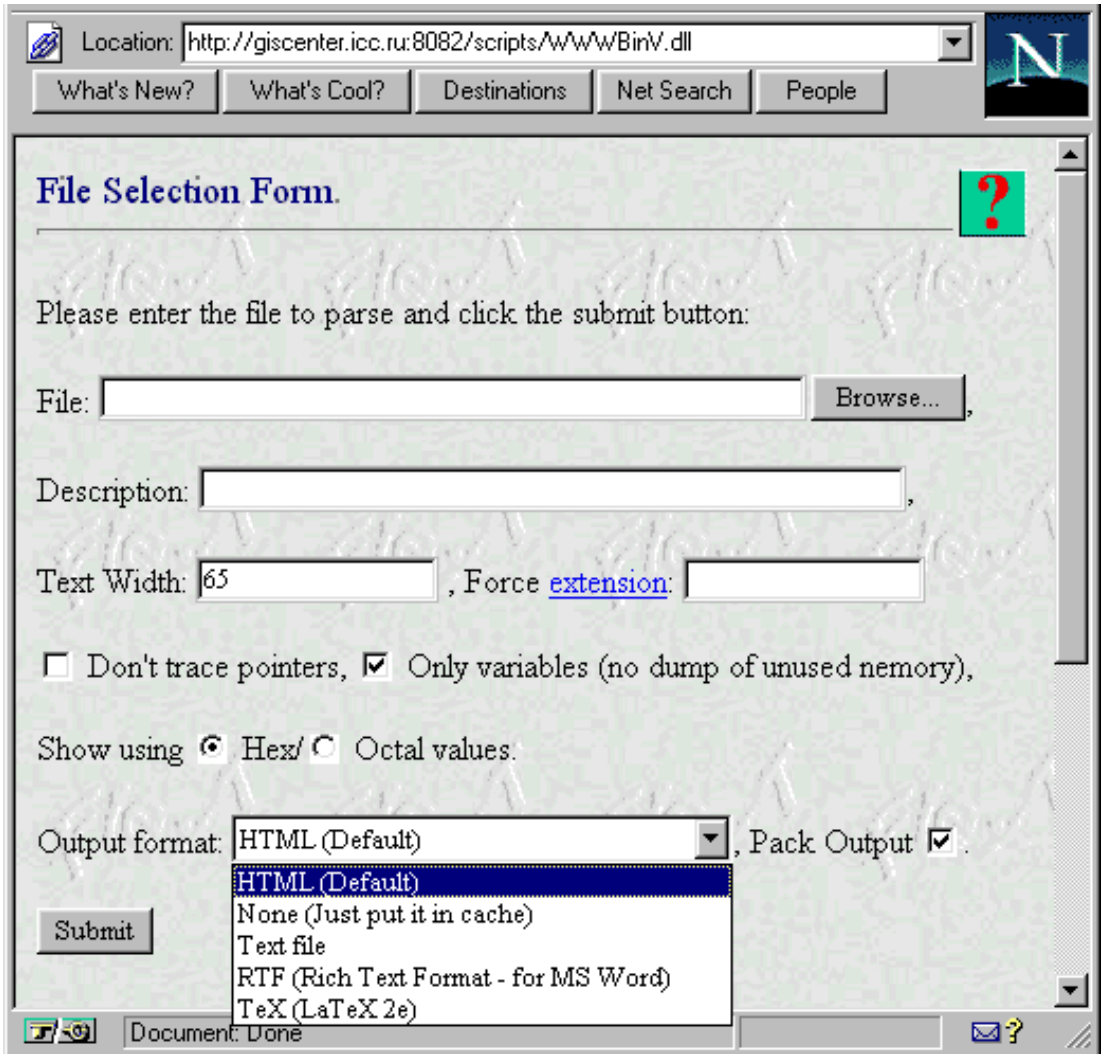


Рис. 14. Форма запроса для выбора файла и определения параметров для разбора.

Описание полей формы запроса:

Название	Назначение	Ключ BinView
File	Поле для выбора файла.	
Description	Описание - краткое сообщение, которое будет запомнено на сервере.	
Text width	Ширина результата разбора в символах.	/W
Force extension	Разбирать, как файл с этим расширением. Перейдя по ссылке можно получить таблицу известных системе расширений.	/E
Don't trace pointers	Не проводить прослеживание указателей.	/P
Only variables	Не показывать дамп неразобранной памяти.	/V

Hex/Octal values	Система счисления.	/X, /O
Output format	Формат вывода результата	/F
Pack output	Передавать результат в упакованном GZIP виде (при этом в заголовок HTTP будет добавлено поле `Content-Encoding: gzip`, поэтому броузер автоматически его распакует).	

Таблица 4. Поля формы запроса для WWWBinView.

Кроме собственно разбора файлов, данная программа поддерживает запросы для получения таблицы опубликованных спецификаций (Рис. 15) и таблицы известных системе расширений (Рис. 16). Для того, чтобы можно было выбирать, какие файлы спецификаций будут доступны для просмотра через Интернет, а также, чтобы дать этим файлам дополнительную характеристику, существует специальный файл настройки WWW.CFG. Текущая версия этого файла приведена в приложении 1.2. Доступными для просмотра являются только те файлы спецификаций, которые упомянуты в WWW.CFG.

Catalog of **FlexT** format specifications:

File	Description	Class	Status	Date
ANI.RFH	Windows animated cursors	Graphics and Sounds	Partial	05.07.1999 19:43:02
ARJ.rfh	ARJ Archive Format	Archive	Headers only	20.08.1999 13:02:24
BGI.RFI	Borland Graphics Interface	Executable and Object	Auxiliary	12.11.1998 14:32:40
BMP.RFH	Windows Bitmap file	Graphics and Sounds	Complete	26.01.1999 16:00:06
BMP.RFI	Bitmap file data structures	Graphics and Sounds	Complete	26.01.1999 17:31:08

Рис. 15. Отображение таблицы доступных для просмотра спецификаций.

Extension	Possible formats
.	elf.rfh
ANI	ANI.RFH
ARJ	ARJ.rfh
BMP	BMP.RFH
cla	cla.rfh
class	cla.rfh
cur	ico.rfh
dat	dat.rfi clarion.rfi
db	db.rfh
DBF	DBF.RFH
dcu	tpu.rfi dcu32.rfi
DFM	DFM.RFH
dll	exe.rfi

Рис. 16. Отображение таблицы известных системе расширений.

```
    29:(Tag:drStrucScope{'T'};
      D:(Sz:#01;
        D: (
          0:(hType:#09; hVar:#00; Ofs:#00; LnStart:#0B; LnCnt:#18)))),
    30:(Tag:drSymbolRef{'У'}; D:(Sz:#00; NPrimary:#00; D: ())),
    31:'Ф'))
End of CODE.

To parse this file were used the following specifications:
tpu.rfi
  DOSFTime.rfi
dcu32.rfi
  DOSFTime.rfi

Other specifications.
```

Рис. 17. Фрагмент результата разбора файла - ссылки на использованные спецификации.

При отображении результатов разбора файла в формате HTML в этот результат включаются ссылки на те из использованных спецификаций, которые были описаны в файле WWW.CFG (Рис. 17).

4.3. Просмотр 32-разрядных файлов Windows в программе PE Explorer.

С использованием интерпретатора FlexT была разработана программа PE Explorer для просмотра 32-разрядных исполняемых файлов Windows (формат PE). Эта программа, в отличие от BinView, с одной стороны, является

интерактивной, а, с другой стороны, ориентирована на работу с конкретным форматом файлов - Portable Executable (COFF) **[24]**, **[25]**.

В качестве образца дизайна интерфейса (см. Рис. 18) для программы был взят интерфейс примера RESXPLOR, входящего в комплект стандартной поставки всех 32-разрядных версий Delphi, который, в свою очередь, отображает ресурсы файла в стиле Microsoft Windows Explorer.

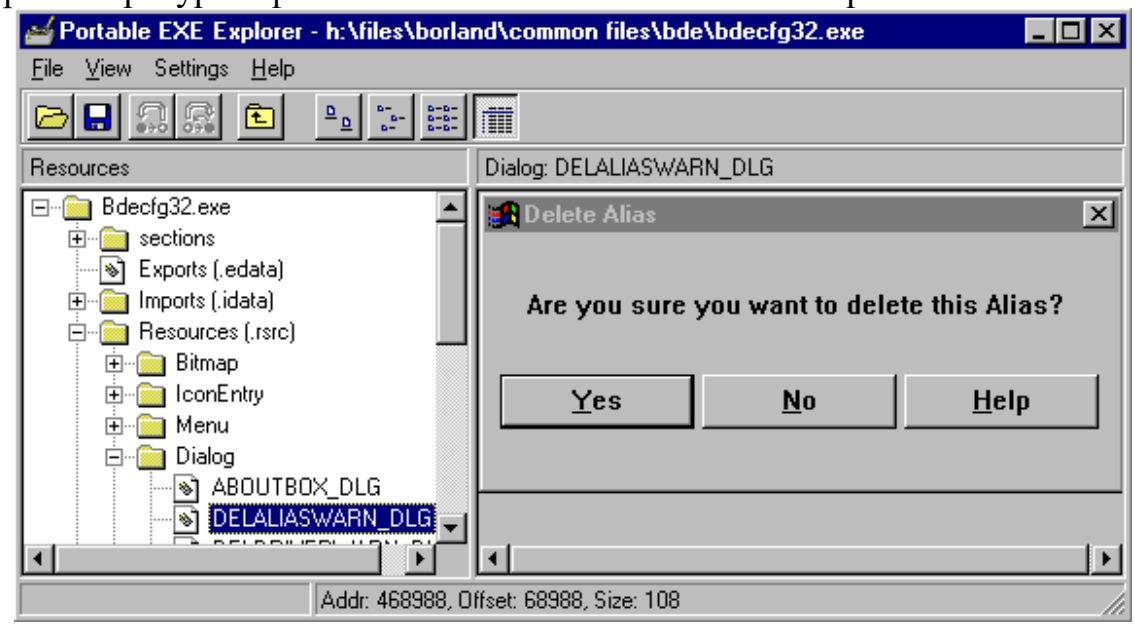


Рис. 18. Просмотр ресурса диалога.

Возможности данного примера были расширены, как по видам поддерживаемых ресурсов, так и, что более важно, по возможностям просмотра других частей исполняемого файла, помимо ресурсов.

Код и данные файлов в формате PE рассчитаны на загрузку программы в 32-битном Flat-режиме и разделены на ряд секций. Все эти секции в результате загрузки отображаются в одно 32-разрядное адресное пространство. Для интерпретатора FlexT каждая секция является отдельным блоком кода. При отображении содержимого файла в дереве, находящемся слева в окне программы, можно выбрать интересующую секцию, при этом справа отображается её содержимое, т.е. результат разбора данных этой секции интерпретатором FlexT. Таким образом, разбираются только отображаемые в виртуальное адресное пространство секции, а не всё содержимое файла.

Без дополнительной информации разбор содержимого файла заключается в дизассемблировании некоторой, как правило небольшой, части кода программы, начиная от известных точек входа: начального адреса для исполнения, записанного в заголовке файла, а также адресов экспортируемых функций. Однако, многие исполняемые файлы, например, файлы, созданные компилятором Delphi, содержат ряд структур данных, после описания которых удаётся определить адреса наиболее интересных частей кода программы (см. п. 4.6). Кроме того, возможность структурированного отображения данных программы имеет и самостоятельную ценность. Также оказывается полезным

для понимания разбираемого кода отображение рядом со ссылками на переменные имён и квалификаторов их составляющих.

Для того, чтобы сообщить системе дополнительную информацию о разбираемом файле используется тот же подход, что и в программе BinView: создаётся "личное" описание файла с тем же именем и расширением `.ref`. Этот файл может размещаться в том же каталоге, что и описываемый файл, либо в каталогах пути поиска спецификаций.

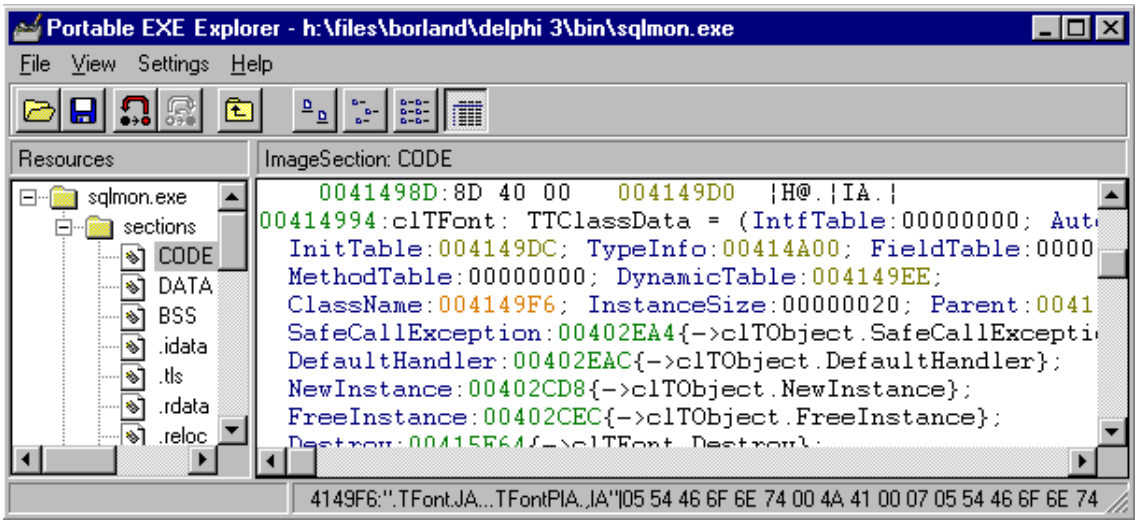


Рис. 19. Просмотр VMT для класса Delphi TFont.

