USING OF "INTERNATIONAL COMPONENTS FOR UNICODE" FOR APPLIED TASKS OF COMPUTATIONAL LINGUISTICS (CASE OF STRUCTURAL TEXT ANALYSIS)

Yuriy Koval, Maxim Nadutenko

(in Russian)

***Abstract**: The present article deals with the peculiarities of character encoding in information-communication systems technologies. Article provides historical information about different systems, approaches and standards of character encoding starting from ASCII to Unicode. Reasons and conditions of the development of each standard have been provided accordingly. Unidoce has been defined as common system for character encoding, counter-compatibility with ASCII and characters encoding method of Unicode have been illustrated. International Components for Unicode (ICU) has been discovered as common instrument for structural text analysis. List of functionality of ICU has been provided as well as spheres of using of ICU.*

The present article provides short description of Swift programming language and interfaces of ICU which are used in this programming language for dealing with regular expressions. Logic for interlanguage text transformation in the Swift described. Detailed process of integration of ICU for using with Swift programming language has been provided.

The present article contains information about applications of text boundary analysis, types of boundaries and principles of text processing by BreakIterator ICU classes, which all listed in the Standard Annex #29 (Unicode text segmentation). Unicode Common Locale Data Repository has been defined as a main source of locale information for specific regional rules, dedicated for specifying the rules for text boundary analysis.

Icu4c-swift framework has been used for manipulation of ICU logic with Swift programming language. Such text boundaries as words and sentences have been marked with specific markers. Correlation between time of marker insertion, text size in memory, quantity of symbols in text has been provided in the corresponding tables and graphs. Special attention has been paid to the usage of Swift.String.index(_:offsetBy:) function. Correct usage of such function can increase the speed of program execution dramatically.

Advantages and disadvantages of using the International Components for Unicode with Swift programming language have been analyzed. The emphasis was made on using specific functions of Swift with International Components for Unicode for text analysis.

***Keywords**: character encoding, Unicode, structural text analysis, computational linguistics.*

Использование “International Components for Unicode” в прикладных задачах компьютерной лингвистики (на примере структурного анализа текста)

Юрий Коваль, Максим Надутенко

***Abstract**: В данной статье рассматриваются особенности кодирования символьных данных в системах информационно-коммуникационных технологий. В качестве стандарта для кодирования символов был выбран Unicode, а International Components for Unicode как основной инструмент для структурного анализа текста. Были проанализированы и показаны преимущества и недостатки использования International Components for Unicode совместно с языком программирования Swift. Предложены рекомендации по применению функционала Swift для работы с текстом с помощью International Components for Unicode.*

***Ключевые слова:** кодирование символов, Unicode, структурный анализ текста, компьютерная лингвистика*

***ITHEA Keywords:** D.2.3 Coding Tools and Techniques, I.2.7 Natural Language Processing*

Введение

Текст состоит не из слов, а из словосочетаний. При взаимодействии человека с текстом, границы в нем помогают адекватно воспринимать последовательность символов и интерпретировать эту последовательность в собственном когнитивном аппарате как информацию. Именно эти границы помогают «увидеть» слова и сложить их в

словосочетания, присвоив последовательности символов семантику. Поскольку задачи обработки текста сейчас все больше передаются от человека компьютеру, возникает проблема в адекватном обучении компьютера определять границы в тексте. Для человека, который работает с текстом на компьютере, в свою очередь, вызовом является правильный выбор системы для обработки символов. От правильного выбора такой системы зависит достижение целей, которые ставит перед собой человек, обрабатывая текст, ведь результат работы различных систем обработки символов может отличаться.

Так, практически любой специалист по компьютерной лингвистике в процессе профессиональной деятельности сталкивается с проблемой кодирования текстовой информации и токенизации текста, то есть, разбиения текста на слова, предложения и тому подобное. Например, символ Emoji, с изображением семьи из мужчины, женщины, мальчика и девочки в одной системе, будет представлен как отдельные символы человека мужского пола, человека женского пола, ребенка мужского пола и ребенка женского пола в другой системе из-за различий в кодировке символов. Другой пример можно взять из структурной разметки текстов, где чтобы выделить прямую речь, диалоги, слова автора, части предложения и другие необходимые для дальнейшей обработки текста единицы, необходимо сначала выполнить разбиение текста на предложения, слова и тому подобное. Хотя вышеупомянутые задачи и имеют много имеющихся на сегодняшний день путей решения, но все равно являются нетривиальными, а конкретные решения зависят от использования конкретных систем обработки естественного языка, языков программирования и даже операционных систем.

Целью статьи является исследование использования преимуществ International Components for Unicode (ICU) на практике, а именно применение ICU совместно с языком программирования Swift и выявления положительных и отрицательных сторон использования инструментов ICU для задач структурной разметки текста, предоставление практических рекомендаций для разбиения текста на слова и предложения. ICU является библиотекой программных средств консорциума Unicode предназначенной для обработки текста, символьных данных, решения проблем интернационализации, временных поясов, календарей, дат и др. [2]

Поскольку каждая платформа имеет свои требования и стандарты функционирования, имеет смысл использовать кроссплатформенное решение для работы с текстом, которое будет вести себя одинаково независимо от стандартов конкретной платформы, или разработать один стандарт интерфейса, который будет иметь нативную реализацию подобно BLAS (Basic Linear Algebra Subprograms). Здесь и выходит на передний план ICU, которая выступает единственной системой для адекватной обработки символьных данных

и которая по умолчанию используется различными платформами, операционными системами и языками программирования.

Кроме "источника" стандартизированных подходов к кодировке символьных данных, группа разработчиков ICU предоставляет и постоянно совершенствует алгоритмы для работы с текстом, в частности алгоритмы токенизации. Конечно, для решения специфических задач может быть целесообразной реализация собственного программного решения, но тогда возникнет необходимость реализации функционала, который уже содержится в ICU и прошел испытание временем.

Другим аргументом в пользу использования ICU явилась способность библиотеки обрабатывать тексты, написанные на разных языках. Разработка необходимого функционала для работы, скажем, с кириллицей не является слишком сложной задачей, но если возникает необходимость в обработке текстов на латинице, китайских иероглифов, то сложность системы значительно возрастает, и требуется поиск решения множества различных подзадач с большим количеством исключений из правил.

