Вычислительные технологии, 2022, том 27, № 3, с. 125–140. © ФИЦ ИВТ, 2022
Computational Technologies, 2022, vol. 27, no. 3, pp. 125–140. © FRC ICT, 2022
ISSN 1560-7534
eISSN 2313-691X

DOI:10.25743/ICT.2022.27.3.010

Алгоритмы и структуры данных для автоматической генерации кода чтения и отображения бинарных данных по спецификациям форматов данных на языке FlexT

А. Е. Хмельнов
Институт динамики систем и теории управления им. В.М. Матросова СО РАН, 664033, Иркутск, Россия
Контактный автор: Хмельнов Алексей Евгеньевич, e-mail: hmelnov@icc.ru

Поступила 28 февраля 2022 г., принята в печать 05 апреля 2022 г.

Язык FlexT разработан для спецификации бинарных форматов данных, изначально — с целью отображения данных в соответствии со спецификацией формата в понятном для человека виде. Обычно следующим шагом после изучения формата является написание кода для работы с ним. Поэтому далее был разработан генератор кода чтения данных, который полностью автоматизирует эту задачу для значительной части форматов данных, описанных на FlexT. Генератор создает модуль чтения данных, а также может создать тестовую программу, демонстрирующую основные способы применения модуля чтения на примере решения задачи отображения всего содержимого файла. В статье рассмотрены основные принципы предлагаемого подхода, а также структуры данных и алгоритмы, используемые при генерации кода.

Ключевые слова: спецификации бинарных форматов данных, декларативный язык, генерация кода, код чтения данных, код отображения данных.

Цитирование: Хмельнов А.Е. Алгоритмы и структуры данных для автоматической генерации кода чтения и отображения бинарных данных по спецификациям форматов данных на языке FlexT. Вычислительные технологии. 2022; 27(3):125–140. DOI:10.25743/ICT.2022.27.3.010.

Введение

Бинарные форматы данных намного эффективнее текстовых с точки зрения как потребления памяти, так и скорости доступа к ним. Основной недостаток бинарных форматов — их непрозрачность: используя простое представление содержимого бинарных файлов (например, в виде шестнадцатеричного дампа), очень сложно определить назначение отдельных байтов и, в частности, найти ошибки в данных.

Для решения проблемы непрозрачности бинарных форматов автором разработан язык спецификации бинарных форматов данных FlexT (от Flexible Types) [1]. Его основными конструкциями являются определения типов данных, которые похожи на определения типов в императивных языках программирования, но при этом являются более гибкими. Так, типы данных FlexT могут иметь параметры и составляющие

переменного размера. Интерпретатор FlexT отображает содержимое бинарного файла в соответствии со спецификацией формата. Это позволяет пользователю проанализировать содержимое файла, дополнительно данные проверяются на соответствие спецификации.

Такая проверка данных может быть очень полезна, например, при разработке кода записи бинарных файлов. Действительно, даже незначительное несоответствие записанных данных собственному формату программы, для которой этот файл предназначен, обычно приводит к тому, что файл не читается. В большинстве случаев программы не показывают никаких диагностических сообщений, которые могли бы пролить свет на природу несоответствий, поскольку конечным пользователям эта информация не нужна. Типичное сообщение об ошибке выглядит как "Несоответствие формату файла" без каких-либо дополнительных подробностей. Спецификацию формата файла на FlexT можно написать таким образом, чтобы она работала с частично корректными файлами и останавливала анализ данных по тому адресу, где есть серьезное несоответствие формату. С другой стороны, примеры данных можно использовать для проверки спецификаций. Действительно, описания форматов файлов на естественном языке обычно содержат неоднозначности, которые можно легко прояснить, быстро проверив, какая из возможных интерпретаций описания позволяет спецификации правильно разобрать имеющиеся примеры данных. Спецификация формата данных на FlexT — это представление информации о формате в чистом виде, без лишних деталей. И это представление легко воспринимается программистами, поскольку язык FlexT является удобочитаемым.

Одной из основных причин, заставляющих программиста изучать формат данных, является необходимость создания кода для работы с этим форматом. Код может создаваться для чтения или записи/изменения данных. Спецификации на FlexT описывают только окончательное размещение данных в соответствующем файле и могут не учитывать такие характеристики формата, как, например, порядок выделения блоков памяти или алгоритмы вычисления значений некоторых вспомогательных полей. Таким образом, спецификация формата на FlexT описывает в основном способ чтения данных, и этой информации может оказаться недостаточно для получения всех необходимых подробностей о правильном способе записи данных для некоторых форматов, поэтому далее рассматривается только задача генерации кода чтения.

Наличие спецификации формата существенно облегчает задачу создания кода для работы с данными. По сути, эта задача уже является достаточно рутинной, однако для сложных форматов программисту все равно придется потратить значительное время на написание больших объемов однотипного кода.

Поскольку спецификация на FlexT содержит всю информацию, которая необходима для написания кода чтения данных, процесс создания такого кода можно автоматизировать. Эту задачу удалось решить для значительной части типов данных FlexT и, как следствие, для большого количества форматов, которые были описаны на этом языке. В данной статье рассматриваются предложенные для ее решения методы генерации кода.

После создания кода для чтения данных необходимо проверить его работоспособность. Для тестирования кода чтения необходимо написать программу, которая будет правильно использовать все возможности сгенерированного модуля для решения некоторой задачи обработки данных. Самая простая, наглядная и универсальная из возможных задач — это отображение (вывод на печать) всех прочитанных данных.

Для сложного формата данных задача создания кода вывода на печать также является довольно трудоемкой (хотя она существенно проще задачи разработки кода чтения данных). После написания вручную нескольких таких программ возникло желание автоматизировать и этот процесс.