Для того, чтобы предотвратить разбор другого файла с тем же именем, рекомендуется использовать следующую директиву условной компиляции:

```
%$IF FileSize=<Размер файла в байтах>;
  <Описание структур данных и точек входа файла>
%$END
```

Константа `FileSize` объявляется автоматически после загрузки файла и ей присваивается значение размера файла в байтах. Конечно, таким способом можно проверять и другие характерные для описываемого файла данные, чтобы "удостоверить его личность".

4.4. Описание формата файла класса Java-машины.

В приложении 3.2.1 приведено описание формата файла класса Java-машины (далее CLASS), а в приложении 3.2.2 - описание кодирования машинных команд Java-машины, которое используется в спецификации формата CLASS. По своей роли этот формат относится к форматам исполняемых файлов, т.е. для Java-машины он играет ту же роль, что и, например, формат EXE для DOS. Информация о данном формате получена из **[23]**. В каждом файле класса содержится информация об одном классе Java, причём, к этой информации относятся такие детали, как имена полей или методов класса, их типы и т.д., поэтому по своей структуре данный формат является, скорее, форматом объектных файлов, содержащим информацию о

типах, которая отсутствует в обычных исполняемых файлах, и предназначенным для обработки редактором связей перед исполнением.

С одной стороны, в описании данного формата используется достаточно много различных конструкций языка FlexT, поэтому его изучение позволяет лучше разобраться в особенностях использования этого языка. С другой стороны, рассматриваемое описание формата было получено, практически, за один проход чтения спецификации формата **[23]**, т.е. по ходу чтения в спецификацию добавлялись новые определения и иногда её корректность проверялась на реальных данных. Также интересно, что кодирование машинных команд Java-машины хорошо описывается, как обычный тип данных - теговая запись.

В рассматриваемом формате не используются указатели, и всё содержимое файла класса может быть представлено одной большой записью `TClassFile`. Для удобства проверки принадлежности к формату из этой записи в отдельную переменную выделено первое поле `magic`, которое содержит сигнатуру формата. После полей `minor_version` и `major_version` в записи `TClassFile` находится таблица констант, которая является массивом переменной длины. В поле `C_pool_count` записано количество констант, а за ним следует сама таблица `C_pool`, причём, номер константы 0 используется в качестве Nil и в таблице не хранится, поэтому она имеет `C_pool_count-1` индексов, а из значения индекса также вычитается 1 при обращении к этой таблице. Некоторые элементы таблицы констант (с тегами `C_Long` и `C_Double`) занимают два индекса, поэтому в определении массива используется блок числа индексов `TAKES`. Эта особенность формата возникла из желания хранить 64-битные значения непосредственно в таблице из 32-битных элементов и впоследствии была признана неудачным решением, но при описании форматов необходимо иметь возможность описывать и такие решения.

Элементы таблицы констант описываются типом `cp_info` и могут содержать различные данные, в зависимости от тега с типом `TCPInfoTag`. Данные, зависящие от тега, представлены в записи `cp_info` полем `info`, которое является вариантом. В некоторых случаях данные константы содержат индекс другой константы из таблицы констант. Так, для тега `C_InterfaceMethodref`, данные константы представлены записью `TRefInfo`, которая содержит два поля с типами `TClassNDX` и `TNameTypeNDX`. Тип `TClassNDX`, является индексом константы, которая должна иметь тег `C_Class`, а данные этой константы являются, в свою очередь, индексом строковой константы `TUtf8NDX`. Таким образом, чтобы получить имя интерфейса, соответствующего константе с тегом `C_InterfaceMethodref`, потребуется ещё два раза обратиться к массиву констант. Чтобы описать такие зависимости, в рассматриваемой спецификации используется переопределение типов индексов констант (`TUtf8NDX`, `TClassNDX`, `TNameTypeNDX`) при помощи отображения выражением

(п. 2.5.4.2). Чтобы можно было использовать переопределение, эти типы первоначально объявляются, как вызовы, а не как переименования типа `u2`, на что указывают круглые скобки после `u2`. В результате этих определений становится известен размер типов индексов, и все определения типов, использующие индексные поля, становятся завершёнными. Далее, после объявления переменной `Hdr` в блоке `data`, соответствующей всему содержимому файла, становится возможным использовать в выражениях для отображения индексов ссылки на эту переменную, чтобы получить доступ к массиву констант, например,

```
TUtf8NDX enum TUtf8NDX Hdr.C_pool[@-1].info.C_Utf8.val;
```

Также стоит обратить внимание, на использование квалификатора выбранного случая (в этом примере - `.C_Utf8`) для ссылки на ожидаемый вариант данных константы.

В файле класса строки представлены в кодировке UTF8 **[28]**, которая использует для представления различных символов разное число байтов, причём, для символов основной таблицы (код меньше 0x80) используется 1 байт с ASCII кодом символа, поэтому то, что в определении типа `TUtf8Str` строка представлена, как `array of Char`, оказывается полностью справедливым для идентификаторов с латинскими символами. В дальнейшем возможно более точное описание таких строк после реализации символьных типов с произвольными базовыми типами (п. 2.5.3). Игнорирование кодировки UTF8 не приводит к ошибке в определении размера строковой константы, поскольку поле `len` задаёт размер, а не число символов.

После массива констант и ряда полей - индексов, описывающих свойства определяемого класса, следуют три таблицы переменной длины: `interfaces` - таблица индексов интерфейсов, `fields` - описание полей класса, `methods` - описание методов класса и `attributes` - таблица атрибутов класса. В свою очередь, и данные поля `field_info`, и данные метода `method_info` также содержат таблицу атрибутов.

Тип элемента таблица атрибутов `attribute_info` предварительно объявляется при помощи ключевого слова `forward`. Впоследствии он полностью определяется с использованием информации, содержащейся в других элементах данных файла. Запись `attribute_info` также является теговой, только здесь разработчиками формата было принято достаточно нестандартное решение: использовать в качестве тегов индексы строковых констант с определёнными значениями. Поэтому, специфические для значения тега данные (поле `info`) описываются в спецификации при помощи варианта по строковым значениям.

Поскольку в формате CLASS учитывается возможность существования неизвестных системе тегов, запись `attribute_info` содержит информацию о размере специфических данных в поле `len`. Чтобы сообщить интерпретатору FlexT, что поле `len` представляет размер поля `info`, в спецификации используется блок утверждений. Заметим, что при этом действуют

автоматически добавляемые правила для вычисления размера составляющих, а именно - для задания размера данных, соответствующих случаю варианта, если известен размер всего варианта.

Данные записей `attribute_info`, могут содержать очень важную информацию, в частности, именно здесь находятся машинные команды (тег `'Code'`). Машинные команды, соответствующие коду некоторого метода размещаются в одном из атрибутов этого метода. Поскольку все константы программы сосредоточены в массиве констант, в блоке кода могут находиться только команды, но не данные. Поэтому всё содержимое такого блока можно представить, как массив из записей, описывающих кодирование машинных команд - этот массив задан типом `TOpSeries`, определённым в файле `java_cod.rfi` (приложение 3.2.2).

При описании кодирования машинных команд сначала объявляется перечисление `TOpKind`, сопоставляющее мнемонические имена команд их кодам, затем определяются ряд структур данных, соответствующих аргументам различных команд, после чего определяется кодирование одной машинной команды в записи `TOp`. Стоит обратить внимание на записи `TLookupSwitchRec` и `TTableSwitchRec`: в обоих из них содержатся таблицы для реализации оператора `switch`, которые должны быть выровнены на кратный четырём байтам от начала блока кода адрес. Для описания выравнивания служит тип `TPadZeroes`, определённый при помощи конструктора `align`. Последним в файле находится определение типа `TOpSeries`, в блоке отображения (п. 2.2.4.3) которого используется команда `ShowArray`, чтобы сообщить, что каждая команда отображается на новой строке, а перед её отображением выводится смещение от начала блока.

Пример простейшей программы на языке Java приведён в приложении 3.2.3, а результат разбора файла класса, полученного в результате компиляции этого примера, - в приложении 3.2.4.

4.5. Описание кодирования машинных команд процессора Z-80.

В приложении 3.3.1 приведено описание кодирования машинных команд процессора Z-80 (а также Intel 8080 и К580ИК80 **[32]** - это, практически, одна система команд). Рассматриваемая система команд является одной из простейших, тем не менее, для её описания необходимо использовать перечисление термов.

Поскольку в качестве типов полей термов могут использоваться только типы с битовым размещением, первоначально в битовом режиме определяются сначала целочисленные битовые типы, а затем перечисления на их основе. Далее объявляется двухбайтовый указатель на последовательность команд `POpZ80Seq`, при этом происходит и неявное предварительное объявление типа `TOpZ80Seq`. После этого следует объявление перечисления термов `TByteOpCode`, в котором описывается способ разделения первого байта

команды на битовые поля в зависимости от значения этого байта. За этим определением объявляется собственно тип команды TOpZ80, в котором для выбора вида дополнительных аргументов команд (двух- и трёхбайтовых) используется вариант по перечислению термов, после чего при помощи конструктора CODES определяется тип последовательности команд TOpZ80Seq, который описывает команды с безвозвратной передачей управления (ret, jmp, hlt, pchl). Для команд перехода в TOpZ80 аргументом является POpZ80Seq, т.е. указатель на тип CODES, за счёт этого дизассемблер определяет, что из этих команд происходит передача управления на соответствующие их аргументам адреса.

В приложении 3.3.2 приведён фрагмент (одна часть кода) результата дизассемблирования с использованием приведённой спецификации 64Kb образа памяти бытового компьютера "Львів", использующего рассматриваемый процессор. Для задания информации о файле образа памяти используется файл с его "личным" описанием, содержащий, в том числе, и следующие строки:

```
 include cmd_z80.rfi
.....................
 code(TOpZ80Seq)
.....................
 0xDCC6 BSAVE
 0xDCF5 BLOAD
 0xDB30 ReadFName
 0xDBA8 ReadCmdLine
.....................
```

В приведённом фрагменте видно, что в результатах дизассемблирования можно успешно прочитать мнемонику закодированных команд, хотя, для лучшего соответствия принятому ассемблерному синтаксису требуется добавить блок отображения в определение TOpZ80, а, чтобы можно было описывать отображение таких типов, как перечисление термов, требуется ввести дополнительные команды условного отображения.

4.6. Использование описания RTTI Delphi при дизассемблировании программ, написанных на Delphi.

В приложении 4.1 приводится описание на FlexT RTTI Delphi 3.0. Аналогичные описания существуют и для других версий Delphi.

Все ресурсы форм помещаются компилятором Delphi в секцию ресурсов RCData, под именем UpperCase(<Имя класса формы>). Программа PE Explorer отображает такие ресурсы в виде текста, т.е. так, как это делается в редакторе Delphi по команде "View as text" из всплывающего меню формы. Таким образом, в результате просмотра секции ресурсов RCData можно понять, какие классы форм существуют в данном приложении. Эти классы представляют особый интерес, так как обращения к прикладному коду (в отличие от системного кода) происходят в основном из обработчиков событий

таких классов, а определение адресов RTTI классов форм приводит к автоматическому обнаружению этих обработчиков событий, а также RTTI классов используемых компонентов, что позволяет дизассемблировать значительную часть прикладного кода и получить информацию об используемых структурах данных.

Для обнаружения RTTI классов форм можно выполнить следующие шаги:

1. Выбрать интересующий ресурс формы и скопировать его имя.
2. Открыть секцию "CODE" и переместиться в начало текста.
3. Вызвать диалог для поиска и искать в данных последовательность байтов 0x07 (TTypeInfo.Kind=tkClass), <Длина имени класса формы>, <Имя класса формы>, что соответствует первым двум полям записи `TTypeInfo`;. Можно также просто искать <Имя класса формы>, но тогда могут быть найдены и другие совпадения.
4. Убедиться, что найденный адрес является единственным, где находятся искомые данные, и добавить переменную с найденным адресом (адресом байта, со значением 0x7) и типом TTypeInfo в блок DATA файла описания (*.ref).

В результате файл описания может иметь вид:

```
%$IF FileSize=<Размер файла в байтах>;
include delphi40.rfh //Версию Delphi надо определить
data
0x<Найденный VA> TTypeInfo <Type name>_Info
%$END
```

Пример такого файла приводится в приложении 4.2.

4.7. Реконструкция формата 32-разрядных DCU.

С использованием программы BinView был реконструирован формат 32-разрядных объектных файлов Delphi (DCU). Это стало возможным благодаря лёгкости проверки спецификаций на реальных данных и наличию генератора файлов с известной семантикой (компилятора Delphi).

Тело цикла восстановления формата можно разбить на следующие этапы:

1) анализ результатов разбора тестовых файлов;
2) формирование гипотезы о виде некоторой структуры данных или её уточнение;
3) генерация тестовых файлов для проверки гипотезы;
4) уточнение спецификации на FlexT; .
5) разбор тестовых файлов.

Ключевыми моментами для понимания структуры формата DCU32 были:

- понимание того, что файл состоит из небольшого заголовка, за которым следует поток теговых записей;
- расшифровка способа кодирования целых чисел в типе данных "индекс";
- восстановление способа нумерации имён и типов.