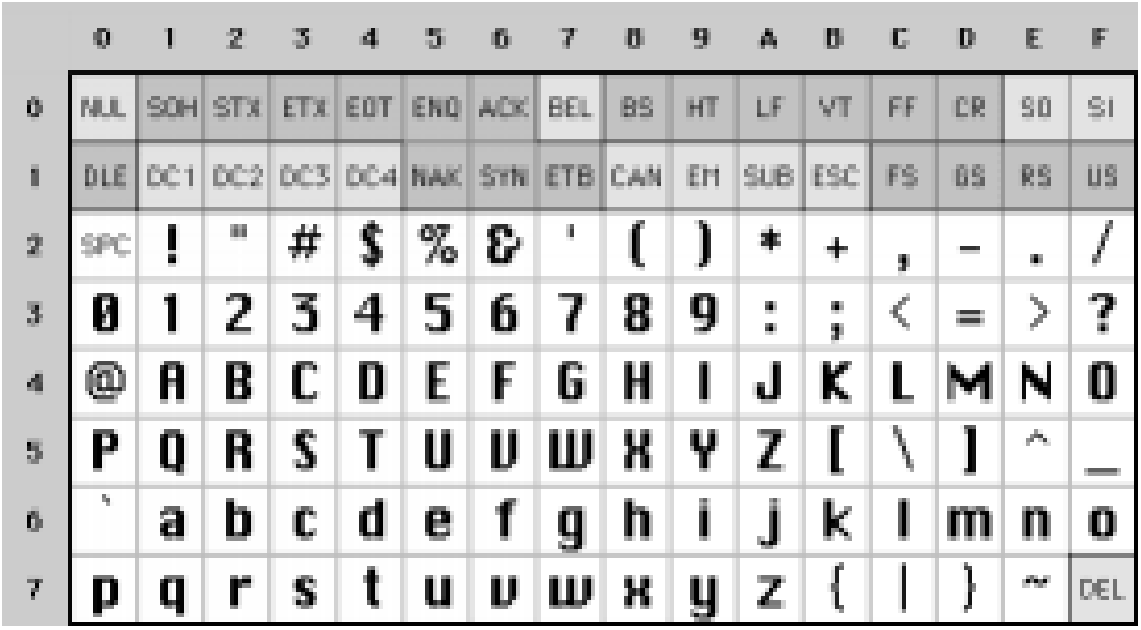
International Components for Unicode написана на низкоуровневом языке программирования C, что позволяет получить более высокую скорость работы библиотеки по сравнению с другими библиотеками, для которых не проводилась фундаментальная многолетняя работа по оптимизации использования памяти и улучшению быстродействия.

Краткие сведения о International Components For Unicode

Исторические сведения о International Components for Unicode

Как указано на сайте разработчика: «ICU - это фундаментальный, широко используемый набор библиотек C/C++ и Java, обеспечивающих поддержку Unicode и Globalization для программных приложений. ICU портативен и обеспечивает одинаковые результаты для приложений, выполняющихся на различных платформах и написанных на языках C/C++ или Java» [2]. Разработке данной библиотеки предшествовал ряд исторических событий, связанных с представлением символьных данных в компьютерных системах.

Первым стандартом для размещения набора символов была таблица ASCII (American Standard Code for Information Interchange, Американский стандартный код для информационного обмена), которая содержала 33 сервисных символа с кодами от 0 до 31, а также 95 печатных символов ASCII с кодами от 32 до 126 (рис.1).



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SPC	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Рис. 1. ASCII - Американский стандартный код для информационного обмена

Многие решения на стадии разработки функционала для работы с символьными данными принималось исходя из требований, которые диктовались аппаратным обеспечением той или иной платформы [1]. Например, символу Delete в таблице ASCII соответствует код x'7F ", из-за необходимости выбить все отверстия в колонке перфокарты, чтобы сигнализировать, что колонка должна быть проигнорирована.

ASCII можно было вместить в 7 бит, а большинство компьютеров того времени использовали 8-битную систему. Все происходило хорошо до тех пор, пока не появлялось необходимости в использовании символов не английского алфавита. Так как компьютеры использовали 8 бит, а вся таблица занимала 7 - во многих одновременно возникла идея задействовать целый бит в собственных целях (а это коды от 128 до 255). Это, конечно, создало определенный хаос в передаче информации от одного компьютера другому, так как коды после 128 в одной системе отличались от кодов на тех же позициях в другой. Тогда IBM-PC ввела так называемые «кодовые страницы» (code pages), которые впоследствии были переименованы в наборы символов (character set) OEM (Original Equipment Manufacturer), которые покрыли символы большинства европейских языков, использующих латинский алфавит. Кроме символов алфавита, наборы символов IBM-PC OEM содержали в себе символы для рисования (линии, двойные линии, наклонные линии и т.д.) рис. 2.