Поэтому в дополнение к автоматической генерации кода чтения данных была реализована автоматическая генерация программы печати, которая демонстрирует правильные приемы использования полученного класса считывателя данных. Верное отображение примеров данных программой печати позволяет убедиться в правильности работы сгенерированного считывателя. Кроме того, исходный код такой программы демонстрирует использование сгенерированного модуля. Эти шаблоны кода можно будет далее применять при решении реальных задач.

Конструкция `dispI` в языке FlexT позволяет переопределить способ отображения значения типа данных. Эта информация также может быть использована при создании программы печати. В результате вывод программы будет близок к выводу интерпретатора FlexT.

Ряд проектов [2–10] решают задачу генерации кода по спецификациям бинарных форматов данных. В статье [11] выполнен обзор возможностей этих проектов, поэтому не будем повторять его здесь. Обзор показал, что основными преимуществами предлагаемого подхода являются: более широкие возможности и выразительность типов данных языка FlexT, более высокая эффективность сгенерированного кода за счет использования в нем статических типов данных целевого языка в тех случаях, когда это возможно, а также за счет использования чтения данных по мере необходимости.

В статье [11] рассмотрена специфика трансляции различных типов данных генератором кода, рассмотрены методы трансляции битовых типов данных, а также методы трансляции выражений FlexT на целевой язык. В настоящей статье после краткого описания возможностей языка FlexT, которое необходимо, чтобы сделать ее самодостаточной, рассмотрены: основные принципы, используемые при разработке алгоритмов генерации кода; структуры данных, служащие для промежуточного представления сгенерированного императивного кода; алгоритмы генерации кода тестового приложения и, наконец, результаты применения генератора для небольшого формата данных.

1. Типы данных языка FlexT

Основными конструкциями FlexT являются определения типов. Наиболее важные части спецификации формата данных — разделы `type`, в которых определяются типы данных формата, и разделы `data`, которые определяют *переменные* — элементы данных с указанием их адресов и типов. Синтаксис определений типов FlexT подобен традиционному синтаксису аналогичных конструкций императивных языков программирования. В отличие от императивных языков программирования, размер в памяти различных значений одного и того же типа данных FlexT может различаться. Будем называть *динамическими* типы данных с переменным размером и/или внутренней структурой их значений и *статическими* — типы данных постоянного размера с фиксированной внутренней структурой.

Типы данных FlexT обладают рядом свойств. Значения свойств определяют размер и другие характеристики данных. Свойство `Size` является общим для всех типов, оно определяет размер элемента данных. Размер может быть автоматически рассчитан по

Т а б л и ц а 1. Типы данных FlexT
Table 1. The FlexT data types

Тип	Пример	Описание/назначение
Целые*	`num-(6)`	Различаются размером и наличием знака
Пустой*	`void`	Тип с размером 0, отмечает место в памяти
Символы*	`char, wchar, wcharr`	В выбранной кодировке или Unicode с разными порядками байтов
Перечислимый*	`enum byte (A=1,B,C)`	Задает наименования констант базового типа данных
Перечисление термов	`enum TBit8 fields( R0: TReg @0.3,...) of( rts(R0) = 00020_,...)`	Описание кодирования машинных команд с указанием битовых полей, наличие которых определяется значениями оставшихся битов
Множество*	`set 8 of ( OLD ^ 0x02, ...)`	Дает наименование битам, биты могут задаваться своим номером (символ "=" после имени) или маской (символ "^" после имени)
Запись*	`struc Byte Len array[@.Len]of Char S ends`	Последовательное размещение в памяти именованных и, возможно, разнотипных составляющих
Вариант*	`case @.Kind of vkByte: Byte else ulong endc`	Выбор типа содержимого по заданной в параметрах (внешней) информации
Проверка*	`try FN: TFntNum Op: TDVIOp endt`	Выбор типа содержимого по внутренней информации (первый тип, удовлетворяющий условиям корректности)
Массив*	`array[@.Len]of str` `array of str ?@[0]= 0!byte;`	Последовательное размещение однотипных составляющих в памяти (размеры которых могут различаться). Может задаваться количество элементов, общий размер массива или стоп-условие
Сырые данные*	`raw[@.S]`	Неинтерпретируемые данные, отображаются как hex-дамп
Выравнивание*	`align 16 at &@;`	Пропуск неиспользуемых данных для выхода на кратное заданному значению смещение относительно базового адреса
Указатель	`^TTable near=DWORD, ref=@:Base+@;`	Использует значение базового типа для задания адреса данных указанного типа в памяти (для файлов — смещения от начала)
Предварит. объявление*	`forward`	Используется при циклических зависимостях между типами данных
Машин. команды	`codes of TOpPDP ?(@.Op >=TWOpCode.br)and...;`	Дизассемблирование машинных команд

* Поддерживается генератором кода чтения.

содержимому сложного типа данных или по значениям других свойств этого типа, также он может быть указан явно с помощью соответствующего параметра типа данных. Помимо свойства `Size` некоторые виды типов данных могут иметь другие свойства, например, массивы характеризуются количеством своих элементов, а вариантные типы зависят от выбранного номера варианта. Для описания информационных связей между элементами данных используются параметры типов данных.

Основные типы данных языка FlexT показаны в табл. 1. Большинство типов данных имеют и битовые версии, размеры которых измеряются в битах, а адрес в памяти задается битовым указателем. Целочисленные типы данных и некоторые другие типы, использующие целочисленное внутреннее представление (символы, указатели), а также все битовые типы данных зависят от выбранного порядка байтов (LSB или MSB).

2. Блоки с дополнительной информацией о типе данных

Ко всем определениям типов данных могут присоединяться блоки с дополнительной информацией при помощи символа соединения "`:`". Блоки дополнительной информации, используемые в языке FlexT, перечислены в табл. 2.

Блок утверждений описывает информационные зависимости внутри типа данных, определяя значения свойств типа данных и еще не заданных параметров его составляющих с использованием выражений над значениями параметров типа, а также значениями и уже известными параметрами и свойствами составляющих. Блок состоит из серии присваиваний. Левая часть каждого присваивания может быть квалификатором свойства определяемого типа (как в первом утверждении примера, где задается значение свойства `Size`) или квалификатором параметра составляющей (как во втором утверждении примера, где задается значение параметра `Cnt` поля `offset`). Правая часть присваивания — выражение, которое задает значение свойства или параметра составляющей.

