УДК:

Интернационализация и локализация программ на языке программирования семейства Артель

Сурков Д.А., Сурков К.А., Четырько Ю.М.
ООО «Незабудка Софтвер», Минск, РБ

В статье сравниваются принятые в современных языках программирования подходы к динамическому переключению языка пользовательского интерфейса приложения. Описывается новый подход, предложенный и реализованный авторами в семействе языков программирования Артель, который основан на том, что переводимые текстовые литералы синтаксически отличаются от непереводимых, а перевод задаётся с помощью конструкций языка программирования. Показаны преимущества такого подхода: устойчивость к изменению текстовых литералов, возможность отслеживания литералов, для которых не задан перевод, полный контроль типов для параметров текстовых шаблонов, а также возможность задать перевод в виде функции. Рассмотрен пример функции перевода для сообщения с числительным.

Введение

Если взглянуть на популярные в Республике Беларусь веб-приложения и приложения для мобильных устройств, а это, например, приложения банков и сотовых операторов, то нетрудно заметить, что большинство из них предоставляют возможность выбрать язык пользовательского интерфейса. Варианты выбора включают: русский, белорусский, а иногда ещё и английский. Два первых языка — государственные, третий — популярен среди иностранцев. Этот пример иллюстрирует то, что при разработке промышленных программ с пользовательским интерфейсом часто возникает задача их перевода на разные естественные языки.

Эта задача известна как задача интернационализации и локализации программ. Интернационализация подразумевает подготовку программы к поддержке нескольких языков общения с пользователем без внесения изменений в исходный код программы, а локализация подразумевает перевод программы на язык конкретной страны или региона с учётом местных стандартов представления дат, времени, денежных единиц и прочих величин (в литературе часто используют одно из этих двух понятий, поскольку они тесно связаны).

При создании языка программирования Артель авторы пришли к выводу, что интернационализация и локализация программ должны поддерживаться на уровне языка программирования. В этой статье объясняется: почему был сделан такой вывод, и как эта поддержка реализована в языке программирования Артель.

Своё объяснение мы начнём с анализа существующих подходов.

Подход на основе встроенных ресурсов

Один из возможных подходов к интернационализации основан на том, что компилятор помещает все текстовые литералы в отдельные таблицы ресурсов, где каждый литерал идентифицируется уникальным номером, а обращения к литералу в программе заменяется на функцию получения текстового значения по этому номеру. Таблицы ресурсов встраиваются внутрь исполняемых модулей. Локализация программы при таком подходе заключается в создании версии таблицы ресурсов для целевого языка и региона.

Это подход является неудачным, причём по нескольким причинам. Во-первых, он ориентирован на создание отдельной версии программы для каждого целевого языка, а нам требуется переключать язык в работающей программе. А во-вторых, собирание без разбора всех текстовых литералов в общие таблицы ресурсов — это на самом деле плохая идея: переводчик может не разобраться со смыслом текста (ведь контекст его использования при этом отсутствует) и ошибочно перевести то, что нельзя было переводить. Например, имя файла конфигурации, имя поля или таблицы в базе данных, фрагмент веб-ссылки для загрузки скрипта или другую важную информацию, от которой зависит логика работы программы. В результате программа может оказаться сломанной. Отметим, что ошибка другого рода — отсутствие перевода там, где он требуется, — не была бы столь критичной: пользователь увидел бы на экране текст на иностранном языке, но программа осталась бы работоспособной и пользователь скорее всего смог бы продолжить работу.

Поэтому предпочтительным является подход, когда именно программист решает, требует ли текстовый литерал перевода, причём, глядя в текст программы.

Подход на основе функции перевода и текстовых словарей

Современный подход к интернационализации программ основан на том, что требующие перевода текстовые литералы оборачивают в вызов функции перевода из библиотеки интернационализации и локализации. Имя этой функции для лаконичности часто сокращают до одной буквы «t» (от англ. translate — перевод).

С помощью вспомогательных программных утилит и инструментов обёрнутые в функцию перевода тексты собираются в файлы текстовых словарей. В этих словарях ключами являются записанные в программе тексты на исходном языке, а значениями — соответствующие им тексты на целевом языке перевода. Файлы словарей обеспечивают локализацию и сопровождают исполняемые файлы программы.

Локализация выполняется не статически через создание вариантов программы для того или иного языка, а динамически, путём перевода текстовых сообщений с исходного языка на целевой язык с помощью функции перевода. Во время работы программы функция перевода выбирает словарь, соответствующий текущему языку, и выполняет поиск текста на этом языке, используя исходный текст в качестве ключа.