Для представления целых чисел в рассматриваемом формате используется структура данных переменной длины, которая по аналогии с подобной

структурой данных для формата OBJ **[38]** была названа "индексом". На Рис. 20 приведено определение этого типа на языке FlexT (тип TNDXB1). Такой способ кодирования целочисленных значений позволяет использовать для представления небольших чисел (которые составляют большинство) небольшое число байт. В приведённом определении типа за счёт использования блоков отображения удаётся добиться, чтобы представление числа индексом отображалось, как '#'HEX(<Значение>). К сожалению, поскольку в языке пока не реализованы интерфейсы или определяемые пользователем функции, для того, чтобы обратиться к значению индекса из выражений, приходится использовать следующий код:

```
TCodeBlockInf struc pas
  Sz: TNDXB1
  D: raw[(@.Sz.V.0)exc((@.Sz.V.1.V.0)exc
      ((@.Sz.V.1.V.1.V.0)exc((@.Sz.V.1.V.1.V.1.V.0)exc
      (@.Sz.V.1.V.1.V.1.V.1.V.0))))]
ends
```

(подчёркнуто), который повторяется в описании формата несколько раз.

Активное использование типа "индекс" в формате DCU32, являлось основным препятствием при его реконструкции, так как из-за этого ни одна структура не имеет постоянного размера. Предположение о том, что некоторое поле является индексом, можно было сделать, если это поле всегда принимало чётные значения на небольших модулях. Чтобы проверить это предположение, необходимо было сгенерировать модуль, содержащий большое количество нумеруемых объектов, и убедиться, что при этом появляются и двухбайтные значения рассматриваемого поля.

```
type bit
  TBit num+(1)
 ..................
  TBit32 num+(32)
  TNDXB1 struc pas
    IsW: TBit
    V: case @.IsW of
     0: TBit7
     else struc pas
       IsB3: TBit
       V: case @.IsB3 of
        0: TBit14
        else struc pas
          IsB4: TBit
          V: case @.IsB4 of
           0: TBit21
           else struc pas
             IsB5: TBit
             V: case @.IsB5 of
              0: TBit28
              else struc pas
                Z: TBit4
                V: case @.Z of
                  0: TBit32
                 else struc pas //Delphi 4.0 - 64 bit
                    Lo: TBit32
                    Hi: TBit32
                  ends: displ=(HEX(@.Hi),HEX(@.Lo))
                 endc: displ=((@.0)exc(@.1))
              ends
              endc:displ=((@.0)exc(@.1))
           ends
           endc:displ=((@.0)exc(@.1.V))
        ends
        endc:displ=((@.0)exc(@.1.V))
     ends
     endc:displ=((@.0)exc(@.1.V))
  ends:displ=('#',@.V)
```

Рис. 20. Определение типа "индекс" из формата DCU32 на FlexT.

В результате этой работы над форматом DCU32 была написана программа DCU32INT для отображения содержимого файлов в близком к синтаксису Паскаля виде. Подробности см. на странице [47], там же можно найти ссылку на полученную спецификацию формата DCU32.

При описании рассматриваемого формата проявились ограничения текущей версии языка FlexT. В процессе чтения файла в формате DCU32 компилятором Delphi все объявляемые и импортируемые из других модулей имена помещаются в таблицу имён, кроме того, все определяемые и импортируемые из других модулей типы помещаются в таблицу типов, после чего для ссылок на имена и типы используются их порядковые номера в соответствующих таблицах. Сейчас в языке не поддерживается возможность описания подобных таблиц, поэтому работать с ними пришлось уже в специализированной программе DCU32INT. Впоследствии предполагается поддержать в языке описание подобных структур данных за счёт реализации интерфейса таблицы. Заметим, что подобные ограничения выходят за пределы задачи идентификации типов данных.

Заключение

Выполненная работа посвящена разработке методов и средств спецификации бинарных форматов данных. В рамках диссертации получены и на защиту выносятся следующие результаты:

1. Разработан язык описания форматов данных FlexT, который позволяет автоматизировать обработку широкого круга бинарных форматов данных, включая форматы исполняемых и объектных файлов, а также является эффективным средством описания форматов, позволяющим существенно повысить достоверность этих описаний за счёт возможности их проверки на реальных данных.
2. Разработан интерпретатор языка FlexT и несколько программных систем, использующих этот интерпретатор: программа просмотра файлов произвольных форматов BinView, программа просмотра/дизассемблер исполняемых файлов Windows (формат Portable Executable - COFF) PE Explorer, а также ISAPI расширение IIS - WWWBinView для удалённой обработки файлов (подробности - на странице FlexT http://monster.icc.ru/~alex/FlexT).
3. На языке FlexT описано несколько десятков различных форматов файлов. В основном это форматы исполняемых и других файлов, содержащих программный код, что объясняется областью интересов автора. В их числе форматы файлов со следующими расширениями: EXE (MZ,NE,LE), ELF, CLA, TPU, OBJ, BGI, TTF, WAV, BMP, DBF. Конструкции языка использовались при декомпиляции различных программ и описании характерных для конкретных компиляторов форматов данных, например, RTTI Delphi.
4. На примере объектных файлов 32-разрядных версий Delphi (формат DCU) продемонстрирована возможность использования спецификаций на языке FlexT для реконструкции неизвестного формата данных при наличии генератора данных этого формата с известной семантикой. Это стало возможным благодаря лёгкости проверки спецификации формата на FlexT на реальных данных.

Перспективными направлениями развития языка FlexT являются: реализация в нём механизма интерфейсов, дополнение языка средствами спецификации семантики машинных команд, разработка механизмов определения спецификаций редактирования, построение автоматических генераторов кодов на различных языках программирования для обработки форматов данных по их спецификациям, расширение возможностей программы PE Explorer для работы с произвольными форматами, оснащение её средствами взаимодействия с подключаемыми модулями (plug-in) для отображения/обработки данных, предоставляющих конкретные интерфейсы.

Литература.

[1] F. Faase. BFF: A grammar for Binary File Formats, http://wwwis.cs.utwente.nl:8080/~faase/BFF, 1996.

[2] F. Faase. Binary File Format Definition, http://www.wotsit.org, 1996.

[3] Cr. Cifuentes, M. Van Emmerik et all. UQBT - A Resourceable and Retargetable Binary Translator, http://www.csee.uq.edu.au/csm/uqbt.html, 1999.

[4] Cr. Cifuentes, M. Van Emmerik et all. Binary File Format language, http://www.csee.uq.edu.au/csm/srl/table.htm, 1999.

[5] Д. Цикритзис, Ф. Лоховски. Модели данных: Пер. с англ., М.: "Финансы и статистика", 1985.

[6] Г. Борн. Форматы данных, К.: "Торгово-издательское бюро BHV", 1995.

[7] General Format of .dbf files in Xbase languages, http://www.wotsit.org, 1996.

[8] А.Е. Хмельнов. Язык описания данных FlexT: гибкие типы для описания статических данных // Сб. Третий Международный симпозиум "Интеллектуальные системы".- М.: ООО "ТВК", 1998, С. 150-154.

[9] A. Hmelnov, S.Vassilyev. Data description language FlexT: flexible types for description of static data // In: Proc. of The 3rd IMACS International Multiconference on: Circuits, Systems, Communications and Computers (CSCC'99), 1999, P. 1371-1376.

[10] A. Hmelnov, S.Vassilyev. Data description language FlexT: flexible types for description of static data // In: Software and Hardware Engineering for the 21th Century (N. Mastorakis, ed.), WSES Press, 1999, P. 146-151.

[11] А.Е. Хмельнов. Язык описания данных FlexT: использование динамических типов для описания статических данных // Сб. Оптимизация, Управление, Интеллект, №4. - Иркутск: ИДСТУ СО РАН, 2000, С. 148-166.

[12] В.Н.Агафонов. Спецификация программ: понятийные средства и их организация, Новосибирск:Наука. Сиб. отд-ние, 1990.

[13] А.Тей и др. Логический подход к искусственному интеллекту: от классической логики к логическому прграммированию: Пер. с франц., М.: "Мир", 1993.

[14] Б.Лисков, Дж.Гатэг. Использование абстракций и спецификаций в разработке программ: Пер. с англ., М.: "Мир", 1989.

[15] Borland Intl., Inc. Borland Delphi 3.0. run image, файл `.\SOURCE\VCL\TypeInfo.pas`, 1997.

[16] Т.Сван. Форматы файлов Windows: Пер. с англ. - М.:Бином, 1994.

[17] K.Mitchell. PARADOX 4.x file formats, http://www.wotsit.org, 1998.

[18] Randy Beck. The PARADOX File Structure, http://www.wotsit.org, 1996.

[19] Red Hat Software. The Red Hat Package Manager, файл `./docs/format`, ftp://ftp.rpm.org/pub/rpm, http://www.rpm.org, 1998.

[20] Clarion BBS. Bulletin #117, Clarion data files, http://www.wotsit.org, 1991.

[21] Borland Intl., Inc. Open Architecture Handbook - The Borland Developer's Technical Guide, http://www.wotsit.org, 1993.

[22] W.Wouters, Windows Bitmap File Format Specifications, http://www.wotsit.org, 1997.

[23] T.Lindholm, F.Yellin. The Java Virtual Machine Specification. Addison-Wesley: The Java Series, (http://java.sun.com/docs/books/vmspec/index.html, ftp://ftp.javasoft.com/docs/specs/vmspec.html.zip), 1996.

[24] M.J.O'Leary. Portable Executable Format. Документ Microsoft Developer Support, PE.TXT.

[25] Microsoft Corp. Microsoft Portable Executable and Common Object File Format Specification, MSDN Library, http://www.microsoft.com/msdn/, 1997.

[26] LX - Linear eXecutable Module Format Description, http://www.wotsit.org, 1992.

[27] Linear Executable Format, http://www.wotsit.org, 1993.

[28] Unicode, Inc. The Unicode Standard, Version 3.0, http://www.unicode.org, 1999.

[29] N.Ramsey, M.Fernández. The New Jersey Machine-Code Toolkit, http://www.eecs.harvard.edu/~nr, 1997.

[30] Intel. Executable and Linkable Format, http://www.wotsit.org, 1993.

[31] Netscape Communications Corp. Client-Side JavaScript Reference, http://developer.netscape.com/docs/manuals/js/client/jsref/, 1999.

[32] Н.П.Брусенцов. Микрокомпьютеры, М.: "Наука", 1985.

[33] Borland Intl., Inc. Object Pascal Language Guide, 1997.

[34] R. Jung. ARJ Technical Information, http://www.wotsit.org, 1993.

[35] TrueType format specification, http://www.wotsit.org, 1995.

[36] D.E.Knuth, The DVItype processor, http://www.wotsit.org, 1995.

[37] Borland Intl., Inc. Borland Delphi 3.0. run image, файл `.\SOURCE\VCL\classes.pas`, 1997.

[38] TIS Committee, Tool Interface Standards - OMF: Relocatable Object Module Format, Version 1.1., http://www.wotsit.org, 1995.

[39] PKWARE. General Format of a ZIP file, http://www.pkware.com/download.html - appnote.zip, 1998.

[40] А.Филд, П.Харрисон. Функциональное программирование, М.: "Мир", 1993.

[41] Microsoft Corp. Microsoft multimedia standards update, http://www.wotsit.org - riff.doc, 1993.

[42] Microsoft Corp. Windows Shell API/File Viewers, MSDN Library, http://www.microsoft.com/msdn/, 1997.

[43] К.Хоггер. Введение в логическое программирование: Пер. с англ., М.: "Мир", 1988.

[44] M. Kifer, G. Lausen, J. Wu: Logical Foundations of Object-Oriented and Frame-Based Languages, Journal of the ACM, 1995, vol. 42, ftp://ftp.informatik.uni-freiburg.de/documents/papers/JACMflogic.ps.gz.

[45] SunOS, файлы `sys/elf_*.h`, 1997.

[46] В.Липский. Комбинаторика для программистов: Пер. с польск. - М.:Мир, 1998.

[47] А.Е.Хмельнов. Программа для разбора юнитов Delphi 2.0-5.0., http://monster.icc.ru/~alex/DCU/, 2000.

Приложения

1. Файлы конфигурации.