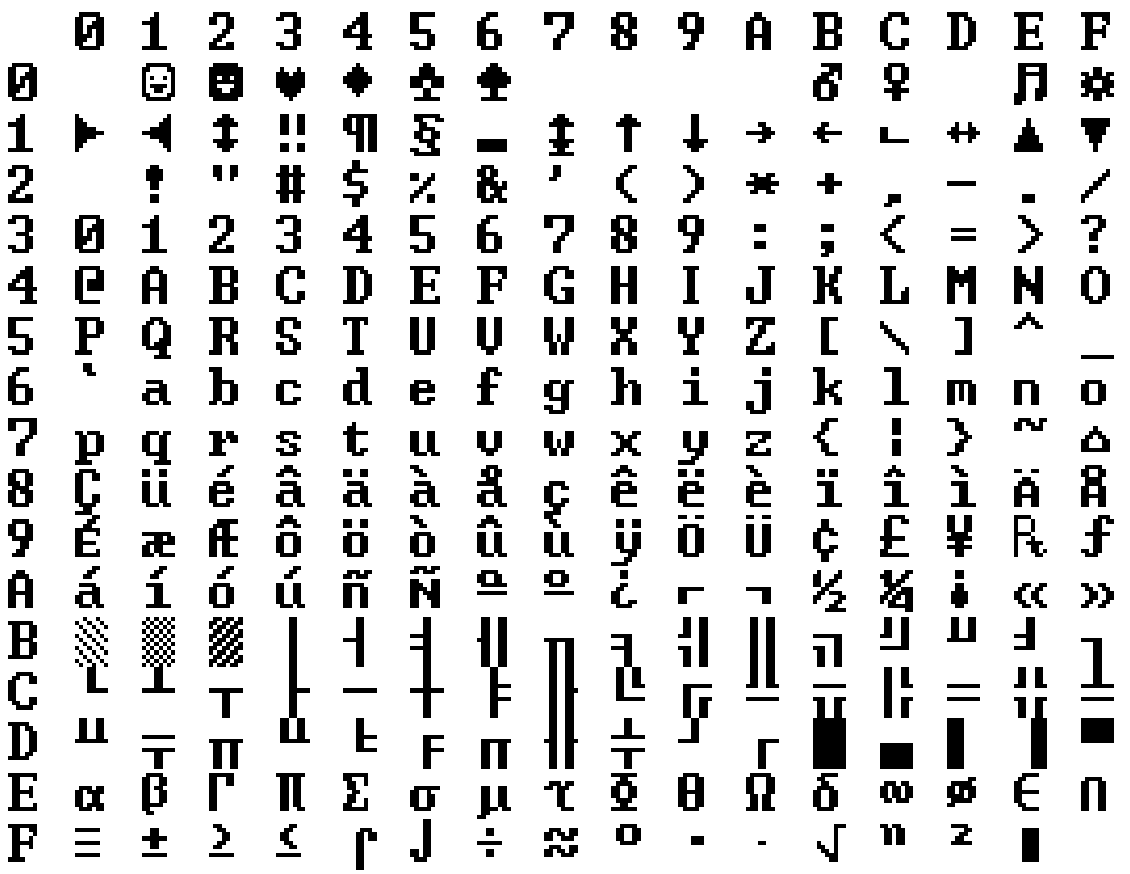


Рис. 2. Набор символов IBM-PC OEM

С момента, когда персональные компьютеры стали покупать за пределами США, все необходимые для обмена информацией символы локальных алфавитов стало возможно разместить в позиции выше 128 (первые 128 формально было договорено выделять для ASCII). Это и стало моментом создания стандарта ANSI, с различными системами кодов, которые уже здесь получили название "кодовые страницы". Например, в Израиле использовалась кодовая страница 862, в тот момент как в Греции – 737, то есть каждая национальная операционная система MS-DOS содержала в себе специфическую для региона страницу кодов. Существовали даже страницы "межнациональной" языка – эсперанто. Но для того, чтобы объединить в одном компьютере, скажем, украинский и немецкий языки, нужно было писать собственную программу для отображения различных символов.

Что касается систем письменности азиатский языков, которые имеют тысячи символов, поместить их в 8 бит явно не представлялось возможным. Поэтому использовалась система под названием DBCS (double byte character set) - набор двухбайтных символов, где иногда один символ занимал один байт, иногда - два. Чтобы исключить перевод каретки на байт, лежащий в пределах одного символа, не рекомендовалось проходить по символам

обычным итератором (как вперед, так и назад). Взамен предлагалось использовать функции вроде Windows.AnsiNext и Windows.AnsiPrev. В основном символы обрабатывались исходя из правила "один символ занимает один байт", которое работало хорошо до того момента, когда приходилось передать текст с одного компьютера на другой, что стало насущной проблемой с началом использования Интернета.

Для того, чтобы сохранить единый функционал в старых и новых системах, а также из-за ограниченной вместимости таблицы символов ранних вычислительных систем, группа разработчиков системы Unicode наложила много ограничений на таблицы кодировок и на сам функционал обработки символьных данных.

Понимая необходимость плавного и наименее болезненного перехода на новую систему кодирования, разработчики Unicode сохранили соответствие с фактически действующим на то время стандартом ASCII. На пример, в ASCII символ “H” имел позицию 48. В Unicode он обозначается такой последовательностью как “U+0048”, что называется единицей кодирования (code point). Если посмотреть на кодирование слова “Hello”, то можно увидеть четкую аналогию между единицами кодирования и позициями в ASCII:

U+0048 U+0065 U+006C U+006C U+006F

Что касается символов, которые не входят в английский алфавит, то эту проблему разработчики Unicode разумно решили, выделив несколько единиц кодирования для интерпретации одного символа. То есть ограничение в 8 бит было снято и появилась возможность представить любой символ, выделив на него (при необходимости) несколько байт. Так, большинство символов Emoji и китайской системы письменности могут занимать до четырех байт.

После создания стандарта для хранения символов и унификации их кодирования, стала возможной адекватная передача и однозначная интерпретация текстовой информации, написанной на различных языках. После этого возникла потребность в автоматизации обработки последовательностей символов в тексте, которые образуют слова, предложения, даты и прочее. Более того, возникла необходимость делать это с учетом особенностей обработки такого рода информации в различных культурах и отдельных странах. Решения для этих проблем вышли в свет в 1999 году с выпуском International Components for Unicode (конечно, не окончательные, но необходимые для функционирования систем глобализации и интернационализации в различных языках программирования и информационных продуктах).

Чтобы лучше понять возникновение системы Unicode и проекта International Components for Unicode, а также предпосылки создания наборов стандартных символов, которые

использовались и продолжают использоваться в компьютерных системах и алгоритмов обработки текстов, стоит рассмотреть этот процесс в хронологическом порядке:

- 1986-1987 разработка шрифтов для китайского языка в Xerox, Apple присоединяется к ANSI X3L2 и работает над универсальным набором символов;
- Февраль 1988 – в Apple начинают работать над 16-битной системой кодирования "High Text";
- Апрель 1988 – в Apple разработаны первые прототипы кодировки текста в Unicode;
- Сентябрь 1988 – Apple покупает базу данных азиатских символов (CJK – Chinese, Japanese, Korean) для работы с базой данных Han;
- Февраль 1989 – начало регулярных встреч между Sun, Adobe, HP, NeXT, Pacific Rim Connections для работы над Unicode;
- Август 1989 – Xerox и Apple объединяют базы данных Han для совместной работы над азиатскими символами;
- Октябрь 1989 – Unicode представляют для Microsoft и IBM на фоне сотрудничества Apple и Microsoft над TrueType;
- Май 1990 – презентация Unicode на глобальной конференции разработчиков Apple;
- Ноябрь 1990 – презентация Unicode на конференции IEEE;
- Январь 1991 – создание Unicode Consortium;
- Декабрь 1991 – создание базы данных UniHan;
- Июнь 1992 – публикация второго выпуска Unicode Standard Version 1.0;
- Начало 1999 – выход в свет "IBM Classes for Unicode", позже переименованного в International Components for Unicode [18].

Более подробную информацию можно найти в Chronology of Unicode Version 1.0 [17].

Итак, мы видим, что историю кодирования символов можно свести к утверждению трех основных стандартов: ASCII, ANSI и Unicode, последний из которых стал тем, который вобрал в себя все символы алфавитов и систем письменности на Земле.