Т а б л и ц а 2. Блоки дополнительной информации
Table 2. Extra information blocks

Тип блока	Описание и пример
Блок утверждений	Позволяет задать значения свойств типа данных или параметров, его составляющих, с использованием параметров типа или значение, уже известное информации о его составляющих `:[@:Size=@.Len, @.offset:Cnt=@.count]`
Блок условий корректности	Задает логическое условие, которое должно выполняться для данных, относящихся к этому типу `:assert[@.Op>=0x80]`
Блок отображения	Позволяет переопределить способ отображения значений типа `:displ=('#',HEX(2*@))`
Блок именования	Аналогичен `displ`, но служит для автоматического задания имен переменных `:autoname=('sec_',@.tag)`
Блок добавочного свойства	Добавляет вычислимое свойство типа `:let Val=(@.0)exc(@.1)`

Блок условий корректности используется для проверки соответствия элемента данных спецификации. Проверки соответствия выполняются: для выбора типа внутреннего содержимого в типе данных *проверка* (`try`); в стоп-условиях массивов, а также в блоках `assert` спецификации формата (т. е. условиях корректности спецификации). Блок содержит логические выражения, которые должны быть истинными для любого правильного значения определяемого типа данных.

Блоки добавочных свойств типа позволяют описать часто используемые числовые выражения над атрибутами, свойствами и значениями самого типа и/или его составляющих. После этого можно ссылаться на выражение по его имени в других выражениях, как если бы оно было свойством типа. Эта возможность упрощает написание спецификаций для форматов, которые, например, используют типы данных переменного размера, кодирующие числовые значения разной величины. В этом случае дополнительное свойство позволяет получить закодированное значение, а выражение для этого свойства описывает метод декодирования.

Блок отображения позволяет изменить способ отображения значений типа данных. Блок поддерживает ряд команд отображения, которые могут использоваться для указания метода вывода как для простых скалярных данных, так и для более сложных типов данных, таких как массивы или варианты.

3. Основные принципы генерации кода чтения данных

Для текстовых форматов данных, использующих синтаксис XML, существуют генераторы кода, которые автоматически создают библиотеку классов для чтения и изменения данных в этом формате с использованием XML-схемы формата (привязка данных XML [12]). Генератор кода чтения бинарных данных должен давать подобный результат по спецификации на FlexT.

При разработке генератора кода следует уделить большое внимание качеству получаемых исходных текстов. Код должен быть высокого качества, чтобы программисты могли его понять и при необходимости доработать. Более того, генератор кода может реализовать для нас все возможности, которые хотелось бы иметь, включая те, которые было бы слишком трудоемко писать вручную.

Следующее требование — код чтения данных должен поддерживать произвольный доступ и не должен ограничиваться только последовательной обработкой данных. При этом необходимо минимизировать потребление памяти: если какой-либо тип данных FlexT (например, статический массив или запись) может быть представлен типом данных целевого языка программирования, то эту возможность необходимо использовать для исключения накладных расходов при работе со статическими данными. Память для вспомогательных структур данных, необходимых для доступа к более сложным элементам данных, должна выделяться по необходимости, а не вся сразу без учета того, какие из этих элементов данных реально используются в программе.

Для выполнения этих требований приняты следующие технические решения.

Использование FileMapping. Генерируемая программа использует отображение содержимого файла в оперативную память (FileMapping). Преимущество отображения в память заключается в том, что, хотя каждый байт файла сразу получает свой адрес в памяти, соответствующие данные считываются туда операционной системой по мере необходимости, лишь когда программа попытается получить к ним доступ. Таким образом, если, например, конкретное приложение использует только небольшую часть

данных, то ненужное чтение всего файла в память выполняться не будет. Недостатком этого подхода для 32-битных приложений является то, что максимальный размер данных, которые могут быть обработаны сгенерированной программой, ограничен четырьмя гигабайтами 32-битного адресного пространства. На самом деле ограничение по размеру еще жестче и зависит от текущей фрагментации памяти в адресном пространстве программы. Однако для большинства форматов данных такое ограничение по размеру не критично в силу типичных размеров файлов. Заметим, что сам интерпретатор FlexT также использует FileMapping.

Использование указателей на сопоставленную файлу память для статических типов данных. Для доступа к значению статического типа данных FlexT, представленного типом данных целевого языка, достаточно получить указатель на то место в созданном отображением файла блоке памяти, где расположен интересующий элемент данных.

Использование классов доступа для динамических типов. Для более сложных динамических типов данных FlexT используются экземпляры вспомогательных классов для доступа к их данным. Вспомогательный класс может хранить значения параметров и свойств типа FlexT и при необходимости ссылки на его составляющие. Например, для массива переменного размера требуется хранить его размер или количество элементов. Если размер элементов массива не фиксирован, то также потребуется хранить таблицу адресов этих элементов. Будем использовать термин "класс доступа" (`DataAccessor`) для этих вспомогательных классов.

Значения и адреса составляющих. Класс доступа для сложного составного типа данных должен иметь методы доступа к его составляющим. Для всех составляющих должны быть реализованы методы получения указателя на их данные (адреса данных). Также требуются методы, возвращающие значения составляющих. А именно: для простого числового типа данных значением является представленное им число; для статического составного типа данных, представимого типом целевого языка, значением является указатель на его данные (здесь можно использовать тот же метод получения адреса данных); для сложного типа данных, у которого есть класс доступа, значением является экземпляр этого класса доступа.