При таком подходе программа сопровождается текстовыми ресурсами сразу для всех целевых языков, переводимые тексты пишутся на некотором исходном языке, а перевод выполняется непосредственно перед показом пользователю сообщения. Накладные расходы на перевод текста не оказывают заметного влияния на пользовательский интерфейс, поскольку количество текстов, одновременно показываемых на экране дисплея, относительно невелико. Их количество ограничено размерами экрана и измеряется тысячами, но не миллионами.

Пример на языке программирования TypeScript

В качестве примера возьмём простейшую программу с пользовательским интерфейсом в виде текстовой консоли и покажем, как осуществляется её интернационализация и локализация.

Вот программа на популярном языке программирования TypeScript для платформы NodeJS:

```
import { createInterface } from "node:readline/promises"
import { stdin, stdout } from "node:process"

const input = createInterface({ input: stdin, output: stdout });

let name = await input.question("Как Вас зовут? ")
let age = await input.question("Сколько Вам лет? ")
console.log(`Здравствуйте, ${name}.`)
console.log(`Вам ${age} лет.`)
console.log("Спасибо за информацию!")

input.close()
```

Всё, что делает эта программа: запрашивает у пользователя его имя и возраст, а затем дублирует эту информацию в текстовой консоли. Как можно заметить, программа общается с пользователем на русском языке и других языков не поддерживает.

Именно поддержкой какого-то одного языка часто ограничивается пользовательский интерфейс программы в первой версии. Поддержку других (иностранных) языков добавляют позже, когда функциональность программы уже практически готова. Даже, когда заранее известно, что программа должна поддерживать нескольких языков общения, программисты стремятся писать все сообщения для пользователя на каком-то одном понятном им языке, а перевод на другие языки откладывают на конец разработки, делегируя переводчикам, ведь они могут не владеть всеми целевыми языками.

Воспользовавшись известной библиотекой интернационализации i18next для языка TypeScript, мы переделали наш пример в соответствии с современным подходом. Вот, что получилось:

Главный файл index.ts:

```
import { createInterface } from "node:readline/promises"
import { stdin, stdout } from "node:process"
// Импортируем наш вспомогательный модуль, подключающий словари:
import i18n from "./i18n.js"

const input = createInterface({ input: stdin, output: stdout });

// Запрашиваем у пользователя язык интерфейса и устанавливаем его:
let language = await input.question("Русский (ru) / English (en)? ")
if (language !== "en") language = "ru"
i18n.changeLanguage(language)

// Перевод текстов происходит динамически, перед показом пользователю:
let name = await input.question(i18n.t("Как Вас зовут? "))
let age = await input.question(i18n.t("Сколько Вам лет? "))
```

```
// Интерполяция текста заменена форматированием:
console.log(i18n.t("Здравствуйте, {{name}}.", { name: name }))
console.log(i18n.t("Вам {{age}} лет.", { age: age }))
console.log(i18n.t("Спасибо за информацию!"))

input.close()
```

Вспомогательный модуль i18n.ts:

```
// Импортируем модуль поддержки интернационализации и словари языков:
import i18next from "i18next"
import ru from "./locales/ru.json" with { type: "json" }
import en from "./locales/en.json" with { type: "json" }

// Настраиваем локализацию и языки:
i18next.init({
  lng: "ru",                  // язык по умолчанию
  nsSeparator: false,         // у нас знак ":" это обычный символ текста
  keySeparator: false,        // у нас знак "." это обычный символ текста
  fallbackLng: "ru",          // запасной язык
  resources: {                // список словарей переводов
    ru: { translation: ru }, // словарь переводов на русский
    en: { translation: en }  // словарь переводов на английский
  },
})
export default i18next
```

Текстовый файл locales/ru.json с переводами на русский:

```
{
  "Как Вас зовут? ": "Как Вас зовут? ",
  "Сколько Вам лет? ": "Сколько Вам лет? ",
  "Здравствуйте, {{name}}.": "Здравствуйте, {{name}}.",
  "Вам {{age}} лет.": "Вам {{age}} лет.",
  "Спасибо за информацию!": "Спасибо за информацию!"
}
```

Текстовый файл locales/en.json с переводами на английский:

```
{
  "Как Вас зовут? ": "What is your name? ",
  "Сколько Вам лет? ": "Your age? ",
  "Здравствуйте, {{name}}.": "Hello, {{name}}.",
  "Вам {{age}} лет.": "You are {{age}} years old.",
  "Спасибо за информацию!": "Thank you for the information!"
}
```

Перечислим тезисно то, на что здесь нужно обратить внимание:

- Ко многим текстовым литералам применяется функция перевода «t» из модуля «i18n», например:

```
let name = await input.question(i18n.t("Как Вас зовут? "))
```