Усилия, вложенные в создание Unicode и International Components for Unicode, явились основой для начала нового этапа в обмене информацией в "Сети сетей", унифицировав в одном месте набор символов, содержащий каждую из общепринятых систем письма (знаков) на планете. Таким образом, создание Unicode в начале 90-х годов XX века (а именно в июне 1992 [17]), сыграло ключевую роль не только в удобстве обмена информацией в Интернете, но и в переходе человечества ко второй волне сетевой эпохи развития.

Сферы применения International Components for Unicode

International Components for Unicode применяется в самых разных областях для обработки текстовой информации. Так, на сайте проекта ICU приводятся следующие основные (не полный перечень) функции, которые эта система предоставляет [1]:

- сравнение символов (в соответствии с выбранными правилами);
- представление множества символов юникода;
- сравнение порядка кодовых единиц (code points, code units);
- итерация по символам юникода;
- локаль (региональная специфика);
- сервисы даты и времени (календарь, временные зоны и т.д.);
- преобразование внутреннего представления даты в текстовое представление времени, атрибуты календаря и т.д.;
- работа с регулярными выражениями;
- "двусторонняя" обработка текста (обработка текста как с последовательностью символов слева направо, так и справа налево);
- анализ границ в тексте (определение позиций слов, предложений, абзацев в тексте и т.д.).

В данной статье больше внимания будет уделено токенизации и анализу границ в тексте, но стоит помнить, что основное, для чего следует использовать ICU, так это для идентификации символов, потому что она является наиболее полным "складом" кодовых единиц юникода, включая Emoji и CJKV (Chinese, Japanese, Korean, Vietnamese).

Очевидно, что многие языки программирования используют в своей основе логику ICU для работы с некоторыми задачами из приведенного выше списка, но в основном этот функционал ограничен и рекомендуется, при необходимости решить более узкоспецифическую проблему, использовать именно ICU [2], а не логику, взятую из конкретного языка программирования.

Итак, можно сделать вывод, что хотя и приведен далеко не полный список возможностей, которые предоставляет ICU, но из него понятно, что эта логика используется в большинстве систем, работающих с датами и календарями, локалью, кодированием символов систем письменности различных языков (включая системы с различным направлением чтения), что и обеспечивает набор средств для решения проблемы интернационализации и глобализации информационных технологий.

Интеграция ICU с системами обработки естественного языка

Интеграция со Swift programming language

Swift programming language – один из часто используемых высокоуровневых языков программирования на сегодняшний день, занимает 15 место в рейтинге TIOBE [5]. Представленный на World Wide Developers Conference в 2014 году и разработанный группой, возглавляемой Крисом Латнером, как язык программирования для операционных систем macOS, iOS, watchOS и tvOS [6], сейчас широко используется не только для выполнения своих первоочередных задач, но и для написания программ под операционную систему Linux, реализации back-end логики и т.д. [8]. Именно после перевода Swift programming language в открытый доступ, Swift начал использоваться для более широкого спектра задач.

Swift, как и многие другие языки, использует ICU для работы с текстом. То есть часть функционала ICU поставляется внутри стандартных библиотек, а конкретнее в Foundation framework. Например, Apple Inc. [9] приводит такие интерфейсы ICU, используемые в Foundation Framework для работы с регулярными выражениями (regular expressions):

parseerr.h	*uregex.h*
platform.h	*urename.h*
putil.h	*ustring.h*
uconfig.h	*utf_old.h*
udraft.h	*utf.h*
uintrnal.h	*utf16.h*
uiter.h	*utf8.h*
umachine.h	

Олег Бегемана, разработчик и член комиссии по развитию Swift programming language, на своем веб-сайте [10] приводит пример, как ICU используется в Foundation Framework для текстовых трансформаций. В частности, он говорит о транслитерации, которая наглядно иллюстрирует, что ICU является прекрасным инструментом, когда нужно работать с символьными данными алфавитов разных языков:

```
import Foundation
let shanghai = "上海"
shanghai.applyingTransform(.toLatin, reverse: false)
// → "shàng hǎi"//результат после трансформации
```

Swift содержит библиотеку для анализа естественного языка с помощью методов машинного обучения, которая также решает задачи токенизации (Tokenization) [7]. Однако, Swift не содержит интерфейсов ICU, необходимых для выполнения задач структурной разметки текста, а именно его разбиения на предложения и слова. Из-за этого появляется необходимость использования логики ICU, к которой приходится обращаться конкретным объектам, написанным на Swift. Как готовое решение проблемы была использована библиотека icu4c-swift [11], в которой реализован доступ к ICU через такие объекты-обертки как CharacterBreakCursor, LineBreakCursor, RuleBasedBreakCursor, SentenceBreakCursor, WordBreakCursor и другие.

Если же есть необходимость в использовании ICU через Swift напрямую, то это можно сделать, импортировав необходимые интерфейсы в собственноручно созданный модуль, а саму библиотеку ICU установить в операционную систему как отдельный компонент, отличный от того, который уже используется системой (если есть необходимость использовать определенную версию ICU и системная версия не содержит нужных интерфейсов). То есть выглядит это следующим образом:

1. Создание "упаковки" модуля для установки ICU. Файл "упаковки" выглядит следующим образом:

```
Let package = Package (name: "icu4c-swift", pkgConfig: "icu-uc", providers: [ .Brew("icu4c"), .Apt("libicu-dev"), ])
```

2. Создание непосредственно файла модуля:

```
module ICU4C {
header "umbrella.h"
link "icucore"
export *
}
```

3. Создание файла для импорта необходимых интерфейсов с ICU, так называемого "зонта" для интерфейсов:

```
//umbrella.h
#import <unicode/icudataver.h>
#import <unicode/parseerr.h>
#import <unicode/platform.h>
```

После выполнения этих шагов можно использовать ICU-логику в собственных объектах. Важно помнить, что ICU включает в себя множество функционала, который является стандартом и используется в различных языках программирования. Не является исключением и Swift. Те функции, которые использует сам язык программирования, называются приватными и их использование "напрямую" может рассматриваться с стороны Apple как несанкционированное проникновение в нативное API, поэтому, соответственно, Apple может отказать в публикации приложения в AppStore на этапе его рассмотрения. К такому API, кроме прочего, относятся такие компоненты как ubrk_current, ubrk_first, ubrk_next, используемые в ICU для разбиения текста. Выходом из такой ситуации может быть использование не системного ICU, а статически скомпилированной библиотеки.