Главный класс модуля чтения данных. Для конкретного формата файла генератор кода создает `DataReader` — класс считывателя. Конструктор класса `DataReader` вызывается со строковым параметром, задающим имя файла, который надо открыть. Экземпляр `DataReader`, созданный конструктором, обеспечивает доступ ко всем структурам данных, определенным в спецификации формата. Для каждого элемента данных верхнего уровня (переменной), определенного в спецификации, `DataReader` содержит методы для доступа к его адресу и значению (значение определяется так же, как и для элементов составных типов). Имена методов доступа к элементам данных формируются из соответствующих имен элементов данных.

Использование общих модулей в сгенерированном коде. Для упрощения генерируемого кода в нем используются базовые классы из модуля `FmtSys` — основного модуля системной библиотеки для чтения данных. Таким образом, чтобы реализовать классы доступа (`DataAccessor`) и главный класс считывателя (`DataReader`) для формата данных, генератор кода выбирает наиболее подходящие базовые классы, для которых требуется определить и переопределить только часть из необходимых для работы методов. Модуль `FmtSys` должен быть портирован на каждый поддерживаемый целевой язык программирования.

Декодирование нестандартных целочисленных значений. Для типа данных, зависящего от порядка байтов, заданный порядок байтов может отличаться от порядка байтов компьютера. Текущая реализация генератора кода предполагает, что целевой компьютер имеет порядок байтов LSB. Таким образом, простые числовые типы с порядком LSB обычно могут быть напрямую представлены соответствующими типами целевого языка программирования. Исключение составляют целочисленные типы данных нестандартных размеров (3, 5, 6, 7 байтов) — таких типов, как правило, нет в императивных языках. Таким образом, для всех целочисленных типов данных с порядком байтов MSB и типов нестандартных размеров с порядком LSB используется общий подход: модуль `FmtSys` содержит готовые реализации типов данных для их представления — это классы без виртуальных методов (и, следовательно, без указателя на таблицу виртуальных методов в памяти экземпляра класса) с экземплярами соответствующего размера, у которых есть метод `Value`, возвращающий представленное типом целочисленное значение.

Таким образом, декодирование значения в методе `Value` выполняется заново при каждом обращении к нему. Конечно, такой подход несколько снижает скорость работы генерируемого кода. Однако за счет этого становится возможным напрямую использовать такие типы данных в качестве типов составляющих у составных типов данных целевого языка, представляющих статические типы данных FlexT. При этом удается обойтись без создания ненужных классов доступа для всех составных статических типов данных FlexT независимо от наличия у них составляющих с нестандартными размерами и порядком байтов.

Значения статических битовых типов данных. Для работы с битовыми типами данных используются указатели на биты (битовые указатели). Битовый указатель помимо адреса байта содержит номер бита внутри этого байта. Эта информация не помещается в обычный указатель. Поэтому для работы с битовыми указателями в системном модуле чтения данных `FmtSys` определен специальный простой класс `TBitsPtr`, в котором реализована адресная арифметика для битовых указателей, а также определены методы доступа к битовым данным, на которые указатель ссылается. В языке FlexT интерпретация номера бита в байте зависит от порядка байтов: для порядка LSB биты нумеруются от младшего к старшему, а для порядка MSB — от старшего к младшему. Сами битовые указатели не содержат никакой информации о порядке байтов, вместо этого у них есть методы доступа к памяти для обоих порядков байтов, и подходящий метод выбирается для реализации определенного типа данных, когда требуемый порядок байтов уже известен. Для статических битовых типов данных создаются облегченные классы доступа, которые являются потомками `TBitsPtr` (и занимают столько же памяти, так как не хранят никакой дополнительной информации). Экземпляры облегченных классов создаются по мере необходимости в методах, возвращающих значения таких типов. Для динамических битовых типов данных по-прежнему требуются классы доступа.

Использование промежуточного представления кода. При генерации кода сначала создается универсальное (не зависящее от конкретного целевого языка программирования) промежуточное представление программы, которое используется на следующем этапе для генерации кода на целевом языке. Структуры данных промежуточного представления выбираются так, чтобы их можно было представить на всех поддерживаемых целевых языках.

4. Промежуточный код

Использование промежуточного кода позволяет писать алгоритмы генерации независимо от специфики конкретного целевого языка программирования. В качестве промежуточного кода можно было бы разработать новый язык программирования и сначала генерировать тексты на этом языке. Однако впоследствии в любом случае необходимо будет преобразовать этот текст во внутреннее представление, чтобы перевести его на целевой язык. В результате процесс генерации кода будет иметь несколько лишних этапов.

Вместо создания нового языка разработаны функции и методы для внутреннего представления промежуточного кода таким образом, чтобы их вызовы, описывающие используемые шаблоны кода, визуально приближались к коду на промежуточном языке, который было решено не разрабатывать. Для достижения визуального сходства используются цепочки вызовов методов и динамические массивы в параметрах подпрограмм. Имена всех функций и констант для создания промежуточного кода начинаются с префикса `il`, который является сокращением для словосочетания Intermediate Language (или, как вариант, Imperative Language).

Чтобы проиллюстрировать этот подход, рассмотрим фрагмент кода на Паскале, который генерирует метод доступа к данным класса (листинг 1). Здесь в шаблон кода подставляются ряд выражений (переменные с именами, содержащими `Expr`) и список операторов, на которые ссылается поле `Ops` вспомогательной структуры данных `VCvtInfo`.