- Функция перевода не применяется к литералам, не требующим перевода, например, к тексту сообщения с названиями языков функция «i18n.t» не применяется:

```
let language = await input.question("Русский (ru) / English (en)? ")
```

- Словари переводов — это файлы в формате JSON. Ключом в словаре является исходный текст на русском языке, например:

```
"Как Вас зовут? ": "Как Вас зовут? "
```

```
"Как Вас зовут? ": "What is your name? "
```

- Один из словарей переводов — это перевод с русского на русский. Наличие такого словаря позволяет перейти к упрощённым текстовым ключам. Например, вместо текста «Как Вас зовут? » в качестве ключа можно было бы использовать текст «имя?». В исходной программе и в файлах переводов при этом были бы следующие строки:

```
let name = await input.question(i18n.t("имя?"))
```

```
"имя?": "Как Вас зовут? "
```

```
"имя?": "What is your name? "
```

  Упрощённые текстовые ключи нужно придумывать, причём так, чтобы они не совпадали с другими упрощёнными ключами. Кроме того, они усложняют работу переводчика, поэтому на практике в качестве ключей часто используют текст на одном из целевых языков.

- Для перевода текста, который по какой-то причине отсутствует в словаре целевого языка, используется запасной язык (в примере — русский).

- Тексты с интерполяцией заменены на тексты с форматированием, а подставляемые в шаблон значения превращены в дополнительные аргументы функции перевода. Например, текст с интерполяцией

```
`Здравствуйте, ${name}!`
```

  заменён выражением

```
i18n.t("Здравствуйте, {{name}}.", { name: name })
```

  С помощью специальных символов форматирования (двойных фигурных скобок) в шаблоне текста указаны имена полей, передаваемого в функцию перевода объекта (в примере — объект с одним полем «name»). Значения полей подставляются в текст при переводе.

- Шаблон текста в выражении

```
i18n.t("Вам {{age}} лет.", { age: age })
```

  не учитывает наличие в тексте числительных. Поэтому, например, когда значение возраста равно «24», то пользователю показывается сообщение «Вам 24 лет», хотя правильно было бы писать «Вам 24 года».

Пример нашей программы написан на языке TypeScript, но демонстрируемый им подход с минимальными изменениями применяется во многих других популярных языках программирования, ведь он прост, очевиден и ориентирован на динамическое переключение языка.

Имеет ли подход недостатки? Да, вот основные из них:

- необходимость замены текстов с интерполяцией на тексты с форматированием;
- сложность перевода текстов с числительными и других подобных текстов;
- отсутствие постоянного контроля полноты переводов;
- отсутствие модульности переводов.

Первый недостаток можно считать непреодолимым в рамках используемого подхода.

Проблему перевода числительных обычно решают через создание массива текстовых ключей, индексируемых значением функции, отображающей числительное в индекс. Для английского языка, где требуется различать лишь единственное и множественное число, такая функция может быть выражена через остаток от деления числительного. Для русского языка, где требуется различать диапазоны числительного 1, 2 — 4, 5 — 20 и учитывать последнюю цифру числительного, остаток от деления не работает и функция выглядит сложнее. Функция отображения числительного в индекс не может быть универсальной и зависит от языка, что усложняет дело.

Отсутствие постоянного контроля полноты переводов означает, что при добавлении, удалении или редактировании в программе текста, нуждающегося в переводе, теряется согласованность этого текста с текстами в словарях переводов. Чтобы поддерживать согласованность, требуется периодически запускать программную утилиту сбора текстов для обновления словарей, что неудобно и затрудняет быстрое обнаружение и исправление ошибок такого рода.

Отсутствие модульности переводов означает, что словари текстов создаются для конечной прикладной программы, а не для отдельных функциональных модулей, что затрудняет интеграцию сторонних модулей с компонентами пользовательского интерфейса. Если функция стороннего модуля создаёт сообщение (например, об ошибке), которое требуется перевести на текущий язык программы перед показом пользователю, то возникает вопрос: какая функция какой библиотеки должна для этого использоваться модулем, если в языке и программной платформе нет никакого стандарта переводов. Возникают и другие вопросы. Как разработчику прикладной программы понять, какие тексты стороннего модуля требуют переводов. И как добавить эти переводы в том случае, если при создании модуля эти языки не были учтены.

По убеждению авторов, все эти недостатки могут быть полностью устранены только на уровне языка программирования, его компилятора и базовой библиотеки. Большой шаг в этом направлении был сделан в языке программирования Swift.

Подход языка программирования Swift

В языке программирования Swift задача интернационализации и локализации решается за счёт поддержки переводов в базовой библиотеке языка и в компиляторе. Сразу оговоримся, что этот подход доступен полноценно только при программировании на языке Swift в инструментальной среде XCode под управлением операционной системы MacOS.