1.1. Файл описания расширений Ref.cfg.

```
: elf.rfh
dat: dat.rfi clarion.rfi
k0?: kxx.rfi
o: elf.rfh coff_obj.rfi
so: elf.rfh
lib: lib.rfi COFF_lib.rfi
obj: obj_file.rfi coff_obj.rfi
wmf: WMF.rfi EMF.rfi
emf: EMF.rfi
dcu: tpu.rfi dcu32.rfi
tpu: tpu.rfi
tpp: tpu.rfi
tpw: tpu.rfi
exe: exe.rfi
vxd: exe.rfi
dll: exe.rfi
ocx: exe.rfi
cur: ico.rfh
class: cla.rfh
gif: GIF89a.rfi
```

1.2. Файл описания форматов для WWWBinView - www.cfg.

```
STATUS
AUX=Auxiliary
Ok=Complete
Ok1=Almost Complete
OkH=Headers only
P=Partial
R=Guess Work

CLASS
GR=Graphics and Sounds
T=General Data Types
EXE=Executable and Object
ARC=Archive
RES=Resource
TXT=Text
OP=Machine OpCodes representation
DB=Database
OS=OS Specific
GIS=GIS files
FNT=Font file

//Files, which can be published, and their descriptions
FILE //(descr,class,status)
ANI.RFH:        'Windows animated cursors',GR,P
ARJ.rfh:        'ARJ Archive Format',ARC,OkH
BGI.RFI:        'Borland Graphics Interface',EXE,AUX
BMP.RFI:        'Bitmap file data structures',GR,Ok
```

```
BMP.RFH:         'Windows Bitmap file',GR,Ok
BMP_DAT.RFI:     'Bitmap file data structures',GR,Ok
BMP_OS2.RFI:     'Bitmap file header and palette (Windows or OS/2)', GR,Ok
cla.rfh:         'Java-machine executable (class file)', EXE,Ok1
CLARION.rfi:     'Clarion DBMS data file.',DB,Ok1
CMD_Z80.RFI:     'Opcode of Z-80 microprocessor as data type',OP,Ok
coff.rfi:        'Microsoft Portable Executable and Common Object File
   Format.',EXE,Ok1
COFF_LIB.RFI:    'COFF Archive (Library) File Format.',EXE,Ok1
COFF_OBJ.RFI:     'COFF Object (*.obj) File Format (Used by Microsoft Visual
C).',
   EXE,Ok1
COFF_rel.rfi:    'COFF Relocation information for different platforms',EXE,AUX
Dat.rfi:         'Windows 95 Register',OS,Ok1
db.rfh:          'Paradox DBMS data file',DB,Ok
DBF.RFH:         'xBase DBMS data file',DB,Ok
DCU32.RFI:       'Borland Delphi (vv. 2.0,3.0,4.0,5.0) Unit',EXE,R
DFM.RFH:         'Borland Delphi form',RES,Ok
DOSFTIME.RFI:    'Dos File Time data type',T,AUX
DPMI.RFI:        'DPMI call registers',T,AUX
DVI.RFH:         'TeX output file',TXT,Ok
ELF.RFH:         'Executable and Linkable Format.',EXE,Ok1
elf_386.rfi:     'ELF Intel 386 specific',EXE,AUX
elf_M32.rfi:     'ELF M32 specific',EXE,AUX
elf_ppc.rfi:     'ELF Power PC specific',EXE,AUX
elfSPARC.rfi:    'ELF SPARC specific',EXE,AUX
EXE.RFI:         'EXE (DOS,WINDOWS (NE,PE,LE),OS/2(LX))',EXE,Ok1
GIF89a.rfi:      'CompuServe GIF (GRAPHICS INTERCHANGE FORMAT)',GR,OkH
GUID.RFI:        'Windows GUID',T,AUX
gz.rfh:          'GZIP Archive Format',ARC,OkH
HLP.RFH:         'Windows Help File',TXT,P
ICO.RFH:         'Windows Icon or Cursor',GR,Ok
ICO.RFI:         'Windows Icon and Cursor data structures',GR,Ok
JAVA_COD.RFI:    'Java-machine opcodes as data types',OP,Ok
KXX.RFI:         'Clarion key/index file.',DB,Ok
LE.RFI:          'Linear Executable (Windows-95 VXD - drivers)', EXE,Ok1
LIB.RFI:         'Library files (DOS,WINDOWS)',EXE,Ok1
LNK.RFH:         'Windows Short Cut (*.lnk)',OS,Ok1
MetaFile.RFI:    'Windows Metafile data structures',GR,Ok
NE.RFI:          'Windows 16-bit Executable',EXE,Ok1
OBJ_FILE.RFi:    'OBJ-files (DOS,Windows)',EXE,Ok1
OBJ.RFI:         'OBJ and LIB files` data structures',EXE,Ok1
PIF.RFH:         'Windows Program description files',OS,Ok1
rar.rfh:         'RAR Archive Format',ARC,OkH
rpm.rfh:         'Linux RPM file header',ARC,OkH
SHP.RFH:         'ArcInfo Shape file',GIS,Ok
SHP.RFI:         'ArcInfo Shape files` data structures',GIS,Ok
SHX.RFH:         'ArcInfo Shape index file',GIS,Ok
TAR.rfh:         'TAR Archive Format',ARC,Ok
TPU.RFI:         'Turbo Pascal 6.0,7.0 and Delphi 1.0 Units',EXE,Ok1
ttf.rfh:         'True Type fonts',FNT,P
UNIXTime.RFI:    'UNIX Time data type (Sec. since 1.1.1970)',T,AUX
VERRES16.RFI:    'Version Resource in 16-bit Windows EXE files',RES,P
wav.rfh:         'Windows Sound',GR,Ok1
WIN_CF.RFI:      'Windows Clipboard Format Tags',GR,AUX
WMF.rfi:         'Windows Metafile',GR,Ok
EMF.rfi:         'Windows Enhanced Metafile',GR,Ok
ZIP.rfh:         'ZIP Archive Format',ARC,OkH
```

2. Использование деревьев решений для реализации перечисления термов в описании системы команд PDP-11.

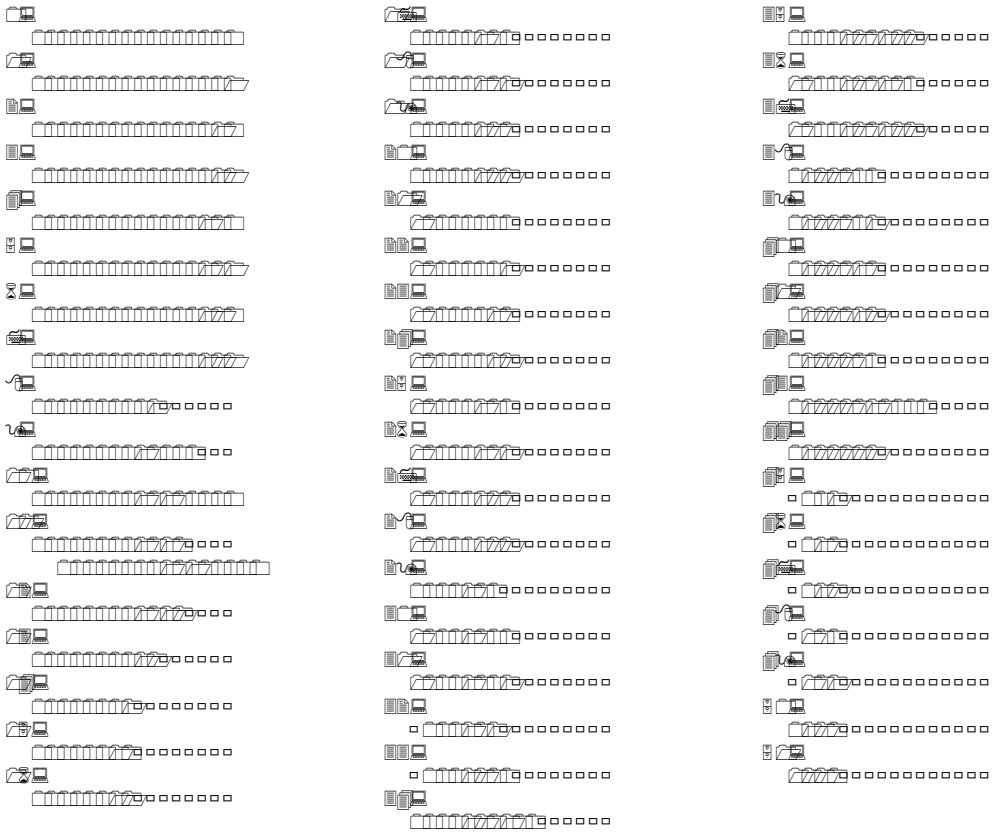
2.1. Описание кодирования машинных команд PDP-11.

```
type
TWOpCode forward
TOpPDP forward
POpSeq ^TOpSeq near=word

type bit
TBit num+(1)
TBit2 num+(2)
TBit3 num+(3)
TBit4 num+(4)
TBit6 num+(6)
TBit8 num+(8)
TBit8s num-(8)

TJmpOfs ^TOpSeq near=TBit8s, REF=(&((@:@ as TWOpCode):@ as TOpPDP)
  +(@+1)*2)and 0xFFFF;:displ=(HEX((&((@:@ as TWOpCode):@ as
   TOpPDP)+ (@+1)*2)and 0xFFFF,4),'{',@,'}')
TSOBOfs ^TOpSeq near=TBit6, REF=(&((@:@ as TWOpCode):@ as TOpPDP)
  +2-@*2)and 0xFFFF;:displ=(HEX((&((@:@ as TWOpCode):@ as TOpPDP)+
   2-@*2)and 0xFFFF,4),'{',@,'}')

TReg enum TBit3 (R0,R1,R2,R3,R4,R5,SP,PC)

TAddrMode enum TBit2 (reg,autoinc,autodec,index)
TAddr6 struc pas
  R: TReg
  Ind: TBit //Indirect mode
  Md: TAddrMode
ends

TCondCode set 4 of (C,V,Z,N)
TUnOp1 enum TBit3 (clr,com,inc,dec,neg,adc,sbc,tst)
TUnOp2 enum TBit2 (ror,rol,asr,asl)
TFOp enum TBit2 (fadd,fsub,fmul,fdiv)

TBit16 num+(16)

type
TWOpCode enum TBit16 fields (
  R0: TReg @0.3,
  R: TReg @6.3,
  CC: TCondCode @0.4,
  D: TAddr6 @0.6,
  S: TAddr6 @6.6,
  X: TJmpOfs @0.8,
  Y: TSOBOfs @0.6,
  N: TBit6 @0.6,
  INo: TBit8 @0.8,
  Fa: TUnOp1 @6.3,
  Fr: TUnOp2 @6.2,
  Ff: TFOp @3.2,
  B: TBit @15.1
) of (
  halt =            0000000,
  wait =            0000001,
  rti =             0000002,
```

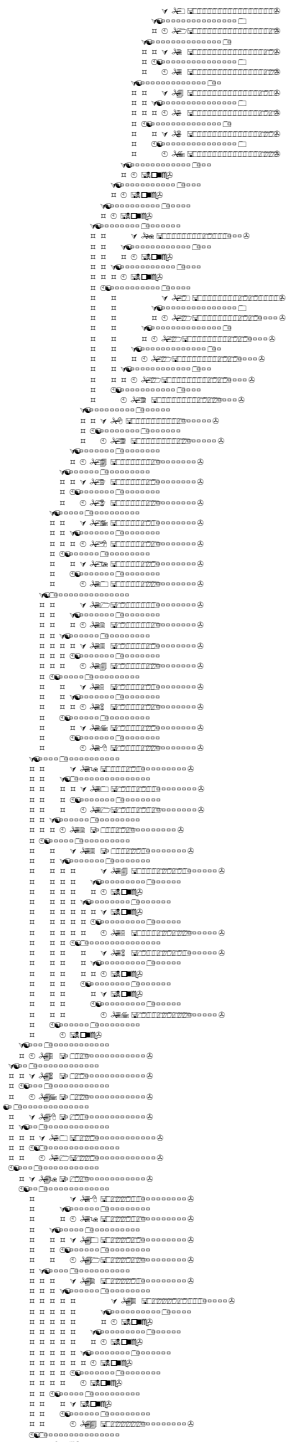
```
  bpt =             0000003,
  iot =             0000004,
  reset =           0000005,
  rtt =             0000006,
  mfpt =            0000007,
  jmp(D) =          00001__,
  rts(R0) =         000020_,
  nop =             0000240,
  clf(CC) =         0000240, //Condition codes
  stf(CC) =         0000260,
  swab(D) =         00003__,
  br(X) =           0000400,
  bne(X) =          0001000,
  beq(X) =          0001400,
  bge(X) =          0002000,
  blt(X) =          0002400,
  bgt(X) =          0003000,
  ble(X) =          0003400,
  bpl(X) =          0100000,
  bmi(X) =          0100400,
  bhi(X) =          0101000,
  blos(X) =         0101400,
  bvc(X) =          0102000,
  bvs(X) =          0102400,
  bcc(X) =          0103000, //bhis
  bcs(X) =          0103400, //blo
  jsr(R,D) =        0004___,
  emt(INo) =        0104000,
  trap(INo) =       0104400,
  _AR(Fa,B,D) =     0005000,
  _ROT(Fr,B,D) =    0006000,
  mark(N) =         00064__,
  stx(D) =          00067__,
  mtps(D) =         01064__,
  mfps(D) =         01067__,
  mul(R,D) =        0070___,
  div(R,D) =        0071___,
  ash(R,N) =        0072___,
  ashc(R,N) =       0073___,
  xor(R,D) =        0074___,
  _F(Ff,R0) =       0075000,
  sob(R,Y) =        0077___,
  mov(B,S,D) =      0_1____,
  cmp(B,S,D) =      0_2____,
  bit(B,S,D) =      0_3____,
  bic(B,S,D) =      0_4____,
  bis(B,S,D) =      0_5____,
  add(S,D) =        006____,
  sub(S,D) =        016____
)

TOperandData(R,Md) case @:R of
 TReg.PC: case @:Md of
   TAddrMode.index: word
   TAddrMode.autoinc: word
  //TAddrMode.autodec: -word
  endc
 else case @:Md of
   TAddrMode.index: word
  endc
endc:displ=(',',@)

PPOpSeq ^POpSeq near=word
POpSeqRel ^TOpSeq near=word, REF=(&@+@)and 0xFFFF; :
  displ=(HEX((&@+@)and 0xFFFF,4));
PPOpSeqRel ^TOpSeq near=word, REF=(&@+@)and 0xFFFF; :
  displ=(HEX((&@+@)and 0xFFFF,4));
```

```
TJmpOperandData(R,Md,Ind) case @:R of
 TReg.PC: case @:Md of
   TAddrMode.index: case @@:Ind of
     0: POpSeqRel
     else PPOpSeqRel
   endc
   TAddrMode.autoinc: case @@:Ind of
     0: POpSeq
     else PPOpSeq
   endc
  //TAddrMode.autodec: -word
  endc
 else case @:Md of
    TAddrMode.index: word
  endc
endc

TOpPDP struc pas
  Op: TWOpCode
  Dat: case @.Op of
    jmp,jsr: struc pas
       D: TJmpOperandData(@@@.Op.D.R exc -1,@@@.Op.D.Md exc -1,
           @@@.Op.D.Ind exc -1)
     ends: displ=(' ',@.D)
    swab,_AR,_ROT,stx,mtps,mfps,mul,div,xor: struc pas
      D: TOperandData(@@@.Op.D.R exc -1,@@@.Op.D.Md exc -1)
     ends: displ=(@.D)
    mov,cmp,bit,bic,bis,add,sub: struc pas
      S: TOperandData(@@@.Op.S.R exc -1,@@@.Op.S.Md exc -1)
      D: TOperandData(@@@.Op.D.R exc -1,@@@.Op.D.Md exc -1)
     ends: displ=(@.S,@.D)
  endc
ends: displ=(@.Op,@.Dat)

TOpSeq codes of TOpPDP ?(@.Op>=TWOpCode.br)and(@.Op<TWOpCode.br+256)
  or(@.Op>=TWOpCode.jmp)and(@.Op<TWOpCode.jmp+64)
  or(@.Op>=TWOpCode.rts)and(@.Op<TWOpCode.rts+8)
  or(@.Op>=TWOpCode.rts)and(@.Op<TWOpCode.rts+8)
  or(@.Op>=TWOpCode.mark)and(@.Op<TWOpCode.mark+64)
  or(@.Op=TWOpCode.rti)or(@.Op=TWOpCode.rtt)
  or(@.Op=TWOpCode.halt); : displ=('(',
  ShowArray(@,(NL,HEX(&@ /*-&@:@*/ ,4),': ',@)),NL,')')
```