Итак, мы видим, что Swift programming language содержит большое количество логики ICU для работы с символьными данными и текстом в библиотеке Foundation, но если нужно разбивать текст на предложения, слова и т.д., то можно использовать недостающую в Swift ICU-логику, которую можно задействовать как через прямой доступ к системной ICU, содержащейся в операционной системе, так и собрав отдельный модуль, что безопаснее, благодаря гарантированного не использования нативного API, а также работе с конкретной версией библиотеки.

Использование International Components For Unicode для структурной разметки текста

Разбиение текста на слова

Анализ границ текста с помощью ICU (определение лингвистических границ при форматировании и обработке текста) включает в себя [4]:

1. определение позиции соответствующих точек для обрамления текста в слово для помещения между специфическими отступами при отображении или печати;
2. определение начала слова, выбранного пользователем;

3. подсчет знаков, слов, предложений, абзацев;
4. определение, как далеко переставить курсор при нажатии на клавишу стрелки (некоторые символы занимают более одной позиции в тексте, а некоторые символы не отображаются вообще);
5. создание списка уникальных слов в тексте;
6. капитализацию первой буквы слов;
7. определение отдельной единицы в тексте, слове (десятое слово второго абзаца и т.д.).

Классы BreakIterator в ICU были созданы именно для подобных задач. Существует четыре типа границ в тексте, в отношении которых применяются итераторы ICU:

1. границы знаков;
2. границы слов;
3. границы разрыва линий;
4. границы предложений.

Каждый тип границ определен в соответствии с правилами, описанными в Приложении номер 29 к Стандарту Unicode [12], а также с алгоритмом разбиения строк того же Unicode [13].

Итератор символьных границ определяет границы согласно правилам границ графемного кластера из вышеупомянутого Приложения номер 29. Границы определены таким образом, чтобы соответствовать классификации "что такое символ" самого пользователя, то есть базовой единицы системы письменности для конкретного языка, символ которой часто может занимать более одной единицы кодирования (code point) в системе Unicode. В качестве примера на странице инструкции по использованию итераторов разбиения ICU [4] приводится буква Ä, которая может занимать как одну единицу кодирования, так и две, когда в первой сохраняется буква A, а во второй хранится ее умлаут (диакритический знак в немецком и других языках, имеет вид двух точек над буквой). В обоих случаях это будет расценено как один символ.

Итератор словесных границ определяет позиции границ слов. Примерами применения итератора словесных границ являются: выделение слова при двойном щелчке клавишей мыши на нем, поиск полных слов, подсчет количества слов в тексте.

Границы слов также определены в Приложении 29 к стандарту Unicode, но в случае для слов они дополнены словарями для китайского, японского и других языков. Важной и, наверное, основной особенностью итератора словесных границ является то, что им принимаются во внимание алфавиты и традиции правописания различных языков.

Для разработки программы для структурной разметки текста мы использовали библиотеку ICU4C-swift, где класс WordBreakCursor является оберткой над итератором словесных границ ICU.

Если же говорить о быстродействии, то очень наглядно демонстрируется быстродействие работы самой ICU по сравнению с высокоуровневыми структурами в Swift. Прохождение курсора по границам слов в тексте занимает не так много времени, как вставка символов и поиск позиции в тексте с помощью функции String.index (_: offsetBy :). Сложность функции указана в документации [14] как O(n), но если использование данной функции еще возможно для небольших объемов текста, то с увеличением количества слов в тексте, время выполнения программы возрастает до критического уровня (Рис. 3):

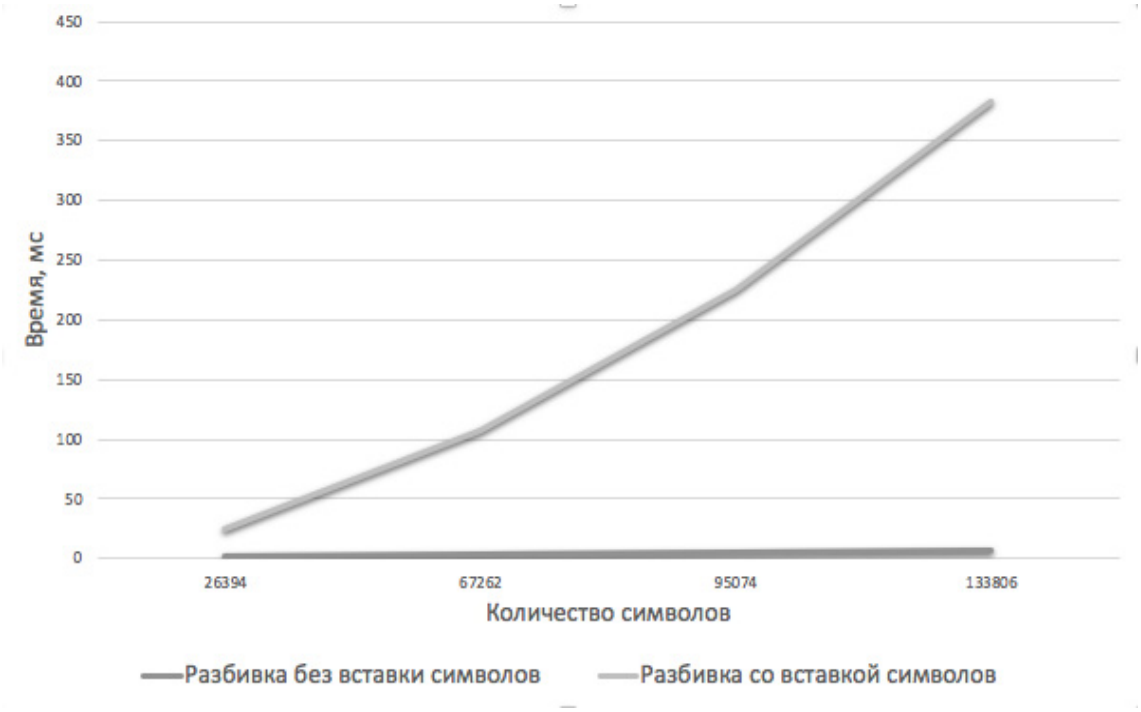


Рис. 3. Зависимость времени разбиения на слова от количества символов в тексте

Для анализа использовался текст Конституции Украины, который был условно поделен на четыре части. При этом время разбиения без вставки символов выросло с 5,9 миллисекунд до 28,6 миллисекунд при увеличении количества символов с 26394 до 133806. То есть с увеличением количества символов в тексте в 5,07 раза, время разбиения выросло в 4,85 раза, что не является удовлетворительным результатом, но приемлемо для дальнейшего совершенствования написанной программы.