Чтобы его продемонстрировать, мы переписали пример нашей консольной программы на языке Swift. Сначала без поддержки переводов текста, причём, с исходными текстовыми сообщениями на английском:

Главный файл программы main.swift

```
import Foundation

print("What is your name?", terminator: " ")
let name = readLine() ?? ""
print("Your age?", terminator: " ")
let age = readLine() ?? ""
print("Hello, \(name).")
print("You are \(age) years old!")
print("Thank you for the information!")
```

А потом добавили поддержку переводов на основе файла ресурсов с расширением «xcstrings» (в среде XCode этот тип ресурса имеет обозначение String Catalog). Вот, что получилось:

Главный файл программы main.swift

```
import Foundation

print("Русский (ru) / English (en)?", terminator: " ")
let language = readLine() ?? "en"
var bundle = Bundle.module
if let bundlePath = Bundle.module.path(forResource: language, ofType: "lproj") {
  bundle = Bundle(path: bundlePath) ?? Bundle.main
}

print(String(localized: "What is your name?", bundle: bundle), terminator: " ")
let name = readLine() ?? ""
print(String(localized: "Your age?", bundle: bundle), terminator: " ")
let ageText = readLine() ?? ""
let age = Int(ageText) ?? 0
print(String(localized: "Hello, \(name)!", bundle: bundle))
print(String(localized: "You are \(age) years old!", bundle: bundle))
print(String(localized: "Thank you for the information!", bundle: bundle))
```

Текстовый файл Localizable.xcstrings с переводами (формат JSON):

```
{
  "sourceLanguage" : "en",
  "strings" : {
    "Hello, %@!" : { "extractionState" : "manual",
      "localizations" : {
        "ru" : {
          "stringUnit" : { "state" : "translated",
            "value" : "Здравствуйте, %@!" } } } },
    "Thank you for the information!" : { "extractionState" : "manual",
      "localizations" : {
        "ru" : {
          "stringUnit" : { "state" : "translated",
            "value" : "Спасибо за информацию!" } } } },
    "What is your name?" : { "extractionState" : "manual",
      "localizations" : {
        "ru" : {
          "stringUnit" : { "state" : "translated",
            "value" : "Как Вас зовут?" } } } },
    "You are %lld years old!" : { "extractionState" : "manual",
      "localizations" : {
        "en" : {
          "variations" : {
            "plural" : {
              "one" : {
                "stringUnit" : { "state" : "translated",
                  "value" : "You are %lld year old!" } },
              "other" : {
                "stringUnit" : { "state" : "translated",
                  "value" : "You are %lld years old!" } } } } },
        "ru" : {
          "variations" : {
            "plural" : {
              "few" : {
                "stringUnit" : { "state" : "translated",
                  "value" : "Вам %lld года!" } },
              "many" : {
                "stringUnit" : { "state" : "translated",
                  "value" : "Вам %lld лет!" } },
              "one" : {
                "stringUnit" : { "state" : "translated",
                  "value" : "Вам %lld год!" } },
              "other" : {
                "stringUnit" : { "state" : "translated",
                  "value" : "Вам %lld лет!" } } } } } } },
    "Your age?" : { "extractionState" : "manual",
      "localizations" : {
        "ru" : {
          "stringUnit" : { "state" : "translated",
            "value" : "Сколько Вам лет?" } } } }
  },
  "version" : "1.1"
}
```

Демонстрируемые принципы этого подхода:

- Переводимые тексты упаковываются в пакеты ресурсов, для работы с которыми служит стандартный класс Bundle. Вычисляемая статическая переменная «Bundle.module» создаётся компилятором и позволяет программному пакету получить доступ к своему пакету ресурсов. Для доступа к пакету ресурсов главной программы используется вычисляемая статическая переменная «Bundle.main». В нашем примере фрагмент кода

```
        var bundle = Bundle.module
```

```
if let bundlePath = Bundle.module.path(forResource: language,
    ofType: "lproj") {
  bundle = Bundle(path: bundlePath) ?? Bundle.main
}
```

создаёт объект класса Bundle с текстовыми ресурсами для выбранного пользователем языка.

- Для перевода текстовых литералов применяется конструктор класса String с параметрами localized и bundle, например:

```
String(localized: "What is your name?", bundle: bundle)
```

Именованный параметр localized задаёт переводимый текст, а параметр `bundle` — используемый пакет ресурсов, где нужно искать перевод.