Здесь функция `ilBlock` создает список операторов путем объединения нескольких операторов (или их списков). Операторы блока задаются открытым массивом — аргументом функции `ilBlock` (т. е. любое их число через запятую записывается в квадратных скобках). Функция `ilIf` создает условный оператор. Она имеет единственный параметр — логическое условие и возвращает экземпляр класса `TIfOperator`, у которого есть методы `ilThen` и `ilElse`. Каждый из этих методов принимает открытый массив операторов, составляющих соответствующую ветвь условного оператора, и возвращает тот же экземпляр класса `TIfOperator`, для которого он был вызван. Возврат экземпляра класса позволяет создавать цепочки вызовов методов (method chaining). Функция `ilLet` создает оператор присваивания, она принимает два аргумента, которые являются выражениями для левой и правой частей оператора. Функция `ilAssigned` генерирует логическое выражение, проверяющее, что указатель не пуст. Далее логическое

Листинг 1. Генерация метода с использованием промежуточного представления
Listing 1. Generation of a method using intermediate representation

```
TClassMethodExpInfo.Create(Cl,
  CatPrefix('Get',RName),mvProtected,Nil{Parms},
  hRangeType or hdtPtr,[]{Flags},VCvtInfo.Vars,
  ilBlock([
    ilIf(ilAssigned(FldExpr).UnOp(ilNot)).ilThen([
      ilLet(PrevFldExpr,PrevExpr),
      VCvtInfo.Ops,
      ilLet(FldExpr,PrevFldExpr.BOp(ilPtrAdd,FldExpr1)
          )]),
    ilRet(FldExpr)])
);
```

значение инвертируется с помощью метода `UnOp` для выполнения унарной операции `ilNot`. Функция `ilRet` создает оператор возврата из функции.

Для представления выражений используется иерархия классов с корневым классом `TExpr`, а для представления операторов — иерархия классов с корнем `TOperator`. Для классов выражений применяется подсчет ссылок, что позволяет использовать один и тот же экземпляр выражения в нескольких местах промежуточного кода. Например, выражение `FldExpr` в листинге 1 используется трижды.

При выборе операторов для промежуточного представления рассматривалось пересечение наборов возможностей целевых языков программирования, которые предполагается поддержать. Например, нельзя использовать множественное наследование, поскольку в большинстве языков программирования оно не поддерживается (исключение — C++).

Приведем и обратный пример (когда пришлось адаптироваться к C++). В стандарте C++ нет аналога оператора `try ... finally`. Подобный оператор поддерживается некоторыми компиляторами, например Visual C++, но не стандартом языка. Более того, в некоторых дискуссиях по этому поводу программисты, знающие только C++, просто не могут понять, зачем может понадобиться такой оператор.

Оператор `try ... finally` используется в основном для того, чтобы гарантировать высвобождение некоторых ресурсов независимо от возможных ошибок в коде, который он защищает. Поэтому в документации по Паскалю этот оператор часто называется блоком защиты ресурсов. При генерации кода чтения данных `try ... finally` требуется использовать именно в этих целях. Поэтому вместо оператора `try ... finally` в промежуточном представлении был реализован оператор защиты ресурсов, который можно создать с помощью вызова функции `ilResourceGuard`. Функция имеет три аргумента: имя переменной; выражение, создающее объект; открытый массив операторов, использующих объект, которые должны быть защищены.

При переводе оператора защиты ресурсов на Паскаль код генерируется с помощью оператора `try ... finally` (листинг 2). А для C++, чтобы гарантировать, что объект будет освобожден, используется шаблон `std::unique_ptr`. Код C++, соответствующий тому же оператору промежуточного языка, показан в листинге 3.

Помимо синтаксиса и набора поддерживаемых в языке операторов важной частью реализации языка программирования является его системная библиотека. При генерации промежуточного кода необходимо учитывать различия в системных библиотеках поддерживаемых целевых языков. Чтобы учесть эти различия, используются виртуальные методы генераторов кода для целевых языков. Такие методы по-разному генериру-

Листинг 2. Перевод оператора защиты ресурсов на Паскаль
Listing 2. Translation of the resource guard operator into Pascal

```
Reader := TSHPReader.Create(FN);
try
  <Operators>
finally
  Reader.Free;
end;
```

Листинг 3. Перевод оператора защиты ресурсов на C++
Listing 3. Translation of the resource guard operator into C++

```
std::unique_ptr<TSHPReader>
  must_free_Reader(
    new TSHPReader(FN));
Reader = must_free_Reader.get();
<Operators>
```

ют промежуточные фрагменты кода для ряда подзадач, решаемых с использованием системных библиотек.

Например, виртуальный метод `GenProgramArgExpr` возвращает выражение для получения значения параметра командной строки программы. У генератора кода на Паскале этот метод создает выражение `ilFCall('ParamStr',[NdxP])`, а у генератора кода на C++ — выражение `ilV0x('argv').Fetch[NdxP]`. Здесь выражение `NdxP` представляет индекс аргумента. Таким образом, для Паскаля вызывается системная функция `ParamStr`, а для C++ происходит обращение к элементу массива `argv` функции `main` (тем самым область применения выражения ограничивается телом функции `main`, но здесь надо учесть, что в генерируемом коде обращение к аргументам программы и происходит только в этом контексте).

Генераторы фрагментов работают на этапе построения промежуточного кода. В результате создаваемый промежуточный код частично зависит от целевого языка. Основное преимущество этого подхода состоит в том, что не требуется расширять набор операторов промежуточного представления для поддержки использования различных возможностей системных библиотек.

Все модули, описывающие промежуточное представление кода, не зависят от модулей интерпретатора FlexT. Это позволяет использовать их и для других задач, требующих генерации кода.

5. Генерация тестового приложения

Для проверки автоматически созданного кода чтения данных разработаны алгоритмы генерации кода тестового приложения. Тестовое приложение предназначено для проверки работоспособности кода чтения данных при решении задачи отображения содержимого бинарного файла. Реализованы два стиля генерации тестового кода: с использованием подпрограмм для печати значений типов данных и без использования таких подпрограмм. Использование подпрограмм печати увеличивает размер кода и делает его менее понятным, но без них нельзя обойтись, когда спецификация имеет рекурсивные зависимости между типами данных.

Для создания кода вывода на печать значений типа данных алгоритмы могут использовать информацию из блока отображения (`displ`) этого типа данных. Использование этой информации при генерации кода печати позволяет получить программу, которая отображает бинарные данные в близком к тому, как это делает интерпретатор FlexT, виде, но без использования библиотек интерпретатора.