Что касается разбиения текста на слова со вставкой символов структурной разметки, то время выросло с 5509,3 миллисекунд до 130395,1 миллисекунд при соответствующем росте количества символов. То есть с увеличением количества символов в тексте в те же 5,07 раза, время выросло в 23,7 раза, что говорит о том, что использование данного алгоритма никак невозможно без его улучшения и изменения.

Таблица 1. Зависимость времени разбиения текста на слова от количества символов в тексте

Количество знаков	Размер текста, байт	Время разбиения без вставки символов, мс	Время разбиения со вставкой символов, мс
26394	53590	5,9	5509,3
67262	136534	14,5	33662,6
95074	193082	19,9	66060,6
133806	271536	28,6	130395,1

Из таблицы 1 видно, что наибольший прирост времени наблюдается в момент вставки символов. При рассмотрении этой операции было проанализировано время, которое выделяется на каждый ее этап, так как это время изменяется при изменении количества символов в обрабатываемом тексте. Из графиков на рис. 4 видно, что время вставки символов является относительно устойчивым – среднее значение равно $3{,}12 \times 10^{-8}$ миллисекунд. Что касается времени определения позиции для вставки символов, то эта величина возрастает пропорционально росту количества символов в тексте начиная от 5×10^{-9} до $6{,}23 \times 10^{-6}$ (то есть почти в тысячу раз). Причины: функция String.index (_: offsetBy :) выполняет важную для программиста работу – определяет позицию в тексте, учитывая количество code points, которые выделяются на один символ в тексте (как это было показано в примере с буквой Ä). То есть если текст состоит из десяти символов и необходимо вставить новый символ после символа номер 7, то функция String.index (_: offsetBy :) определяет позицию в тексте, в которой необходимо выполнить вставку символа, и возвращает именно количество символов, а не code points. Даже, если для отображения символа выделяется 3-4 юникодовских code points (например на флаги в символах Emoji), функция распознает Unicode последовательность и интерпретирует ее как один символ, то есть именно так, как видит перед собой пользователь. Это очень

полезно, в первую очередь не только потому, что не нужно распределять code points по символам "вручную", но и из-за невозможности вставки нового символа между code points, что преобразует его в другой символ и разрушит имеющийся. Но это имеет свою цену и требует прохода через весь текст, в котором нужно определить позицию для вставки символа. Итак, если каждый раз вызывать функцию String.index (_: offsetBy :) для вставки символа в n позицию от начала и каждый раз проходить линейно по тексту от начала до n, что, в свою очередь, будет занимать много времени на больших объемах текста с большим количеством вставок, то нужно искать подходы к оптимизации логики обработки текста. Основной целью такой оптимизации должно быть уменьшение текста, который должен быть обработан функцией String.index (_: offsetBy :).

Рис. 4. Время определения позиции для вставки и вставки символов, миллисекунд

После оптимизации алгоритма, применяемого для разбиения текста на слова, время работы программы снизилось с более чем двух минут до 800 миллисекунд, то есть скорость обработки текста увеличилась почти в 170 раз.

Итак, нами было исследовано разбиение текста на слова с помощью языка программирования Swift и ICU и получен неудовлетворительный результат, причина которого заключалась в использовании функции String.index (_: offsetBy :) ненадлежащим

образом, что и было исправлено. Можно сделать следующий вывод: сочетание Swift с ICU увеличивает быстродействие, что является преимуществом такой системы.

Разбиение текста на предложения

Разбиение текста на предложения используется для таких прикладных, ориентированных на пользователя задач, как, например, выделение предложения после тройного щелчка клавишей мыши на любом фрагменте предложения, на котором находится указатель мыши; выделение части текста, в которой более одного слова. Определение границ предложений даже в простом тексте не является однозначно решенной задачей. Например, точка, которую принято считать концом предложения в украинском правописании, может быть знаком в десятичной дроби, а кавычки в прямой речи в английских текстах могут указывать как на конец предложения, так и на окончание прямой речи без окончания основного предложения. Подобный пример приведен в Приложении 29 к стандарту Unicode [12]:

He said, “Are you going?” John shook his head.
“Are you going?” John asked.

Символы <?, “, пробел, прописная буква> в первом предложении примера являются указателями на конец предложения, а во втором первом предложении примера расположены в середине предложения. Без семантического анализа достаточно сложно правильно определить границы предложений, и даже после семантического анализа часто остаются неопределенности, но обычно, подход, реализованный в алгоритме разбиения текста на предложения ICU, работает удовлетворительно.

Очень важным моментом при определении границ предложений является их локализация, которая очень сильно влияет на результат работы алгоритмов ICU. Активация или деактивация какой-то части логики может быть выполнена как вручную разработчиком, который использует ICU, так и выполняться автоматически с помощью локализации. Для этого консорциумом Unicode был создан «Репозиторий данных общей локализации» [15], в котором можно найти информацию о том, как та или иная логика работает в зависимости от текущей локализации. Репозиторий данных общей локализации (CLDR - Unicode Common Locale Data Repository) широко используется корпорациями, продукты которых зависят от региональных стандартов. Вот краткий список организаций, который приводится на сайте CLDR:

- Google (Web Search, Chrome, Android, Adwords, Google+, Google Maps, Blogger, Google Analytics, …)
- IBM (DB2, Lotus, Websphere, Tivoli, Rational, AIX, i/OS, z/OS,…)
- Microsoft (Windows, Office, Visual Studio, …)

Кроме локализации в Приложении 29 к стандарту Unicode [12] также приведены знаки членов группы Sentence_Break (границы предложения), которыми обычно являются знаки препинания, перевод каретки и тому подобное. Перечень группы Sentense_Break можно найти в таблице 4 Приложения 29 к стандарту Unicode. Интересно, что там приведены и знаки препинания группы SContinue, сигнализирующие о том, что предложение не закончилось, например, двоеточие, запятая, дефис и тому подобное. Полный перечень этой группы можно найти в той же таблице 4.

Также важную роль в итераторе границ предложений ICU играют правила их идентификации. По сути они являются повторяющимся сочетанием членов группы Sentence_Break и служат для задания макросов (набора инструкций) для правила определения границ предложений. Например, макрос "прервать предложение после терминаторов (элементов, сигнализирующих остановку в контексте), но включать закрытия пунктуации, конечные пробелы и любые разделители абзацев" обозначается как множество SB9 ... SB11. Полный список макросов приведены в таблице 4a "Sentence_Break Rule Macros" в Приложении 29.