- Словари переводов — это файлы в формате JSON с расширением xcstrings. В них задаются варианты переводов сразу для всех целевых языков, например:

```
"What is your name?" : { "extractionState" : "manual",
  "localizations" : {
    "ru" : {
      "stringUnit" : { "state" : "translated",
        "value" : "Как Вас зовут?" } } }
```

Перевод для исходного языка (в примере — английского с кодом «en») может быть не задан, в этом случае используется оригинальный текст. Свойство extractionState показывает то, как текст был добавлен в словарь: вручную пользователем или автоматически компилятором. Свойство state позволяет отслеживать этапы перевода.

- Тексты с интерполяцией неявно заменяются на тексты с форматированием. Результатом перевода текста с форматированием является другой текст с форматированием, который в свою очередь используется для подстановки значений вместо форматных символов. Например, текст с интерполяцией

```
"Hello, \(name)!"
```

неявно заменяется компилятором на текст

```
"Hello, %@!"
```

в котором символы «%@» являются форматными. Знак «%» начинает последовательность форматных символов, а знак «@» задаёт тип Object для дополнительного аргумента — переменной «name». Форматные символы могут содержать номера параметров, что позволяет менять порядок подставляемых значений при переводе текста. Если бы, например, шаблон приветствия содержал в качестве параметров отдельно имя и фамилию, то изменить их порядок на фамилию и имя можно было бы через указание значения следующим образом:

```
"Hello, %2$@ %1$@!"
```

- Поддерживается перевод текстов с числительными. Например, для текста

```
"You are \(age) years old!"
```

перевод задан через указание диапазонов числового форматного параметра «%lld» (означает 64-битное целое число) для английского и русского языков следующим образом:

```
"You are %lld years old!" : { "extractionState" : "manual",
  "localizations" : {
    "en" : {
      "variations" : {
```

```
                "plural" : {
                  "one" : {
                    "stringUnit" : { "state" : "translated",
                      "value" : "You are %lld year old!" } },
                  "other" : {
                    "stringUnit" : { "state" : "translated",
                      "value" : "You are %lld years old!" } } } } },
            "ru" : {
              "variations" : {
                "plural" : {
                  "few" : {
                    "stringUnit" : { "state" : "translated",
                      "value" : "Вам %lld года!" } },
                  "many" : {
                    "stringUnit" : { "state" : "translated",
                      "value" : "Вам %lld лет!" } },
                  "one" : {
                    "stringUnit" : { "state" : "translated",
                      "value" : "Вам %lld год!" } },
                  "other" : {
                    "stringUnit" : { "state" : "translated",
                      "value" : "Вам %lld лет!" } } } } } } }
```

Возможные значения для английского языка: «one» (один) и «other» (иное), а для русского: «zero» (ноль), «one» (один), «few» (числа, заканчивающиеся на 2, 3, 4), «many» (много) и «other» (иное). При переводе значение числительного сначала преобразуется в ключ диапазона, по нему выбирается соответствующий форматный текст, затем выполняется подстановка значения числительного. В базовой библиотеке и компиляторе поддержаны все варианты диапазонов для всех основных языков мира.

Для работы со словарями переводов используется встроенный в среду программирования XCode редактор ресурсов. Программист не видит и не редактирует текстовое представление файла ресурсов Localizable.xcstrings. Однако, интерес для нас представляет именно внутреннее устройство переводов, поэтому мы показали его исходный вид в формате JSON.

Подход библиотеки SwiftUI

При создании программ с графическим пользовательским интерфейсом на основе библиотеки SwiftUI появляется возможность использовать тип LocalizedStringKey, обеспечивающий автоматический перевод текстовых литералов.

Параметр именно такого типа имеет конструктор класса Text, который используется для создания надписей в пользовательском интерфейсе. Поэтому надписи в пользовательском интерфейсе автоматически переводятся при добавлении соответствующих текстов в словарь переводов в файле Localizable.xcstrings. Пример:

```
Text("Hello, \(name)!")
```

Однако, вмешаться в перевод и обратиться произвольно к функции перевода из программы для заданного текста будет проблематично, язык Swift и библиотека SwiftUI изолируют программиста от этого.

Таким образом, подход языка Swift обеспечивает модульность переводов через поддержку на уровне базовой библиотеки языка, контроль полноты переводов, и даже поддержку текстов с интерполяцией на уровне компилятора. Перевод текстов с числительными выполняется через таблицу соответствия текстов значениям встроенной функции от числительного, специфичной для целевого языка перевода. Автоматический перевод текстовых литералов доступен только при использовании классов библиотеки SwiftUI, программист не может сам обращаться к функции перевода для преобразования объектов класса LocalizedStringKey к типу String.

Несмотря на некоторые ограничения, подход языка Swift и библиотеки SwiftUI является, пожалуй, наиболее продвинутым среди имеющихся. Впрочем, и он может быть улучшен и даже упрощён по нашему мнению путём поддержки переводов не только в базовой библиотеке и компиляторе, но и в синтаксисе языка программирования. Что мы и попытались сделать при создании семейства языков программирования Артель.