6. Пример использования

В качестве небольшой иллюстрации возможностей разработанного генератора кода рассмотрим очень простой формат файла STL [13] (от Stereo Litography), который, несмотря на его простоту, широко используется в 3D-печати.

Полное описание формата файла STL на FlexT приведено в листинге 4. Файл состоит из небольшого заголовка, содержащего 80 произвольных символов, и целого числа, задающего количество треугольников у 3D-объекта, за которым следует массив записей для граней (треугольников) этого объекта. Также существует текстовая версия формата STL, которая, к сожалению, имеет такое же расширение `.stl`. К счастью, текстовые

файлы STL всегда должны начинаться с идентификатора 'solid', поэтому конструкция assert в спецификации служит для того, чтобы отклонять текстовые файлы STL. Поскольку у двоичных файлов STL нет сигнатуры, вторая конструкция assert служит для того, чтобы дополнительно проверить файлы на соответствие формату, сопоставляя фактический размер файла и указанное количество граней.

Интерфейсная часть сгенерированного на Паскале модуля чтения данных приведена в листинге 5. Код начинается с определений типов данных, которые непосредственно представляют статические типы данных FlexT из спецификации STL (ulong, THdr0, TSTLPoint, TSTLFaceVertex, TSTLFace, THdr1). Некоторые типы данных из этого списка были анонимными в спецификации, и генератор кода назвал их, используя имена соответствующих переменных (THdr0, THdr1) или полей (TSTLFaceVertex). Единственный динамический тип данных в спецификации — это тип переменной Faces. Генератор кода создал класс доступа TTFacesAccessor для этого типа данных. Поскольку массив Faces имеет статические элементы, его класс доступа наследуется от класса TSimpleArrayAccessor, определенного в модуле FmtSys. Сгенерированный класс чтения данных TSTLReader имеет методы для получения адресов всех переменных из спецификации (Hdr0, Hdr1, CountPtr, FacesPtr). Эти указатели также являются значениями части переменных (Hdr0, Hdr1). У класса чтения данных есть методы Count для получения значения соответствующей целочисленной переменной и Faces для получения экземпляра класса доступа к этому динамическому массиву.

Листинг 6 содержит код тестовой программы, созданной на C++. Программа сгенерирована без использования процедур печати, поскольку формат STL не имеет рекурсивных зависимостей между типами данных. Обратите внимание на использование переменной must_free_Reader. Это шаблон кода, применяемый для перевода оператора ilResourceGuard в C++, здесь он защищает объект Reader. Последний оператор листинга будет заменен в Паскале вызовом Readln — это еще один пример генерации фрагмента кода для целевого языка.

Листинг 4. Полная спецификация бинарного формата файлов STL на FlexT
Listing 4. The complete binary STL file format specification in FlexT

```
data
0 array[5] of char Hdr0

assert not(Hdr0='solid'); //ignore text STL
include Float.rfi

type
TSTLPoint array[3]of TSingle

TSTLFace struc
  TSTLPoint Normal //Normal vector
  array[3]of TSTLPoint Vertex
  Word Attr //Attribute byte count
ends

data
5 array[75] of char Hdr1
80 ulong Count

assert 84+Count*TSTLFace:Size=FileSize;

data
84 array[Count] of TSTLFace Faces
```

Листинг 5. Интерфейсная часть модуля чтения файлов STL на Паскале
Listing 5. Interface part of the STL file format reading module generated in Pascal

```
unit STL;
interface
uses
  SysUtils, FmtSys;

type
  PUlong = ^ulong;
  ulong = LongWord;
  PHdr0 = ^THdr0;
  THdr0 = array[0..4]of AnsiChar;
  PSTLPoint = ^TSTLPoint;
  TSTLPoint = array[0..2]of Single;
  PSTLFaceVertex = ^TSTLFaceVertex;
  TSTLFaceVertex = array[0..2]of TSTLPoint;
  PSTLFace = ^TSTLFace;
  TSTLFace = packed record
    Normal: TSTLPoint;
    Vertex: TSTLFaceVertex;
    Attr: word;
  end;

  PHdr1 = ^THdr1;
  THdr1 = array[0..74]of AnsiChar;
  PFaces = ^TFaces;
  TFaces = array[0..0]of TSTLFace;

  TTFacesAccessor=class(TSimpleArrayAccessor)
   protected
    class function GetItemSize: Integer; override;
    function GetTbl: PFaces;
   public
    function Fetch(AIndex: Integer): PSTLFace;
    constructor Create(AOwner: TComplexDataAccessor;
      AOfs: TOffset; AIndex: Integer; Ap_Count:
          Integer);
    property Tbl: PFaces read GetTbl;
    property p_Count: Integer read GetCount;
  end;

  TSTLReader = class(TBaseDataReader)
   protected
    FHdr0Ptr: PHdr0;
    FHdr1Ptr: PHdr1;
    FCountPtr: PUlong;
    FFacesPtr: PFaces;
    FFaces: TTFacesAccessor;
    procedure SetItem(AIndex: Integer;
      V: TDataAccessor); override;
    function GetItem(AIndex: Integer):
      TDataAccessor; override;
    function GetItemCount: Integer; override;
   public
    function Hdr0: PHdr0;
    function Hdr1: PHdr1;
    function Count: LongWord;
    function Faces: TTFacesAccessor;
    constructor Create(const AFileName:String);
    property CountPtr: PUlong read FCountPtr;
    property FacesPtr: PFaces read FFacesPtr;
  end;

function THdr0ToStr(V: PHdr0): String;
function THdr1ToStr(V: PHdr1): String;
```