Как и в случае с итератором для слов, итератор для разбиения предложений был реализован с помощью библиотеки ICU4C-swift, где класс SentenceBreakCursor является оберткой над итератором границ предложений в ICU. Быстродействие логики ICU опять же показало себя достаточно хорошо, чего нельзя сказать о высокоуровневой логике из библиотеки Swift programming language для поиска позиции в тексте и для вставки текста в конкретную позицию, о которой упоминалось в разделе о разбиении текста на слова. Для анализа был использован тот же текст Конституции Украины. Результаты оказались хотя и значительно лучшими, но подобные предыдущим: с увеличением количества символов в тексте в 5,07 раз, время разбиения текста на предложения без вставки символов выросло в 3,78 раза, что даже несколько лучше, чем в случае со словами (таблица 2, рис. 5). Но время разбиения текста на предложения, со вставкой символов, выросло в 15,9 раз, что немного и лучше, чем в случае со словами, но все равно говорит о том, что высокоуровневую логику со Swift нужно реализовать по-другому. Важно понимать то, что не сама логика в Swift является медленной, а конкретная реализация с ее использованием нуждается в доработке.

Таблица 2. Зависимость времени разбиения текста на предложения от количества символов в тексте.

Количество знаков	Размер текста, байт	Время прохода по тексту, мс	Время со вставкой символов, мс	Количество предложений
26394	53590	1,9	24	474
67262	136534	3,6	107,6	1151
95074	193082	5,5	225,8	1651
133806	271536	7,2	382,4	2224

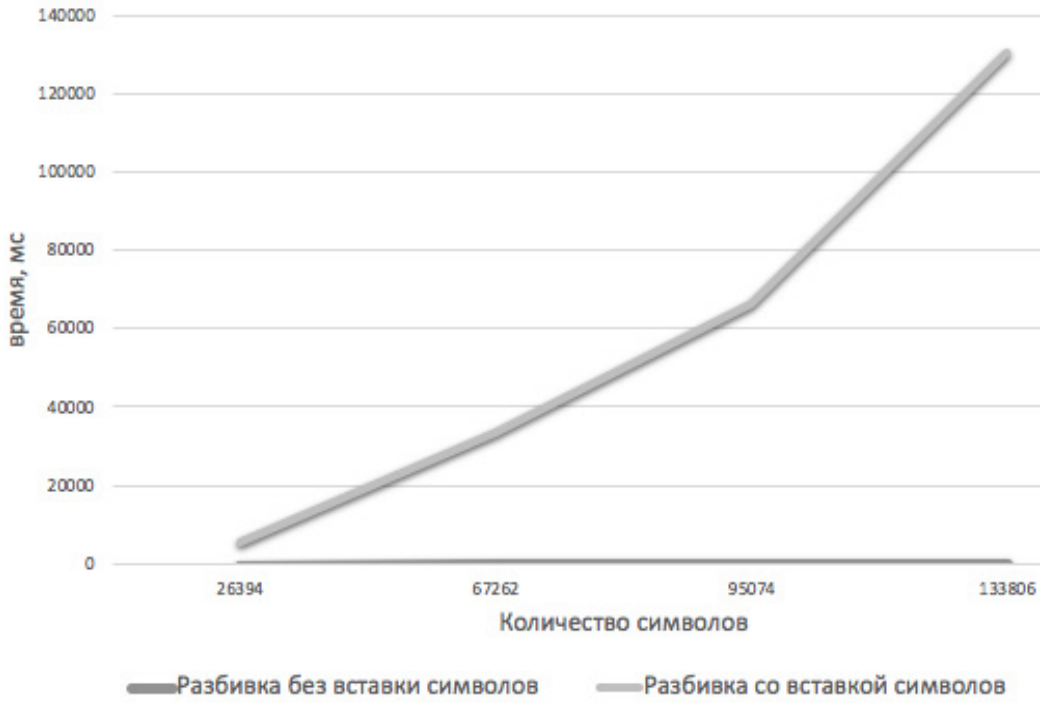


Рис. 5. Зависимость времени разбиения на предложения от количества символов в тексте

Подобно работе с определением границ слов в тексте, границы предложений размечаются определенными сигнальными знаками, используемыми для дальнейшей работы с размеченным текстом. Каждая вставка символов в текст – операция, требующая много времени. Поэтому алгоритмы разметки текста на предложения также были изменены для

того, чтобы определять позицию для вставки в наименьшем отрезке текста. После модификаций время со вставкой символов уменьшился с 382 миллисекунд до 10 миллисекунд, что существенно улучшило работу программы.

Для тех, кто использует Swift programming language для работы с текстами, рекомендуется определение позиции символа в тексте (Swift.String.Index) на минимальной длине текста. Если в качестве входного параметра выступает число, показывающее расстояние от определенной позиции в тексте к желаемой позиции вставки, то время поиска желаемой позиции растет пропорционально величине этого числа.

В Swift (версии 4) текст представлен как "текстовая величина, состоящая из совокупности символов Unicode" [16]. Структура String в Swift является объектом абстрактного типа коллекции, которая позволяет работать с текстом как с массивом (или множеством) символов, поэтому при смещении каретки рекомендуется смещать ее на минимальное расстояние и сохранять прежнюю позицию каретки после конкретного элемента массива.

Итак, подобно предыдущему разделу, было исследовано разбиение текста на предложения с помощью языка программирования Swift и ICU и получен неудовлетворительный результат. Скорость разбиения, конечно, была выше, чем при разбиении на слова, но все равно недостаточно высокой для обработки текстов больших объемов. Ситуация изменилась к лучшему после усовершенствования логики формирования индекса для вставки символа разметки. Снова можно сделать вывод, что сочетание Swift с ICU дает более высокое быстродействие, что также является преимуществом такой системы и для разбиения на предложения.

Выводы

В данной статье были рассмотрены исторические сведения об International Components for Unicode, система Unicode, сферы применения ICU, использование ICU в языке программирования Swift, а также приведены результаты испытаний использования ICU для разбивки текста на слова и предложения.

Историю кодирования символов в компьютерных системах можно свести к утверждению трех основных стандартов: ASCII, ANSI и Unicode, последний из которых стал таким, который вобрал в себя все символы алфавитов и систем письменности на Земле, что сыграло важную роль для адекватного обмена информацией в сети Internet.

Логика ICU используется в большинстве систем, работающих с датами и календарями, локалью, кодированием символов систем письменности различных языков (включая

системы с различным направлением чтения), что и обеспечивает набор для решения проблемы интернационализации и глобализации информационных технологий.