Кратко о языках программирования «Артель»

«Артель» — это название семейства новых универсальных языков программирования, созданных авторами этой статьи. В настоящий момент в семействе имеются два языка-диалекта: «Артель-А» и «Артель-М». Языки семейства объединяет то, что они поддерживают русскую лексику (наравне с английской) и позволяют использовать на русском языке англоязычные библиотеки целевой платформы за счёт встроенных средств перевода имён. Общим для языков является также наличие встроенных средств интернационализации и локализации, то есть средств перевода текстов, которые и являются предметом обсуждения этой статьи.

Далее речь пойдёт о языке Артель-А, тем не менее, рассмотренные в статье возможности доступны и в языке Артель-М.

Подход языка программирования Артель

При создании языков семейства Артель мы стремились обеспечить быстрое и удобное создание программ с пользовательским интерфейсом, поэтому заложили в основу следующие принципы:

- литералы переводимых текстов помечают специальным символом перед кавычками и/или после кавычек. В языке Артель-А — знаком «~» (тильда), что означает: текст приблизительный, заменяется близким по смыслу. Таким образом, встроенный в среду разработки компилятор может обнаруживать переводимые тексты, автоматически добавлять их в словари и проверять полноту переводов при редактировании исходника программы;
- любой программный пакет может быть дополнен пакетами переводов, содержащими словари переводов с текстами. Словари переводов определяются не в виде файлов с данными в каком-то текстовом формате, а на самом языке программирования Артель, с помощью специально предназначенного для этого синтаксиса;
- словари переводов могут содержать соответствия не только для обычных текстов, но и для шаблонов текста, то есть для текстов с интерполяцией. Более того, соответствие может быть задано в функциональном виде, что позволяет реализовать гибкий перевод любой сложности, включая правильный перевод текстов с числительными.

Покажем, как эти принципы реализованы на практике, переписав наш пример на языке Артель-А:

Пакет Интернационализация, файл Программа.арт

```
подключить Диалог

выполнить
{
  перем язык = запросить("Русский (рус) / English (eng)? ")
  если язык != "eng" { язык = "рус" }
  Переводчик.язык = язык

  пусть имя = запросить(~"Как Вас зовут? ")
  пусть возраст = Целое.из-текста(запросить(~"Сколько Вам лет? "))
  написать(~"Здравствуйте, {имя}!")
  написать(~"Вам {возраст} лет!")
  написать(~"Спасибо за информацию!")
}
```

Пакет Интернационализация, файл Конфигурация.json

```
{
  "тип": "Пакет",
  "имя": "Интернационализация",
}
```

Пакет программы называется «Интернационализация» и состоит из файла с исходным текстом программы и файла конфигурации. Добавим к нему два вложенных пакета: «Перевод.русский» и «Перевод.английский» — с переводами текстов на русский и английский.

Пакет Перевод.русский, файл Конфигурация.json

```
{
  "тип": "ПереводыТекстов",
  "имяПереводимогоПакета": "Интернационализация",
  "языкПеревода": "рус",
  "этоПервичныйПеревод": "да"
```

```
}
```

Пакет Перевод.русский, файл Тексты.арт

```
переводы
{
  ~"Как Вас зовут? " -> "Как Вас зовут? "
  ~"Сколько Вам лет? " -> "Сколько Вам лет? "
  ~"Здравствуйте, {имя}!" -> "Здравствуйте, {имя}!"
  ~"Вам {возраст: Целое} лет!" ->
  {
    пусть а = возраст % 10
    пусть б = возраст % 100
    вернуть
      условно(а >= 5 или а == 0 или б > 10 и б < 15,
        "Вам {возраст} лет.",
      условно(а == 1,
        "Вам {возраст} год.",
        "Вам {возраст} года."))
  }
  ~"Спасибо за информацию!" -> "Спасибо за информацию!"
}
```

Пакет Перевод.английский, файл Конфигурация.json

```
{
  "тип": "ПереводыТекстов",
  "имяПереводимогоПакета": "Интернационализация",
  "языкПеревода": "eng",
}
```

Пакет Перевод.английский, файл Тексты.арт

```
переводы
{
  ~"Как Вас зовут? " -> "What is your name? "
  ~"Сколько Вам лет? " -> "Your age? "
  ~"Здравствуйте, {имя}!" -> "Hello, {имя}!"
  ~"Вам {возраст} лет!" -> "You are {возраст} years old!"
  ~"Спасибо за информацию!" -> "Thank you for the information!"
}
```

Перечислим принципы этого подхода:

- Каждый пакет перевода состоит из файла конфигурации в формате JSON и файла с переводами текстов на языке Артель-А.
- Пакеты переводов имеют тип «ПереводыТекстов». В качестве переводимого пакета, то есть пакета, к которому применяются переводы, указано имя главного пакета «Интернационализация». Заданы имена языков для русского и английского: «рус» и «eng».
- Пакет «Перевод.русский» считается первичным переводом, в его конфигурации настройка «этоПервичныйПеревод» установлена в значение «да». Первичный перевод в нашем примере — это перевод с русского на русский.
- Переводы текстов заданы в файле «Тексты.арт» в блоке «переводы» как соответствия, например:

```
~"Как Вас зовут? " -> "What is your name? "
```

- Соответствия заданы не только между текстами, но и между шаблонами текстов, которые содержат параметры, например:

```
~"Здравствуйте, {имя}!" -> "Hello, {имя}!"
```

- Самое интересное. Один из переводов задан в функциональном виде, то есть представляет из себя функцию, использующую команды языка Артель для формирования результирующего текста. Эта возможность используется для показа возраста на русском языке с учётом числительного в тексте:

```
~"Вам {возраст: Целое} лет!" ->
{
  пусть а = возраст % 10
  пусть б = возраст % 100
  вернуть
    условно(а >= 5 или а == 0 или б > 10 и б < 15,
      "Вам {возраст} лет.",
    условно(а == 1,
      "Вам {возраст} год.",
      "Вам {возраст} года."))
}
```

Этот перевод задан функцией, у которой параметром является параметр шаблона «возраст» с типом «Целое». Функция берёт остатки от деления параметра «возраст» на 10 и на 100, а потом использует эти значения для формирования и возврата правильного текста сообщения. Поэтому, например, для аргумента 24 будет возвращён текст «Вам 24 года», а для аргумента 21 — «Вам 21 год». Если тип параметра в шаблоне не указан, он принимается равным типу «Объект?», который совместим со всеми другими типами (знак вопроса после имени типа в языках Артель разрешает значение «пусто» в качестве аргумента функции).

Пакеты переводов имеют не показанную в примере потенциальную возможность подключать обычные программные пакеты и использовать их функции и типы.

Есть также ограничение: перевод не может быть задан через другой перевод в функциональном виде, то есть переводимый текстовый литерал (со знаком «~») не может быть указан после знака «->». Это искусственное ограничение сделано для исключения возможных бесконечных рекурсий при выполнении переводов.

Отметим важное следствие подхода — отсутствие необходимости встраивать в базовую библиотеку поддержку всех вариантов числительных для всех языков мира и проверять соответствие указанных пользователем вариантов целевому языку.

Управление переводом

Всегда найдётся случай, который не вписываются в рассмотренный нами сценарий. Например, программисту может потребоваться перевести текст на язык, отличный от языка пользовательского интерфейса, или сделать перевод на несколько языков. Поэтому программисту доступен стандартный класс «ТекстПереводимый», позволяющий вмешаться в процесс перевода и выполнить перевод самостоятельно.

Присваивание литерала переводимого текста или шаблона текста переменной с типом «ТекстПереводимый» приводит не к переводу литерала, а к созданию из него объекта класса «ТекстПереводимый» без выполнения перевода. Например, команда:

```
пусть шаблон: ТекстПереводимый = ~"Здравствуйте, {имя}!"
```

превращается компилятором в другую команду, которая на самом языке Артель-А записывается следующим образом:

```
пусть шаблон: ТекстПереводимый = ТекстПереводимый(
  фрагменты = ["Здравствуйте, ", "}!"],
  значения = [имя],
  ключ = пусто,
  пакет = Пакет.свой-пакет)
```

Таким образом из литерала текста создаётся объект, содержащий:

- фрагменты шаблона,
- подставляемые в шаблон значения параметров,
- ключ текста,
- пакет, из которого происходит литерал.

Свойства «фрагменты» и «значения» наследуются классом «ТекстПереводимый» у базового класса «ТекстШаблонный», который в свою очередь используется для работы с обычными (не переводимыми) шаблонами текстов. Свойство «ключ» служит для поиска перевода в словаре. Значение «пусто» для него указывает на то, что ключ должен быть сформирован по умолчанию путём соединения фрагментов шаблона через знак «_» (подчёркивание). Свойство «пакет» служит для идентификации пакета, в котором определён литерал текста. Выражение «Пакет.свой-пакет», указанное в качестве его значения, ссылается на компилируемый пакет (у каждого пакета значение своё).