2.2. Таблица вариантов для реализации `TWOpCode`.



2.3. Деревья решений для реализации `TWOpCode`.



Дерево с побитовыми условиями

Упакованное дерево

3. Примеры описаний форматов на языке Flext.

3.1. Формат TPU.

3.1.1. Файл TEST.PAS для Borland Pascal 7.0.

```
unit Test;
interface
type
  TLine = string[7];
  TTypeConstRec = record
    W: word;
    Ch: Char;
    L: LongInt;
    Line: TLine;
  end ;

const
  ConstRec: TTypeConstRec =
    (W: $A7B8; Ch:'X'; L: $89ABCDEF; Line: 'Text Ln');

implementation
end .
```

3.1.2. Результат разбора файла TEST.TPU.

```
File 'test.tpu':
Block: CODE
Total 0 code parts.
Total 0 code references.
Code: 00000000 of 000002F0 = 0%
0000:Hdr: THeader70 = (Magic:'TPUQ'; Long0:0; Units:00E2;
      NameTree:0060; ProcRefTbl:023F; CodeDescrTbl:0247;
      ConstGrOfs:0247; VarGrOfs:024F; DLLsOfs:024F;
      UnitCodeRefOfs:024F; FDscOfs:0263; SourceStrNum:0273;
      DebugInfoOfs:0279; DataStart:0279; SymbolInfo:0054;
      CodeSize:0000; DataSize:0010; CReloSize:0000;
      DReloSize:0000; VarSize:0000; ImplNames:01BD;
      Options:0008; ObjList:0000; W10:0020; W11:0044; W12:0000)
                          0038:00 00 00 00 00 00 00 00 ¦        ........¦
    0040:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ¦................¦
    0050:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ¦................¦
0060:Hdr.NameTree^: TNameTree = (Size:007E;
      Tbl: (0:0000,1:0000,2:0000,3:0000,4:0000,5:0000,6:0000,
        7:0000,8:0000,9:0000,10:0000,11:0000,12:0000,13:0000,
        14:0000,15:0000,16:0000,17:0000,18:0000,19:0000,20:0000,
        21:0000,22:0000,23:0000,24:0000,25:01A6,26:0000,27:0000,
        28:0000,29:0000,30:0000,31:00F3,32:0000,33:0000,34:0000,
        35:0000,36:0000,37:0000,38:0000,39:0000,40:0000,41:0000,
        42:0137,43:0000,44:0000,45:0000,46:0000,47:0000,48:0000,
        49:0000,50:0000,51:0000,52:0000,53:0000,54:0000,55:0106,
        56:0000,57:0000,58:0000,59:0000,60:00E2,61:0000,62:0000,
        63:0000))
00E2:Test_Inf: TNameInf = (Next:0000; Kind:nkUnit{'S'}; Name:'Test';
      Inf:(TPU:79490000; Next:00F3; Prev:00F3; IsSystem:00))
00F3:System_Inf: TNameInf = (Next:0000; Kind:nkUnit{'S'};
      Name:'System';
      Inf:(TPU:C40A0000; Next:0000; Prev:0000; IsSystem:01))
0106:TLine_Inf: TNameInf = (Next:0000; Kind:nkType{'P'};
      Name:'TLine'; Inf:(Ofs:0127; TPUOfs:00EA))
0113:TD_TLine.TI.IndexType.Ofs^: TTypeDefRec = (VMT:vmtByte{020C};
      TD:(size:0001; NameRef:0000; Next:0000);
      TI:(Min:0; Max:7; BasicType:(Ofs:0152; TPUOfs:00FD)))
0127:TD_TLine: TTypeDefRec = (VMT:vmtString{0009};
      TD:(size:0008; NameRef:0106; Next:0000);
      TI:(ValType:(Ofs:01CE; TPUOfs:00FD);
        IndexType:(Ofs:0113; TPUOfs:00EA)))
0137:TTypeConstRec_Inf: TNameInf = (Next:0000; Kind:nkType{'P'};
      Name:'TTypeConstRec'; Inf:(Ofs:014C; TPUOfs:00EA))
014C:TD_TTypeConstRec: TTypeDefRec = (VMT:vmtRecord{0002};
      TD:(size:000F; NameRef:0137; Next:0000);
      TI:(RefTblOfs:0158; NameLstOfs:0162))
```

```
0158:TD_TTypeConstRec.TI.RefTblOfs^: TNameTree = (Size:0006;
      Tbl: (0:0193,1:0172,2:0162,3:0183))
0162:W_Inf: TNameInf = (Next:0000; Kind:nkVar{'Q'}; Name:'W';
      Inf:(IsConst:08; VarOfs:0000; GrOfs:014C; NextOfs:0172;
        TypeRef:(Ofs:013E; TPUOfs:00FD)))
0172:Ch_Inf: TNameInf = (Next:0000; Kind:nkVar{'Q'}; Name:'Ch';
      Inf:(IsConst:08; VarOfs:0002; GrOfs:014C; NextOfs:0183;
        TypeRef:(Ofs:01CE; TPUOfs:00FD)))
0183:L_Inf: TNameInf = (Next:0000; Kind:nkVar{'Q'}; Name:'L';
      Inf:(IsConst:08; VarOfs:0003; GrOfs:014C; NextOfs:0193;
        TypeRef:(Ofs:0152; TPUOfs:00FD)))
0193:Line_Inf: TNameInf = (Next:0000; Kind:nkVar{'Q'}; Name:'Line';
      Inf:(IsConst:08; VarOfs:0007; GrOfs:014C; NextOfs:0000;
        TypeRef:(Ofs:0127; TPUOfs:00EA)))
01A6:ConstRec_Inf: TNameInf = (Next:0000; Kind:nkVar{'Q'};
      Name:'ConstRec';
      Inf:(IsConst:01; VarOfs:0000; GrOfs:0000; NextOfs:0000;
        TypeRef:(Ofs:014C; TPUOfs:00EA)))
01BD:Hdr.ImplNames^: TNameTree = (Size:007E;
      Tbl: (0:0000,1:0000,2:0000,3:0000,4:0000,5:0000,6:0000,
        7:0000,8:0000,9:0000,10:0000,11:0000,12:0000,13:0000,
        14:0000,15:0000,16:0000,17:0000,18:0000,19:0000,20:0000,
        21:0000,22:0000,23:0000,24:0000,25:01A6,26:0000,27:0000,
        28:0000,29:0000,30:0000,31:00F3,32:0000,33:0000,34:0000,
        35:0000,36:0000,37:0000,38:0000,39:0000,40:0000,41:0000,
        42:0137,43:0000,44:0000,45:0000,46:0000,47:0000,48:0000,
        49:0000,50:0000,51:0000,52:0000,53:0000,54:0000,55:0106,
        56:0000,57:0000,58:0000,59:0000,60:00E2,61:0000,62:0000,
        63:0000))
023F:Hdr.ProcRefTbl^: TProcRefTbl =  (
      0:(W0:0000; W1:0000; CTblOfs:FFFF; CodeStart:FFFF))
0247:Hdr.CodeDescrTbl^: TCodeDescrTbl =  ()
0247:Hdr.ConstGrOfs^: TConstGrDescrTbl =  (
      0:(Ofs:0000; Size:0010; ReloSize:0000; ObjRef:0000))
024F:Hdr.VarGrOfs^: TVarGrDescrTbl =  ()
024F:Hdr.DLLsOfs^: TDLLDescrTbl =  ()
024F:Hdr.UnitCodeRefOfs^: TUnitCodeRefTbl =  (
      0:(DataPtr:000C0000; Name:'System'),
      1:(DataPtr:00180000; Name:'Test'))
0263:Hdr.FDscOfs^: TFileDescrTbl =  (
      0:(Tag:srcMain{04}; W0:0000; DateTimePack:255E55EA;
        FName:'TEST.PAS'))
0273:Hdr.SourceStrNum^: TSourceStrInfo = (W0:0000; W1:0000;
      SrcLines:0014)
0279:Hdr.DebugInfoOfs^: TTPUDataBlocks = (DebugInfo:[]; FAl0:
    0279:00 00 00 00 00 00 00 ¦.......¦
      ; SymbolInfo:
    0000:E2 00 01 00 06 01 05 00 37 01 06 00 62 01 07 00 ¦ň.......7...b...¦
    0010:72 01 08 00 83 01 09 00 93 01 0A 00 A6 01 0E 00 ¦r...Ă...Ó...ć...¦
    0020:B4 07 07 00 D2 02 08 00 DF 04 09 00 06 01 0A 00 ¦+...T...-.......¦
    0030:37 01 0E 00 62 01 0F 00 72 01 0F 00 83 01 0F 00 ¦7...b...r...Ă...¦
    0040:93 01 0F 00 06 00 0B 00 4C 01 0B 00 0E 00 10 00 ¦Ó.......L.......¦
    0050:4C 01 0B 00                                     ¦L...            ¦
      ; FAl1:
    02D4:00 00 00 00 00 00 00 00 00 00 00 00 ¦............¦
      ; CodeInf: (); FAl2:[]; DataInf: (0:
    0000:B8 A7 58 EF CD AB 89 07 54 65 78 74 20 4C 6E 00 ¦¬çX˙=ëÉ.Text Ln.¦
      ); FAl3:[]; CRelo: (); FAl4:[]; DRelo: ())
End of CODE.
```

3.2. Формат CLASS.