Листинг 6. Тестовая программа, сгенерированная для формата STL на C++
Listing 6. Test program generated for STL file format in C++

```
#include <typeinfo>
#include <iostream>
#include <memory>
#include "FmtSys.h"
#include "STL.h"
#pragma hdrstop

using namespace std;
using namespace STL;

int main(int argc, char* argv[]){
  TSTLReader * Reader;
  std::string FN;
  int i;
  int i0;
  int i1;
  int i2;
  PSTLPoint V;
  PSTLFace V0;
  if (argc-1<=0) {
    cout<<"Usage:"<<endl;
    cout<<"␣␣"<<extractFileName(argv[0])<<
      "␣<STL␣file>"<<endl;
    exit(1);
  }
  FN = argv[1];
  try {
    {
      std::unique_ptr<TSTLReader>
        must_free_Reader(new TSTLReader(FN)
            );
      Reader = must_free_Reader.get();
      cout<<"Hdr0:␣"<<THdr0ToStr(Reader->
          Hdr0())<<endl;
      cout<<"Hdr1:␣"<<THdr1ToStr(Reader->
          Hdr1())<<endl;
      cout<<"Count:␣"<<Reader->Count()<<
          endl;
      cout<<"Faces:"<<endl;
      for (i=0; i<Reader->Faces()->Count();
           i++) {
        V0 = Reader->Faces()->Fetch(i);
        cout<<"␣␣["<<i<<"]:"<<endl;
        cout<<"␣␣␣␣Normal:"<<endl;
        for (i0=0; i0<3; i0++)
          cout<<"␣␣␣␣␣␣["<<i0<<"]:␣"<<
            V0->Normal[i0]<<endl;
        cout<<"␣␣␣␣Vertex:"<<endl;
        for (i1=0; i1<3; i1++) {
          V = &V0->Vertex[i1];
          cout<<"␣␣␣␣␣␣["<<i1<<"]:"<<endl;
          for (i2=0; i2<3; i2++)
            cout<<"␣␣␣␣␣␣␣␣["<<i2<<"]:␣"<<
                *V[i2]<<endl;
        }
        cout<<"␣␣␣␣Attr:␣"<<V0->Attr<<endl;
      }
    }
  } catch(const std::exception& E) {
    cout<<E.what()<<endl;
  }
  std::cin.ignore(0x100,'\x0A');
}
```

Заключение

Хотя показанный пример не смог проиллюстрировать все возможности генератора кода (например, отсутствовали вариантные или битовые типы данных, а также блоки `displ`), он позволяет сопоставить размер текста и сложность понимания у исходной спецификации формата на FlexT и сгенерированного кода. Пример также позволяет подумать: насколько человек-программист захотел бы изменить эти программы, если бы ему пришлось писать код вручную. Например, человек, скорее всего, отказался бы от вспомогательного типа `TTFacesAccessor`. В результате его программа была бы немного эффективнее, но при этом связь между полем `Count` и количеством элементов в массиве `Faces` стала бы менее очевидной.

Текущая версия генератора кода позволяет автоматически создавать код чтения данных для таких форматов файлов, как ArcView Shape (`.shp`), или файла классов Java (`.class`). Более того, тестовый код, автоматически сгенерированный для файлов классов Java, показывает данные очень детально, вплоть до вывода машинных команд JVM в телах методов.

Необходимо отметить, что работа по созданию алгоритмов генерации кода для языка FlexT еще не завершена. Типы данных языка FlexT, для которых уже реализована генерация кода чтения, помечены сноской в табл. 1. Также пока не поддерживаются некоторые виды выражений и операций отображения. Важное ограничение текущей версии генератора — невозможность учесть условную компиляцию спецификаций, которая выполняется в ходе обработки конкретных данных. Этот механизм позволяет очень компактно записать, например, всю историю развития некоторого формата в одном файле, но пока не удается предложить общий способ создания кода на целевом языке, учитывающего все версии формата с возможными изменениями в нем состава и типов полей записей или значений констант перечислимых типов.

Благодарности. Работа выполнена в рамках гранта № 075-15-2020-787 Министерства науки и высшего образования РФ для выполнения крупного научного проекта по приоритетным направлениям научно-технологического развития (проект “Фундаментальные основы, методы и технологии цифрового мониторинга и прогнозирования экологической обстановки Байкальской природной территории”).

Список литературы

[1] **Хмельнов А.Е., Бычков И.В., Михайлов А.А.** Декларативный язык FlexT — инструмент анализа и документирования бинарных форматов данных. Труды ИСП РАН. 2016; 28(5):239–268. DOI:10.15514/ISPRAS-2016-28(5)-15.

[2] **Risso F., Baldi M.** NetPDL: an extensible XML-based language for packet header description. Computer Networks. 2006; 50(5):688–706. DOI:10.1016/j.comnet.2005.05.029.

[3] **Morandi O., Risso F., Baldi M., Baldini A.** Enabling flexible packet filtering through dynamic code generation. 2008 IEEE International Conference on Communications. 2008; 5849–5856. DOI:10.1109/ICC.2008.1094. Адрес доступа: https://www.semanticscholar.org/paper/Enabling-Flexible-Packet-Filtering-Through-Dynamic-Morandi-Risso/54b6b04c0194e6a7a4c87f64049289ce1a3b09fd.

[4] BinPAC. Адрес доступа: https://github.com/zeek/binpac (дата обращения 1.12.2021).

[5] **Calder B.R., Masetti G.** Huddler: a multi-language compiler for automatically generated format-specific data drivers. U.S. Hydro 2015. Gaylord Hotel, National Harbor, Maryland, USA. Адрес доступа: https://www.researchgate.net/publication/277302751_HUDDLER_a_multi-language_compiler_for_automatically_generated_format-specific_data_drivers.

[6] **Underwood W.** Grammar-based specification and parsing of binary file formats. The International Journal of Digital Curation. 2012; 7(1):95–106. Адрес доступа: http://www.ijdc.net/index.php/ijdc/article/viewFile/207/276.

[7] **Parr T.** ANTLR (ANother Tool for Language Recognition). Адрес доступа: http://antlr.org (дата обращения 01.12.2021).