Чтобы выполнить перевод текста, нужно вызвать у объекта «ТекстПереводимый» метод «перевести» или просто преобразовать объект в текст, присвоив его значение переменной с типом «Текст», например:

```
пусть переведённый-текст = шаблон.перевести()
пусть переведённый-текст-2: Текст = шаблон
```

Перевод выполняется на язык, заданный в статическом свойстве «Переводчик.язык». Если нужно выполнить перевод на какой-то другой язык, используют метод «перевести-на», например:

```
пусть текст-на-русском = шаблон.перевести-на("рус")
пусть текст-на-английском = шаблон.перевести-на("eng")
```

Здесь стоит отметить, что объект класса «ТекстПереводимый» скрытно создаётся из литерала переводимого текста или шаблона всегда, то есть даже тогда, когда целевая переменная принадлежит классу «Текст». Например, команда

```
пусть текст = ~"Здравствуйте, {имя}!"
```

превращается компилятором в команду создания объекта с вызовом метода «перевести»:

```
пусть текст = ТекстПереводимый(["Здравствуйте, ", "}!"], [имя], пусто,
  Пакет.свой-пакет).перевести()
```

Метод «перевести» обращается к одноимённому статическому методу класса «Переводчик». Класс «Переводчик» отнесён к стандартным классам, он управляет словарями переводов, хранит текущий язык пользовательского интерфейса и обеспечивает перевод на этот язык.

Ключи текста

Бывают сценарии, когда для двух идентичных литералов текста нужно определить разные переводы. Обычно это обусловлено контекстом, в котором используется литерал. Например, в зависимости от контекста для текста «Привет!» может потребоваться определить два перевода на английский: «Hello!» и «Hi!».

Если одинаковые текстовые литералы принадлежат разным пакетам (модулям), то проблемы нет, поскольку словари пакетов между собой не пересекаются, а выбор словаря выполняется на основе ссылки на пакет, к которому привязан литерал переводимого текста. А вот для случая когда текстовые литералы находятся в одном и том же пакете, добавлена возможность явно указывать их ключи.

Ключ указывается в конце текстового литерала после знака «~» (тильда), например:

```
пусть текст-1 = ~"Привет!"~"П1"
пусть текст-2 = ~"Привет!"~"П2"
```

Ключи текста «П1» и «П2» позволяют отличать литералы и определить для них разные переводы.

Контроль полноты переводов в компиляторе

Компилятор Артель для инструментальных сред программирования Visual Studio Code и Eclipse Theia обеспечивает не только подсветку синтаксиса и навигацию по коду, но и контроль полноты переводов. Например, если в файле первичных переводов на русский удалить какой-нибудь перевод, то для соответствующего текстового литерала в программе будет показано предупреждение: «Текст отсутствует среди определений первичных переводов текстов».

Заключение

Подведём итог. Интернационализация и локализация программ с пользовательским интерфейсом может быть значительно упрощена за счёт поддержки в языке программирования. Языки семейства Артель подтверждают этот тезис и предлагают решение, ориентированное на модульность, контроль

полноты переводов и учёт контекста. Уникальная возможность этого решения — функции перевода, позволяющие реализовать перевод любой сложности, в частности, перевод текстов с числительными.

Литература
1. https://www.i18next.com/
2. https://developer.apple.com/documentation/xcode/localization
3. https://developer.apple.com/documentation/swift/string
4. https://artel.by/ru/a/overview

Дополнительные источники
1. https://artel.by/ru/a/tutorial

Сурков Дмитрий Андреевич, зам. директора по технологиям, ООО «Незабудка Софтвер»
(dmitry.surkov@nezaboodka.by)
Сурков Кирилл Андреевич, зам. директора по продуктам, ООО «Незабудка Софтвер»
(kirill.surkov@nezaboodka.by)
Четырько Юрий Михайлович, директор, ООО «Незабудка Софтвер»
(yury@nezaboodka.by)
ORCID: 0000-0000-0000-0000

Ключевые слова
язык программирования, интернационализация, локализация, перевод текста, функция перевода текста, модульность, полнота переводов, текстовый литерал, шаблон текста, текст с подстановкой, интерполяция текста, текстовый ресурс.

Dmitry Surkov, Kirill Surkov, Yury Chetyrko. An approach to program localization in the Artel family of programming languages.

Keywords
programming language, internationalization, localization, text translation, text translation function, modularity, completeness of translations, text literal, text template, text substitution, text interpolation, text resource.

DOI:
JEL classification — C65

Abstract
The article compares the approaches adopted in modern programming languages to dynamically switch the language of the user interface of an application. A new approach proposed and implemented by the authors in the Artel family of programming languages is described, which is based on the fact that translatable text literals are syntactically different from untranslatable ones, and translation is set using programming language constructs. The advantages of this approach are shown: resistance to changes in text literals, the ability to track texts for which no translation has been made, full type control for text template parameters, and the ability to set the translation as a function. An example of a translation function for a message with a numeral is considered.