3.2.1. Файл класса Java-машины.

```
set byteorder rev
type
  u1 num +(1)
  u2 num +(2)
  u4 num +(4)

data
0 u4 magic

assert magic=0xCAFEBABE;
```

```
descr ('CLASS file (Java VM executable).',NL,
  'Info Source: The JavaTM Virtual Machine Specification',NL,
  '  by Tim Lindholm and Frank Yellin',NL,
  '  at http://java.sun.com',NL)

type
  TCPInfoTag enum u1 (
    C_Class = 7,
    C_Fieldref = 9,
    C_Methodref = 10,
    C_InterfaceMethodref = 11,
    C_String = 8,
    C_Integer = 3,
    C_Float = 4,
    C_Long = 5,
    C_Double = 6,
    C_NameAndType = 12,
    C_Utf8 = 1
  )

  TUtf8NDX u2()
  TClassNDX u2()
  TNameTypeNDX u2()

  TRefInfo struc
    TClassNDX class_index
    TNameTypeNDX NameTypeNdx
  ends
  TLongInfo struc
    u4 high_bytes
    u4 low_bytes
  ends
  TNameTypeInfo struc
    TUtf8NDX nameNdx %Utf8
    TUtf8NDX descrNdx %Utf8
  ends
  TUtf8Str struc
    u2 len
    array[@.len] of Char val %in fact Utf-8 encoding
  ends

  cp_info struc
    TCPInfoTag Tag
    case @.Tag of
      C_Class: TUtf8NDX %name_index,Utf8
      C_Fieldref,
      C_Methodref,
      C_InterfaceMethodref: TRefInfo
      C_String: TUtf8NDX %string_index,Utf8
      C_Integer: u4 %Bytes
      C_Float: u4 %Bytes,IEEE 754
      C_Long: TLongInfo
      C_Double: TLongInfo
      C_NameAndType: TNameTypeInfo
      C_Utf8: TUtf8Str
    endc info
  ends

  attribute_info forward
  TFieldAccessFlags set 16 of (
  //Flag Name    Value   Meaning                          Used By
    ACC_PUBLIC =    0, //Is public; may be accessed
                       //from outside its package.      Any field
    ACC_PRIVATE =   1, //Is private; usable only within
                       //the defining class.            Class field
    ACC_PROTECTED = 2, //Is protected; may be accessed
                       //within subclasses.             Class field
```

```
  ACC_STATIC =    3, //Is static.                    Any field
  ACC_FINAL =     4, //Is final; no further
                     //overriding or assignment after
                     //initialization.               Any field
  ACC_VOLATILE =  6, //Is volatile; cannot be cached. Class field
  ACC_TRANSIENT = 7  //Is transient; not written or
                     //read by a persistent object
                     //manager.                      Class field
)

field_info struc
  TFieldAccessFlags access_flags
  TUtf8NDX nameNdx
  TUtf8NDX descrNdx
  u2 attr_count
  array[@.attr_count]of attribute_info attributes
ends

TMethodAccessFlags set 16 of (
//Flag Name    Value   Meaning                       Used By
  ACC_PUBLIC =    0, //Is public; may be accessed
                     //from outside its package.     Any method
  ACC_PRIVATE =   1, //Is private; usable only within
                     //the defining class.           Class/instance
  ACC_PROTECTED = 2, //Is protected; may be accessed
                     //within subclasses.            Class/instance
  ACC_STATIC =    3, //Is static.                    Class/instance
  ACC_FINAL =     4, //Is final; no overriding
                     //allowed.                      Class
  ACC_SYNCHRONIZED=5,//Is synchronized; wrap use in
                     //monitor lock.                 Class/instance
  ACC_NATIVE=     8, //Is native; implemented in a
                     //language other than Java.     Class/instance
  ACC_ABSTRACT = 10  //Is abstract; no implementation
                     //is provided.                  Any method
)

method_info struc
  TMethodAccessFlags access_flags
  TUtf8NDX nameNdx
  TUtf8NDX descrNdx
  u2 attr_count
  array[@.attr_count]of attribute_info attributes
ends

TClassAccessFlags set 16 of (
//Flag Name    Value   Meaning                       Used By
  ACC_PUBLIC =    0, //Is public; may be accessed    Class,
                     //from outside its package.     interface
  ACC_FINAL =     4, //Is final; no subclasses
                     //allowed.                      Class
  ACC_SUPER =     5, //Treat superclass methods      Class,
                     //specially in invokespecial.   interface
  ACC_INTERFACE = 9, //Is an interface.              Interface
  ACC_ABSTRACT = 10  //Is abstract; may not be       Class,
                     //instantiated.                 interface
)

TClassFile struc
  u2 minor_version
  u2 major_version
  u2 C_pool_count
  array[@.C_pool_count-1]of cp_info TAKES 2 when(
    (@.Tag=TCPInfoTag.C_Long)or(@.Tag=TCPInfoTag.C_Double)); C_pool
  TClassAccessFlags access_flags
  TClassNDX this_class
  TClassNDX super_class
```

```
    u2 interfaces_count
    array[@.interfaces_count]of TClassNDX interfaces
    u2 fields_count
    array[@.fields_count] of field_info fields
    u2 methods_count
    array[@.methods_count]of method_info methods
    u2 attr_count
    array[@.attr_count]of attribute_info attributes
  ends

data
0004 TClassFile Hdr

type
  TUtf8NDX enum TUtf8NDX
     Hdr.C_pool[@-1].info.C_Utf8.val;
  TClassNDX enum TClassNDX
     Hdr.C_pool[@-1].info.C_Class;
  TNameTypeNDX enum TNameTypeNDX
     Hdr.C_pool[@-1].info.C_NameAndType.nameNdx;
  TRefNDX enum u2
     Hdr.C_pool[@-1].info.C_Fieldref.NameTypeNdx;
  TFldRefNdx TRefNDX
  TMethRefNdx TRefNDX
  TIMethRefNdx TRefNDX

  TConstNDX1 num+(1):displ=(Hdr.C_pool[@-1],'{',@,'}')
  TConstNDX num+(2):displ=(Hdr.C_pool[@-1],'{',@,'}')

include java_cod.rfi

type
  TAttrData raw[]

  TCodeExcRec struc
    u2 start_pc
    u2 end_pc
    u2 handler_pc
    TClassNDX catch_type
  ends

  TCodeAttr struc
    u2 max_stack
    u2 max_locals
    u4 CodeLen
    TOpSeries code
    u2 ExcTblLen
    array[@.ExcTblLen]of TCodeExcRec exc_tbl
    u2 AttrCnt
    array[@.AttrCnt]of attribute_info attributes
  ends:[code:Size=@.CodeLen]

  TExcAttr struc
    u2 NumExc
    array[@.NumExc]of TClassNDX ExcNdxTbl
  ends

  TLinNumRec struc
    u2 pc
    u2 l
  ends

  TLinNumAttr struc
    u2 Len
    array[@.Len]of TLinNumRec LinNumTbl
  ends
```

```
  TLocVarRec struc
    u2 start_pc
    u2 len
    TUtf8NDX name_index
    TUtf8NDX descr_index
    u2 index
  ends

  TLocVarTblAttr struc
    u2 Len
    array[@.Len]of TLocVarRec LocVarTbl
  ends

  attribute_info struc
    TUtf8NDX name_index %Ult8
    u4 len
    case Hdr.C_pool[@.name_index-1].
         info.C_Utf8.val of
      'SourceFile': TUtf8NDX
      'ConstantValue': TConstNDX
      'Code': TCodeAttr
      'Exceptions': TExcAttr
      'LineNumberTable': TLinNumAttr
      'LocalVariableTable': TLocVarTblAttr
    else
      TAttrData
    endc info
  ends:[info:Size=@.len]
```

3.2.2. Кодирование машинных команд Java-машины.

```
type

TOpKind enum u1 (
  aaload = 0x32, aastore = 0x53, aconst_null = 0x01, aload = 0x19,
  aload_0 = 0x2a, aload_1 = 0x2b, aload_2 = 0x2c aload_3 = 0x2d,
  anewarray = 0xbd, areturn = 0xb0, arraylength = 0xbe, astore = 0x3a,
  astore_0 = 0x4b, astore_1 = 0x4c, astore_2 = 0x4d, astore_3 = 0x4e,
  athrow = 0xbf, baload = 0x33, bastore = 0x54, bipush = 0x10,
  caload = 0x34, castore = 0x55, checkcast = 0xc0,
  d2f = 0x90, d2i = 0x8e, d2l = 0x8f,
  dadd = 0x63, daload = 0x31, dastore = 0x52, dcmpg = 0x98, dcmpl = 0x97,
  dconst_0 = 0xe, dconst_1 = 0xf, ddiv = 0x6f, dload = 0x18,
  dload_0 = 0x26, dload_1 = 0x27, dload_2 = 0x28, dload_3 = 0x29,
  dmul = 0x6b, dneg = 0x77, drem = 0x73, dreturn = 0xaf, dstore = 0x39,
  dstore_0 = 0x47, dstore_1 = 0x48, dstore_2 = 0x49, dstore_3 = 0x4a,
  dsub = 0x67, dup = 0x59, dup_x1 = 0x5a, dup_x2 = 0x5b,
  dup2 = 0x5c, dup2_x1 = 0x5d, dup2_x2 = 0x5e,
  f2d = 0x8d, f2i = 0x8b, f2l = 0x8c,
  fadd = 0x62, faload = 0x30, fastore = 0x51, fcmpg = 0x96, fcmpl = 0x95,
  fconst_0 = 0xb, fconst_1 = 0xc, fconst_2 = 0xd, fdiv = 0x6e, fload = 0x17,
  fload_0 = 0x22, fload_1 = 0x23, fload_2 = 0x24, fload_3 = 0x25,
  fmul = 0x6a, fneg = 0x76, frem = 0x72, freturn = 0xae, fstore = 0x38,
  fstore_0 = 0x43, fstore_1 = 0x44, fstore_2 = 0x45, fstore_3 = 0x46,
  fsub = 0x66, getfield = 0xb4, getstatic = 0xb2, goto = 0xa7, goto_w = 0xc8,
  i2b = 0x91, i2c = 0x92, i2d = 0x87, i2f = 0x86, i2l = 0x85, i2s = 0x93,
  iadd = 0x60, iaload = 0x2e, iand = 0x7e, iastore = 0x4f,
  iconst_m1 = 0x2, iconst_0 = 0x3, iconst_1 = 0x4,
  iconst_2 = 0x5, iconst_3 = 0x6, iconst_4 = 0x7, iconst_5 = 0x8,
  idiv = 0x6c, if_acmpeq = 0xa5, if_acmpne = 0xa6,
  if_icmpeq = 0x9f, if_icmpne = 0xa0, if_icmplt = 0xa1,
  if_icmpge = 0xa2, if_icmpgt = 0xa3, if_icmple = 0xa4,
  ifeq = 0x99, ifne = 0x9a, iflt = 0x9b, ifge = 0x9c, ifgt = 0x9d, ifle = 0x9e,
  ifnonnull = 0xc7, ifnull = 0xc6, iinc = 0x84, iload = 0x15,
  iload_0 = 0x1a, iload_1 = 0x1b, iload_2 = 0x1c, iload_3 = 0x1d,
  imul = 0x68, ineg = 0x74, instanceof = 0xc1, invokeinterface = 0xb9,
```

```
  invokespecial = 0xb7, invokestatic = 0xb8, invokevirtual = 0xb6,
  ior = 0x80, irem = 0x70, ireturn = 0xac, ishl = 0x78, ishr = 0x7a, istore =
0x36,
  istore_0 = 0x3b, istore_1 = 0x3c, istore_2 = 0x3d, istore_3 = 0x3e,
  isub = 0x64, iushr = 0x7c, ixor = 0x82, jsr = 0xa8, jsr_w = 0xc9,
  l2d = 0x8a, l2f = 0x89, l2i = 0x88, ladd = 0x61, laload = 0x2f, land = 0x7f,
  lastore = 0x50, lcmp = 0x94, lconst_0 = 0x9, lconst_1 = 0xa,
  ldc = 0x12, ldc_w = 0x13, ldc2_w = 0x14, ldiv = 0x6d, lload = 0x16,
  lload_0 = 0x1e, lload_1 = 0x1f, lload_2 = 0x20, lload_3 = 0x21,
  lmul = 0x69, lneg = 0x75, lookupswitch = 0xab, lor = 0x81, lrem = 0x71,
  lreturn = 0xad, lshl = 0x79, lshr = 0x7b, lstore = 0x37,
  lstore_0 = 0x3f, lstore_1 = 0x40, lstore_2 = 0x41, lstore_3 = 0x42,
  lsub = 0x65, lushr = 0x7d, lxor = 0x83, monitorenter = 0xc2,
  monitorexit = 0xc3, multianewarray = 0xc5, new = 0xbb, newarray = 0xbc,
  nop = 0x0, pop = 0x57, pop2 = 0x58, putfield = 0xb5, putstatic = 0xb3,
  ret = 0xa9, return = 0xb1, saload = 0x35, sastore = 0x56, sipush = 0x11,
  swap = 0x5f, tableswitch = 0xaa, wide = 0xc4
)

TIntSIRec struc
  u1 locndx
  sint c
ends
TIntIRec struc
  u2 locndx
  int c
ends
TInterfaceCallRec struc
  TIMethRefNdx hMethod
  u1 nargs
  u1 O
ends

TPadZeroes(BaseA) align 4 at @:BaseA;

jmpofs2(OpcOfs) num-(2):displ=(HEX(@+@:OpcOfs,4))
jmpofs4(OpcOfs) num-(4):displ=(HEX(@+@:OpcOfs,8))

TLookupRec(OpcOfs) struc
  u4 val
  jmpofs4(@:OpcOfs) ofs
ends

TLookupSwitchRec(BaseA) struc
  TPadZeroes(@:BaseA) Pad
  jmpofs4(&@-1-@:BaseA) Deft
  u4 npairs
  array[@.npairs]of TLookupRec(&@@-1-@@:BaseA) pair
ends

TMultiANewArrayRec struc
  TClassNDX ArrayClass
  u1 Dim
ends

TArrayKind enum u1 (T_BOOLEAN = 4,  T_CHAR = 5,  T_FLOAT = 6,
  T_DOUBLE = 7,  T_BYTE = 8,  T_SHORT = 9,  T_INT = 10,  T_LONG = 11)

TTableSwitchRec(BaseA) struc
  TPadZeroes(@:BaseA) Pad
  jmpofs4(&@-1-@:BaseA) Deft
  u4 low
  u4 high
  array[@.high-@.low+1]of jmpofs4(&@@-1-@@:BaseA) JmpTbl
ends

TWideOp struc %modifies index
  TOpKind K
```

```
  case @.K of
    iload, fload, aload, lload, dload,
    istore, fstore, astore, lstore, dstore,
    ret: u2
    iinc: TIntIRec
  endc Op
ends

TOp struc
  TOpKind K
  case @.K of
    aload,astore,
    bipush,dload,dstore,fload,fstore,
    iload,istore,lload,lstore,ret: u1
    anewarray,checkcast,instanceof: TClassNDX
    getfield,getstatic,putfield,putstatic: TFldRefNdx
    invokespecial,invokestatic,invokevirtual: TMethRefNdx
    ldc: TConstNDX1
    ldc_w,ldc2_w: TConstNDX
    new: TClassNDX
    goto,if_acmpeq,if_acmpne,
    if_icmpeq, if_icmpne, if_icmplt,
    if_icmpge, if_icmpgt, if_icmple,
    ifeq, ifne, iflt, ifge, ifgt, ifle,
    ifnonnull, ifnull,
    jsr: jmpofs2(&@@-&@@:@)
    goto_w,jsr_w: jmpofs4(&@@-&@@:@)
    iinc: TIntSIRec
    invokeinterface: TInterfaceCallRec
    lookupswitch: TLookupSwitchRec(&@@:@)
    tableswitch: TTableSwitchRec(&@@:@)
    multianewarray: TMultiANewArrayRec
    newarray: TArrayKind
    sipush: int
    wide: TWideOp
  endc Arg
ends: displ=(@.K,' ',@.Arg)

TOpSeries array of TOp: displ=('(',
  ShowArray(@,(NL,HEX(&@-&@:@,4),': ',@)),NL,')')
```