[8] **Back G.** DataScript — A specification and scripting language for binary data. Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02). London, UK: Springer-Verlag; 2002: 66–77. Адрес доступа: http://people.cs.vt.edu/~gback/papers/gback-datascript-gpce2002.pdf.

[9] **Back G.** DataScript. Адрес доступа: http://datascript.sourceforge.net (дата обращения 01.12.2021).

[10] Kaitai Struct. Адрес доступа: http://kaitai.io (дата обращения 01.12.2021).

[11] **Hmelnov A., Mikhailov A.** Generation of code for reading data from the declarative file format specifications written in language FlexT. 2018 Ivannikov Ispras Open Conference (ISPRAS). Moscow; 2018: 23–30. DOI:10.1109/ISPRAS.2018.00011.

[12] **Bourret R.** XML data binding resources. Адрес доступа: http://www.rpbourret.com/xml/XMLDataBinding.htm (дата обращения 01.12.2021).

[13] **Burns M.** The STL format — standard data format for Fabbers. Адрес доступа: http://www.fabbers.com/tech/STL_Format (дата обращения 01.12.2021).

Вычислительные технологии, 2022, том 27, № 3, с. 125–140. © ФИЦ ИВТ, 2022 ISSN 1560-7534
Computational Technologies, 2022, vol. 27, no. 3, pp. 125–140. © FRC ICT, 2022 eISSN 2313-691X

INFORMATION TECHNOLOGIES

DOI:10.25743/ICT.2022.27.3.010

Algorithms and data structures for automatic generation of binary data reading and printing code from the data format specifications in the language FlexT

Hmelnov Alexey E.

Matrosov Institute for System Dynamics and Control Theory of SB RAS, 664033, Irkutsk, Russia
Corresponding author: Hmelnov Alexei Evgenievich, e-mail: hmelnov@icc.ru
Received February 28, 2022, accepted April 05, 2022.

Abstract

The language FlexT (acronym for Flexible Types) is designed for specification of binary data formats. Its main statements are data type definitions that resemble type definitions of imperative programming languages, but are more flexible. For example, the FlexT data types may contain subparts of variable size and may have parameters.

The primary purpose of the FlexT interpreter is to display the binary data in accordance with the format specification in a human-readable form. Typically the next step after studying some data format is to write a code for its processing. That's why we have developed the data reading

code generator, which completely automates this task for the substantial part of the data formats described in FlexT. It can generate both the data reading module and the test program. The program demonstrates the correct usage of the module for solving the data visualization task. In the paper we consider the main principles of our approach to code generation, the data structures that we use for representation of information about the generated code and the code generation algorithms.

Keywords: specifications of binary data formats, declarative language, code generation, data reader, data visualization code.

Citation: Hmelnov A.E. Algorithms and data structures for automatic generation of binary data reading and printing code from the data format specifications in the language FlexT. Computational Technologies. 2022; 27(3):125–140. DOI:10.25743/ICT.2022.27.3.010. (In Russ.)

Acknowledgements. The work was supported by the Ministry of Science and Higher Education of the Russian Federation, the grant № 075-15-2020-787 for implementation of Major scientific projects on priority areas of scientific and technological development (the project "Fundamentals, methods and technologies for digital monitoring and forecasting of the environmental situation on the Baikal natural territory").

References

1. **Hmelnov A.Y., Bychkov I.V., Mikhailov A.A.** A declarative language FlexT for analysis and documenting of binary data formats. Proceedings of ISP RAS. 2016; 28(5):239–268. DOI:10.15514/ISPRAS-2016-28(5)-15. (In Russ.)
2. **Risso F., Baldi M.** NetPDL: an extensible XML-based language for packet header description. Computer Networks. 2006; 50(5):688–706. DOI:10.1016/j.comnet.2005.05.029.
3. **Morandi O., Risso F., Baldi M., Baldini A.** Enabling flexible packet filtering through dynamic code generation. 2008 IEEE International Conference on Communications. 2008; 5849–5856. DOI:10.1109/ICC.2008.1094. Available at: https://www.semanticscholar.org/paper/Enabling-Flexible-Packet-Filtering-Through-Dynamic-Morandi-Risso/54b6b04c0194e6a7a4c87f64049289ce1a3b09fd.
4. BinPAC. Available at: https://github.com/zeek/binpac (accessed 12/1/2021).
5. **Calder B.R., Masetti G.** Huddler: a multi-language compiler for automatically generated format-specific data drivers. U.S. Hydro 2015. Gaylord Hotel, National Harbor, Maryland, USA. Available at: https://www.researchgate.net/publication/277302751_HUDDLER_a_multi-language_compiler_for_automatically_generated_format-specific_data_drivers.
6. **Underwood W.** Grammar-based specification and parsing of binary file formats. The International Journal of Digital Curation. 2012; 7(1):95–106. Available at: http://www.ijdc.net/index.php/ijdc/article/viewFile/207/276.
7. **Parr T.** ANTLR (ANother Tool for Language Recognition). Available at: http://antlr.org (accessed 12/1/2021).
8. **Back G.** DataScript — A specification and scripting language for binary data. Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02). London, UK: Springer-Verlag; 2002: 66–77. Available at: http://people.cs.vt.edu/~gback/papers/gback-datascript-gpce2002.pdf.
9. **Back G.** DataScript. Available at: http://datascript.sourceforge.net (accessed 12/1/2021).
10. Kaitai Struct. Available at: http://kaitai.io (accessed 12/1/2021).
11. **Hmelnov A., Mikhailov A.** Generation of code for reading data from the declarative file format specifications written in language FlexT. 2018 Ivannikov Ispras Open Conference (ISPRAS). Moscow; 2018: 23–30. DOI:10.1109/ISPRAS.2018.00011.
12. **Bourret R.** XML data binding resources. Available at: http://www.rpbourret.com/xml/XMLDataBinding.htm (accessed 12/01/2021).
13. **Burns M.** The STL format — standard data format for Fabbers. Available at: http://www.fabbers.com/tech/STL_Format (accessed 12/01/2021).