3.2.3. Пример программы на языке Java.

```
import java.awt.*;
import java.applet.*;
public class Hello extends Applet
  {
   Image art;
   public void init() {
     art = getImage(getDocumentBase(), getParameter("image"));
   }
   public void paint(Graphics g)
     {
      Font font = new Font("TimesRoman", Font.BOLD, 24);
      g.setFont(font);
      g.drawString("Hello",5,25);
      g.drawImage(art, 100, 100, this);
     }
  }
```

3.2.4. Результат разбора файла класса.

```
File 'HELLO.CLA'
CLASS file (Java VM executable).
Info Source: The JavaTM Virtual Machine Specification
  by Tim Lindholm and Frank Yellin
  at http://java.sun.com

Block: CODE
```

```
No code detected. Block size: 0000038B.
0000:magic: u4 = CAFEBABE
0004:Hdr: TClassFile = (minor_version:0003; major_version:002D; C_pool_count:003A;
  C_pool: (0:(Tag:C_String{08}; info:'image'{001D}),
    1:(Tag:C_String{08}; info:'TimesRoman'{0021}),
    2:(Tag:C_String{08}; info:'Hello'{0037}),
    3:(Tag:C_Class{07}; info:'java/awt/Font'{0033}),
    4:(Tag:C_Class{07}; info:'java/applet/Applet'{002B}),
    5:(Tag:C_Class{07}; info:'Hello'{0037}),
    6:(Tag:C_Class{07}; info:'java/awt/Graphics'{0029}),
    7:(Tag:C_Methodref{0A};
      info:(class_index:'java/applet/Applet'{0005}; NameTypeNdx:'<init>'{0012})),
    8:(Tag:C_Methodref{0A};
      info:(class_index:'java/awt/Graphics'{0007};
        NameTypeNdx:'drawImage'{0011})),
    9:(Tag:C_Fieldref{09};
      info:(class_index:'Hello'{0006}; NameTypeNdx:'art'{0019})),
    10:(Tag:C_Methodref{0A};
      info:(class_index:'java/awt/Graphics'{0007};
        NameTypeNdx:'drawString'{0018})),
    11:(Tag:C_Methodref{0A};
      info:(class_index:'java/awt/Graphics'{0007}; NameTypeNdx:'setFont'{0013})),
    12:(Tag:C_Methodref{0A};
      info:(class_index:'java/applet/Applet'{0005};
        NameTypeNdx:'getImage'{0014})),
    13:(Tag:C_Methodref{0A};
      info:(class_index:'java/awt/Font'{0004}; NameTypeNdx:'<init>'{0017})),
    14:(Tag:C_Methodref{0A};
      info:(class_index:'java/applet/Applet'{0005};
        NameTypeNdx:'getParameter'{0015})),
    15:(Tag:C_Methodref{0A};
      info:(class_index:'java/applet/Applet'{0005};
        NameTypeNdx:'getDocumentBase'{0016})),
    16:(Tag:C_NameAndType{0C};
      info:(nameNdx:'drawImage'{0034};
        descrNdx:'(Ljava/awt/Image;IILjava/awt/image/ImageObserver;)Z'{0036})),
    17:(Tag:C_NameAndType{0C};
      info:(nameNdx:'<init>'{002F}; descrNdx:'()V'{0038})),
    18:(Tag:C_NameAndType{0C};
      info:(nameNdx:'setFont'{002C}; descrNdx:'(Ljava/awt/Font;)V'{002E})),
    19:(Tag:C_NameAndType{0C};
      info:(nameNdx:'getImage'{002D};
        descrNdx:'(Ljava/net/URL;Ljava/lang/String;)Ljava/awt/Image;'{001E})),
    20:(Tag:C_NameAndType{0C};
      info:(nameNdx:'getParameter'{001C};
        descrNdx:'(Ljava/lang/String;)Ljava/lang/String;'{0031})),
    21:(Tag:C_NameAndType{0C};
      info:(nameNdx:'getDocumentBase'{0024}; descrNdx:'()Ljava/net/URL;'{002A})),
    22:(Tag:C_NameAndType{0C};
      info:(nameNdx:'<init>'{002F}; descrNdx:'(Ljava/lang/String;II)V'{0030})),
    23:(Tag:C_NameAndType{0C};
      info:(nameNdx:'drawString'{0027};
        descrNdx:'(Ljava/lang/String;II)V'{0030})),
    24:(Tag:C_NameAndType{0C};
      info:(nameNdx:'art'{0023}; descrNdx:'Ljava/awt/Image;'{0032})),
    25:(Tag:C_Utf8{01}; info:(len:0016; val:'(Ljava/awt/Graphics;)V')),
    26:(Tag:C_Utf8{01}; info:(len:000D; val:'ConstantValue')),
    27:(Tag:C_Utf8{01}; info:(len:000C; val:'getParameter')),
    28:(Tag:C_Utf8{01}; info:(len:0005; val:'image')),
    29:(Tag:C_Utf8{01};
      info:(len:0032; val:'(Ljava/net/URL;Ljava/lang/String;)Ljava/awt/Image;')),
    30:(Tag:C_Utf8{01}; info:(len:000A; val:'Exceptions')),
    31:(Tag:C_Utf8{01}; info:(len:000F; val:'LineNumberTable')),
    32:(Tag:C_Utf8{01}; info:(len:000A; val:'TimesRoman')),
    33:(Tag:C_Utf8{01}; info:(len:000A; val:'SourceFile')),
    34:(Tag:C_Utf8{01}; info:(len:0003; val:'art')),
    35:(Tag:C_Utf8{01}; info:(len:000F; val:'getDocumentBase')),
    36:(Tag:C_Utf8{01}; info:(len:000E; val:'LocalVariables')),
    37:(Tag:C_Utf8{01}; info:(len:0004; val:'Code')),
    38:(Tag:C_Utf8{01}; info:(len:000A; val:'drawString')),
    39:(Tag:C_Utf8{01}; info:(len:0004; val:'init')),
    40:(Tag:C_Utf8{01}; info:(len:0011; val:'java/awt/Graphics')),
    41:(Tag:C_Utf8{01}; info:(len:0010; val:'()Ljava/net/URL;')),
    42:(Tag:C_Utf8{01}; info:(len:0012; val:'java/applet/Applet')),
    43:(Tag:C_Utf8{01}; info:(len:0007; val:'setFont')),
    44:(Tag:C_Utf8{01}; info:(len:0008; val:'getImage')),
    45:(Tag:C_Utf8{01}; info:(len:0012; val:'(Ljava/awt/Font;)V')),
    46:(Tag:C_Utf8{01}; info:(len:0006; val:'<init>')),
    47:(Tag:C_Utf8{01}; info:(len:0017; val:'(Ljava/lang/String;II)V')),
```

```
  48:(Tag:C_Utf8{01};
    info:(len:0026; val:'(Ljava/lang/String;)Ljava/lang/String;')),
  49:(Tag:C_Utf8{01}; info:(len:0010; val:'Ljava/awt/Image;')),
  50:(Tag:C_Utf8{01}; info:(len:000D; val:'java/awt/Font')),
  51:(Tag:C_Utf8{01}; info:(len:0009; val:'drawImage')),
  52:(Tag:C_Utf8{01}; info:(len:0005; val:'paint')),
  53:(Tag:C_Utf8{01};
    info:(len:0033; val:'(Ljava/awt/Image;IILjava/awt/image/ImageObserver;)Z')),
  54:(Tag:C_Utf8{01}; info:(len:0005; val:'Hello')),
  55:(Tag:C_Utf8{01}; info:(len:0003; val:'()V')),
  56:(Tag:C_Utf8{01}; info:(len:000A; val:'Hello.java')));
access_flags:[ACC_PUBLIC]; this_class:'Hello'{0006};
super_class:'java/applet/Applet'{0005}; interfaces_count:0000;
interfaces: (); fields_count:0001;
fields: (
  0:(access_flags:[]; nameNdx:'art'{0023}; descrNdx:'Ljava/awt/Image;'{0032};
    attr_count:0000; attributes: ())); methods_count:0003;
methods: (
  0:(access_flags:[ACC_PUBLIC]; nameNdx:'init'{0028}; descrNdx:'()V'{0038};
    attr_count:0001;
    attributes: (
      0:(name_index:'Code'{0026}; len:0000002F;
        info:(max_stack:0005; max_locals:0001; CodeLen:00000013; code:(
            0000: aload_0{2A}
            0001: aload_0{2A}
            0002: aload_0{2A}
            0003: invokevirtual{B6} 'getDocumentBase'{0010}
            0006: aload_0{2A}
            0007: ldc{12} (Tag:C_String{08}; info:'image'{001D}){01}
            0009: invokevirtual{B6} 'getParameter'{000F}
            000C: invokevirtual{B6} 'getImage'{000D}
            000F: putfield{B5} 'art'{000A}
            0012: return{B1}
          ); ExcTblLen:0000; exc_tbl: (); AttrCnt:0001;
          attributes: (
            0:(name_index:'LineNumberTable'{0020}; len:0000000A;
              info:(Len:0002;
                LinNumTbl: (0:(pc:0000; l:0009),1:(pc:0012; l:0008)))))))),
  1:(access_flags:[ACC_PUBLIC]; nameNdx:'paint'{0035};
    descrNdx:'(Ljava/awt/Graphics;)V'{001A}; attr_count:0001;
    attributes: (
      0:(name_index:'Code'{0026}; len:00000052;
        info:(max_stack:0005; max_locals:0003; CodeLen:0000002A; code:(
            0000: new{BB} 'java/awt/Font'{0004}
            0003: dup{59}
            0004: ldc{12} (Tag:C_String{08}; info:'TimesRoman'{0021}){02}
            0006: iconst_1{04}
            0007: bipush{10} 18
            0009: invokespecial{B7} '<init>'{000E}
            000C: astore_2{4D}
            000D: aload_1{2B}
            000E: aload_2{2C}
            000F: invokevirtual{B6} 'setFont'{000C}
            0012: aload_1{2B}
            0013: ldc{12} (Tag:C_String{08}; info:'Hello'{0037}){03}
            0015: iconst_5{08}
            0016: bipush{10} 19
            0018: invokevirtual{B6} 'drawString'{000B}
            001B: aload_1{2B}
            001C: aload_0{2A}
            001D: getfield{B4} 'art'{000A}
            0020: bipush{10} 64
            0022: bipush{10} 64
            0024: aload_0{2A}
            0025: invokevirtual{B6} 'drawImage'{0009}
            0028: pop{57}
            0029: return{B1}
          ); ExcTblLen:0000; exc_tbl: (); AttrCnt:0001;
          attributes: (
            0:(name_index:'LineNumberTable'{0020}; len:00000016;
              info:(Len:0005;
                LinNumTbl: (0:(pc:0000; l:000D),1:(pc:000D; l:000E),
                  2:(pc:0012; l:000F),3:(pc:001B; l:0010),4:(pc:0029; l:000B)))))))),
  2:(access_flags:[ACC_PUBLIC]; nameNdx:'<init>'{002F}; descrNdx:'()V'{0038};
    attr_count:0001;
    attributes: (
      0:(name_index:'Code'{0026}; len:0000001D;
        info:(max_stack:0001; max_locals:0001; CodeLen:00000005; code:(
            0000: aload_0{2A}
            0001: invokespecial{B7} '<init>'{0008}
            0004: return{B1}
          ); ExcTblLen:0000; exc_tbl: (); AttrCnt:0001;
```

```
            attributes: (
              0:(name_index:'LineNumberTable'{0020}; len:00000006;
                info:(Len:0001; LinNumTbl: (0:(pc:0000; l:0005))))))))));
  attr_count:0001;
  attributes: (
    0:(name_index:'SourceFile'{0022}; len:00000002; info:'Hello.java'{0039})))
End of CODE.
```

3.3. Команды процессора Z-80.

3.3.1. Кодирование машинных команд процессора Z-80.

```
type bit

TBit num+(1)
TBit2 num+(2)
TBit3 num+(3)
TRN enum TBit3 (B,C,D,E,H,L,M,A)
TRNp enum TBit2 (BC,DE,HL,SP)
TRNw enum TBit2 (BC,DE,HL,PSW)
TRN1 enum TBit (BC,DE)

TOpAr enum TBit3 (add,adc,sub,sbb,ana,xra,ora,cmp)
TOpArI enum TBit3 (adi,aci,sui,sbi,ani,xri,ori,cpi)
TOpJmpCond enum TBit3 (jnz,jz,jnc,jc,jpo,jpe,jp,jm)
TOpCallCond enum TBit3 (cnz,cz,cnc,cc,cpo,cpe,cp,cm)
TOpRetCond enum TBit3 (rnz,rz,rnc,rc,rpo,rpe,rp,rm)

TOpRot enum TBit2 (rlc,rrc,ral,rar)
TOpFlag enum TBit2 (daa,cma,stc,cmc)
TOpSLD enum TBit2 (shld,lhld,sta,lda)


TBit8 num+(8)

type
POpZ80Seq ^TOpZ80Seq near=word

TByteOpCode enum TBit8 fields (
  Nd: TRN @3.3,
  Ns: TRN @0.3,
  Np: TRNp @4.2,
  N:  TRN1 @4.1,
  Fs: TOpRot @3.2,
  Fm: TOpFlag @3.2,
  Fa: TOpAr @3.3,
  Nw: TRNw @4.2,
  Fr: TOpRetCond @3.3,
  Na: TBit3 @3.3,
  Fi: TOpArI @3.3,
  Fd: TOpSLD @3.2,
  Fj: TOpJmpCond @3.3,
  Fc: TOpCallCond @3.3
) of (
 // 1-byte
  nop =           0b00000000,
  inr(Nd) =       0b00___100,
  dcr(Nd) =       0b00___101,
  hlt =           0b01110110,
  mov(Nd,Ns) =    0b01______,
  inx(Np) =       0b00__0011,
  dcx(Np) =       0b00__1011,
  dad(Np) =       0b00__1001,
  stax(N) =       0b000_0010,
  ldax(N) =       0b000_1010,
  ROT_OP(Fs) =    0b000__111,
```

```
  FLAG_OP(Fm) =  0b001__111,
  AR_OP(Fa,Ns) = 0b10_____,
  push(Nw) =     0b11__0101,
  pop(Nw) =      0b11__0001,
  ret =          0b11001001,
  RETC_OP(Fr) =  0b11___000,
  rst(Na) =      0b11___111,
  pchl =         0b11101001,
  xchg =         0b11101011,
  xthl =         0b11100011,
  sphl =         0b11111001,
  di =           0b11110011,
  ei =           0b11111011,

// 2-byte
  mvi(Nd) =      0b00___110,
  ARI_OP(Fi) =   0b11___110,
  in =           0b11011011,
  out =          0b11010011,

// 3-byte
  lxi(Np) =      0b00__0001,
  SLD_OP(Fd) =   0b001__010,
  jmp =          0b11000011,
  JC_OP(Fj) =    0b11___010,
  call =         0b11001101,
  CC_OP(Fc) =    0b11___100
)

TOpZ80 struc pas
  Op: TByteOpCode
  Arg: case @.Op of
    mvi, ARI_OP: byte
    in, out: byte
    LXI: WORD
    SLD_OP: WORD
    jmp,JC_OP,call,CC_OP: POpZ80Seq
   endc
ends:displ=(@.Op,',',@.Arg)

TOpZ80Seq codes of TOpZ80 ?(@.Op=TByteOpCode.ret)or(@.Op=TByteOpCode.jmp)
  or(@.Op=TByteOpCode.hlt)or(@.Op=TByteOpCode.pchl);
```

3.3.2. Фрагмент результата дизассемблирования файла.

```
*** Part #66 ***

ReadFName:
@l1429(DC24, DCC7):
  DB30: 06 06               ¦..     ¦mvi(Nd:B{0}){06},06
  DB32: 11 8C BE            ¦.M-    ¦lxi(Np:DE{1}){11},BE8C
  DB35: FE 22               ¦¦"     ¦ARI_OP(Fi:cpi{7}){FE},22
  DB37: C2 D0 02            ¦T¦.    ¦JC_OP(Fj:jnz{0}){C2},02D0{->@l12}
  DB3A: C3 4B DB            ¦+K-    ¦jmp{C3},DB4B{->@j432}
@l1431(DCF6):
  DB3D: 06 06               ¦..     ¦mvi(Nd:B{0}){06},06
  DB3F: 11 8C BE            ¦.M-    ¦lxi(Np:DE{1}){11},BE8C
  DB42: B7                  ¦¬      ¦AR_OP(Fa:ora{6}; Ns:A{7}){B7},
  DB43: CA 63 DB            ¦¦c-    ¦JC_OP(Fj:jz{1}){CA},DB63{->@j437}
  DB46: FE 22               ¦¦"     ¦ARI_OP(Fi:cpi{7}){FE},22
  DB48: C2 D0 02            ¦T¦.    ¦JC_OP(Fj:jnz{0}){C2},02D0{->@l12}
@j432(DB3B):
  DB4B: 23                  ¦#      ¦inx(Np:HL{2}){23},
@l1433(DB57, DB57):
  DB4C: 7E                  ¦~      ¦mov(Nd:A{7}; Ns:M{6}){7E},
  DB4D: FE 22               ¦¦"     ¦ARI_OP(Fi:cpi{7}){FE},22
  DB4F: 23                  ¦#      ¦inx(Np:HL{2}){23},
  DB50: CA 63 DB            ¦¦c-    ¦JC_OP(Fj:jz{1}){CA},DB63{->@j437}
  DB53: 12                  ¦.      ¦stax(N:DE{1}){12},
  DB54: 13                  ¦.      ¦inx(Np:DE{1}){13},
  DB55: 05                  ¦.      ¦dcr(Nd:B{0}){05},
  DB56: C2 4C DB            ¦TL-    ¦JC_OP(Fj:jnz{0}){C2},DB4C{->@l1433}
@l1435(DB61, DB61):
  DB59: 7E                  ¦~      ¦mov(Nd:A{7}; Ns:M{6}){7E},
  DB5A: B7                  ¦¬      ¦AR_OP(Fa:ora{6}; Ns:A{7}){B7},
  DB5B: C8                  ¦L      ¦RETC_OP(Fr:rz{1}){C8},
  DB5C: FE 22               ¦¦"     ¦ARI_OP(Fi:cpi{7}){FE},22
  DB5E: 23                  ¦#      ¦inx(Np:HL{2}){23},
  DB5F: C8                  ¦L      ¦RETC_OP(Fr:rz{1}){C8},
  DB60: C3 59 DB            ¦+Y-    ¦jmp{C3},DB59{->@l1435}
@j437(DB44, DB51, DB51):
  DB63: F5                  ¦ï      ¦push(Nw:PSW{3}){F5},
  DB64: 3E 20               ¦>      ¦mvi(Nd:A{7}){3E},20
@l1440(DB6A):
  DB66: 12                  ¦.      ¦stax(N:DE{1}){12},
  DB67: 13                  ¦.      ¦inx(Np:DE{1}){13},
  DB68: 05                  ¦.      ¦dcr(Nd:B{0}){05},
  DB69: C2 66 DB            ¦Tf-    ¦JC_OP(Fj:jnz{0}){C2},DB66{->@l1440}
  DB6C: F1                  ¦ë      ¦pop(Nw:PSW{3}){F1},
  DB6D: C9                  ¦r      ¦ret{C9},
```

4. Описание структур данных в исполняемых файлах.

4.1. Описание RTTI Delphi 3.0.

```
type
  ppchar ^pchar
  pstr ^str

  TTypeKind enum (tkUnknown, tkInteger, tkChar, tkEnumeration, tkFloat,
    tkString, tkSet, tkClass, tkMethod, tkWChar, tkLString, tkWString,
    tkVariant,tkArray,tkRecord,tkInterface)

  TTypeInfoData(Kind) forward

  TTypeInfo struc
    TTypeKind Kind
    Str Name
    TTypeInfoData(@.Kind) Inf
  ends

  TOrdType enum (otSByte, otUByte, otSWord, otUWord, otSLong)

  TFloatType enum (ftSingle, ftDouble, ftExtended, ftComp, ftCurr)

  TMethodKind enum (mkProcedure, mkFunction, mkSafeProcedure, mkSafeFunction)

  TIntfFlags set 8 of (ifHasGuid, ifDispInterface, ifDispatch)
  TParamFlags set 8 of (pfVar, pfConst, pfArray, pfAddress, pfReference, pfOut)

  TMethodRec struc
    word size % Of this record
    CodePtr Adr
    str name
  ends

  PMethodTbl ^TMethodTbl
  TMethodTbl struc
    word cnt
    array[@.cnt] of TMethodRec MethodTbl
  ends

  PTClass ^TClass
  TClass ^TTClassData+60;

  PFldClassTbl ^TFldClassTbl
  TFldClassTbl struc
    word cnt
    array[@.cnt] of PTClass ClTbl
  ends

  THFldClass word()

  TFieldRec struc
    ulong ofs
    THFldClass hClass % in FldClassTbl
    str name
  ends

  TFieldTbl(Cnt) array[@:Cnt] of TFieldRec

  PFieldData ^TFieldData
  TFieldData struc
    word cnt
    PFldClassTbl FldClassTbl
```

```
    TFieldTbl(@.cnt) FldTbl
  ends

include WinMsg.rfi

type
  TDynamicKeyTbl struc
    word cnt
    array[@.cnt] of TDelphiCmd DynKeyTbl
  ends

  PDynamicTbl ^TDynamicTbl
  TDynamicTbl struc
    TDynamicKeyTbl Keys
    array[@.Keys.cnt] of CodePtr DynAddrs
  ends

  PPTypeInfo ^PTypeInfo
  PTypeInfo ^TTypeInfo

  TVirtMethodTbl array of CodePtr ?not(IsFixup @)!void;

  TTClassData struc
%    Pointer SelfPtr
    Pointer IntfTable
    Pointer AutoTable
    PTypeInfo InitTable % as record
    PTypeInfo TypeInfo
    PFieldData FieldTable
    PMethodTbl MethodTable
    PDynamicTbl DynamicTable
    pstr ClassName
    ulong InstanceSize
    PTClass Parent
    CodePtr SafeCallException
    CodePtr DefaultHandler
    CodePtr NewInstance
    CodePtr FreeInstance
    CodePtr Destroy
    TVirtMethodTbl VMT
  ends

  TSetInfo struc
    TOrdType OrdType
    PPTypeInfo CompType
  ends

  TIntInfo struc
    TOrdType OrdType
    long MinValue
    long MaxValue
  ends

  TEnumInfo struc
    TOrdType OrdType
    long MinValue
    long MaxValue
    PPTypeInfo BaseType
    case &(@.BaseType^^.Inf)=&@ of
      1: array [@@.MaxValue-@@.MinValue+1] of Str
    endc Names
  ends

  TPropInfo struc
    PPTypeInfo PropType
    CodePtr GetProc
    CodePtr SetProc
```

```
  CodePtr StoredProc
  long Index
  long Default
  int NameIndex
  str Name
ends

TPropData struc
  word Count
  array[@.Count] of TPropInfo Props
ends

TClassInfo struc
  TClass ClassType
  PPTypeInfo ParentInfo
  int PropCount
  str UnitName
  TPropData PropData
ends

TParamInfo struc
  TParamFlags Flags
  str ParamName
  str TypeName
ends

TMethodParamData struc
  Byte Count
  array[@.Count] of TParamInfo ParmTbl
ends

TMethodInfo struc
  TMethodKind MethodKind
  TMethodParamData Parms
  case @.MethodKind of
    mkFunction: str
  endc F
ends

TArrayInfo struc
  ulong ElSize
  ulong ElCnt
  PPTypeInfo ElTypeInf
ends

TRecFieldRec struc
  PPTypeInfo Inf
  ulong Ofs
ends

TRecInfo struc
  ulong SomeThing
  ulong Cnt
  array[@.Cnt] of TRecFieldRec Fields
ends

TBytes8 array[8] of byte
TGUID struc
  ulong D1
  word D2
  word D3
  TBytes8 D4
ends

TInterfaceInfo struc
  PPTypeInfo IntfParent %  ancestor
  TIntfFlags IntfFlags
```

```
    TGUID GUID
    str IntfUnit
    TPropData PropData
  ends

  TTypeInfoData(Kind) case TTypeKind of
    tkUnknown, tkLString, tkWString, tkVariant: void
    tkInteger, tkChar, tkWChar: TIntInfo
    tkEnumeration: TEnumInfo
    tkSet: TSetInfo
    tkFloat: TFloatType
    tkString: Byte
    tkClass: TClassInfo
    tkMethod: TMethodInfo
    tkArray: TArrayInfo
    tkRecord: TRecInfo
    tkInterface: TInterfaceInfo
  endc:[@:Case=@:Kind]

autoname
  TTypeInfo Name _Inf
  TTClassData ClassName^ cl
  TMethodRec Name mr
  TPropInfo Name pi
  PTypeInfo ^.Name _InfP
  TClass ^.ClassName^ P_

type

THFldClass enum THFldClass = (((@:@ as TFieldRec):@ as TFieldTbl):
  @ as TFieldData).FldClassTbl^.ClTbl[@]^^.ClassName^;

TExitProcInfo struc
  Pointer Next
  Pointer SaveExit
  CodePtr Proc
ends

TMemoryManager struc
  CodePtr GetMem % : function(Size: Integer): Pointer;
  CodePtr FreeMem % : function(P: Pointer): Integer;
  CodePtr ReallocMem % : function(P: Pointer; Size: Integer): Pointer;
ends

TExceptRec struc
  TClass EClass
  long EIdent
ends

TExceptMap array[20] of TExceptRec

TExcDescEntry struc
  TClass vTable
  CodePtr handler
ends

TOnExcTable struc
  long Cnt
  array[@.Cnt] of TExcDescEntry ExcTbl
ends

autoname
TExcDescEntry vTable^.ClassName^ On_
```

4.2. Пример описания исполняемого файла.

```
%$IF FileSize=387584;
```

```
include delphi30.rfh

data
0x004384B8 TTypeInfo TMessageFormInfo
0x0044BF68 TTypeInfo TAboutSqlMonitorInfo
0x0044C820 TTypeInfo TSqlMonitorInfo

code
0x402E0C IsClass(VMT: AX; PVMTRq: DX)
%$END FileSize
```