Язык программирования Swift использует логику ICU для работы с текстом, локалью и т.д, но поддерживает ограниченный набор функций, поэтому авторы рекомендуют использовать ICU как статически скомпилированную библиотеку, что будет удобно и безопасно при публикации программы в AppStore.

Разбиение текста на слова и предложения является достаточно сложной задачей с большим количеством правил и исключений, многие из которых реализованы в ICU, что и говорит о целесообразности использования этой системы. На языке программирования Swift удобно работать с текстом из-за того, что в нем он представлен как коллекция символов, состоящие из code-points. Поэтому разработчик работает с текстом, который он «видит». Важно правильно реализовывать формирования Swift.String.Index для вставки символов разметки, ведь неправильная реализация может увеличить время выполнения программы в сотни раз, как это видно из таблиц и графиков, приведенных в разделе «Использование International Components For Unicode для структурной разметки текста». Для разбиения текста использовались итераторы ICU, а логика программы была реализована на Swift.

Авторами были выявлены негативные аспекты конкретной реализации (как делать не следует) и даны рекомендации по минимизации времени выполнения программы (как необходимо делать).

Итак, если подытожить недостатки использования ICU, то они следующие:

1. требует много усилий для установки на OS Linux и для использования с языком программирования Swift;
2. специфическая реализация структуры String в Swift хоть и уберегает от опасности попасть между кодовых единиц Unicode одного символа, но требует формирования специфического индекса в тексте для позиционирования между символами;
3. при публикации продукта в AppStore его могут отклонить из-за использования приватных интерфейсов Apple, которые являются интерфейсами ICU.

Преимущества использования ICU:

1. кроссплатформенность;
2. содержит наборы символов, что является результатом работы над кодированием различных языков;

3. содержит фундаментальный функционал для работы с символьными данными, текстами (в частности, токенизации);
4. быстродействие.

Наличие единого "хранилища", которое могло бы вместить в себя все символы всех языков мира – серьезная проблема, с которой столкнулось человечество в процессе информационной революции. Отсутствие решения такой проблемы для систем информационно-коммуникационных технологий была бы равноценной коммуникации этих систем между собой "на разных языках". К счастью, Unicode не только стал этим единственным хранилищем, но и обеспечил единый механизм кодирования символов, и предоставил в определенной степени системам обмена информацией возможность идентичного отображения этой информации для человека с обеих сторон сети как до, так и после кодирования этой информации для ее передачи. International Components for Unicode, в свою очередь, на базе Unicode обеспечил дальнейшие шаги по направлению к интернационализации и глобализации, разработав стандарты и правила для работы с локалью календари и датами, границами в тексте и т.д., что обеспечило значительное ускорение перехода от информационной к сетевой эпохи человечества.

ЛИТЕРАТУРА

1. Feature Comparison Chart [Электронный ресурс] // International Components for Unicode – Режим доступа к ресурсу: http://site.icu-project.org/charts/comparison.
2. ICU-TC Home Page [Электронный ресурс] // International Components for Unicode – Режим доступа к ресурсу: http://site.icu-project.org/home#TOC-Who-Uses-ICU-
3. How Unicode relates to prior standards such as ASCII and EBCDIC [Электронный ресурс] // IBM Knowledge Center – Режим доступа к ресурсу: https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_72/nls/rbagsunicodeandprior.htm.
4. ICU User Guide. Boundary Analysis [Электронный ресурс] // International Components for Unicode – Режим доступа к ресурсу: http://userguide.icu-project.org/boundaryanalysis
5. TIOBE Index for January 2019 [Электронный ресурс] // TIOBE Software Quality Assessment Company. – 2019. – Режим доступа к ресурсу: https://www.tiobe.com/tiobe-index/.
6. Swift programming language [Электронный ресурс] // Apple Inc. – Режим доступа к ресурсу: https://developer.apple.com/swift/.

7. Natural Language Framework [Электронный ресурс] // Apple Inc. – Режим доступа к ресурсу: https://developer.apple.com/documentation/naturallanguage.
8. Server-side Swift - Perfect [Электронный ресурс] // Perfectly Soft – Режим доступа к ресурсу: https://perfect.org/.
9. Lower Level Text-Handling Technologies [Электронный ресурс] // Apple Inc. – Режим доступа к ресурсу: https://developer.apple.com/library/archive/documentation/StringsTextFonts/Conceptual/TextAndWebiPhoneOS/LowerLevelText-HandlingTechnologies/LowerLevelText-HandlingTechnologies.html.
10. ICU Text Transforms in Foundation [Электронный ресурс] // Ole Begemann's website – Режим доступа к ресурсу: https://oleb.net/blog/2016/01/icu-text-transforms/.
11. ICU for Swift [Электронный ресурс] // Tony Allevato – Режим доступа к ресурсу: https://github.com/allevato/icu-swift.
12. UNICODE TEXT SEGMENTATION [Электронный ресурс] // Unicode® Standard Annex #29 – Режим доступа к ресурсу: http://www.unicode.org/reports/tr29/#Word_Boundaries.
13. UNICODE LINE BREAKING ALGORITHM [Электронный ресурс] // Unicode® Standard Annex #14 – Режим доступа к ресурсу: http://www.unicode.org/reports/tr14/.
14. Swift programming language documentation [Электронный ресурс] // Apple Inc. – Режим доступа к ресурсу: https://developer.apple.com/documentation/swift.
15. Unicode Common Locale Data Repository [Электронный ресурс] // Unicode – Режим доступа к ресурсу: http://cldr.unicode.org/.
16. String type in Swift programming language [Электронный ресурс] // Apple Inc. – Режим доступа к ресурсу: https://developer.apple.com/documentation/swift/string.
17. Chronology of Unicode Version 1.0 [Электронный ресурс] // Unicode – Режим доступа к ресурсу: http://www.unicode.org/history/versionone.html?fbclid=IwAR2n_hc7ji3OHne0eJJyzLE-zmeoKZQfZ2tB0Y2LGo4XTZgFIN3KykEULII.
18. ICU Technical Committee [Электронный ресурс] // ICU - International Components for Unicode. – 1999. – Режим доступа к ресурсу: http://site.icu-project.org/projectinfo.

Информация об авторах

Коваль Юрий Владимирович *– аспирант, Украинский языково-информационный фонд НАН Украины, Киев, Украина. Тел. +380 (93) 202 2906. E-mail: yurii.smith@gmail.com.*

Сфера научных интересов: прикладная и компьютерная лингвистика, лексикография, системы обработки естественного языка (NLP), машинное обучение (ML), iOS и Swift разработка.

Надутенко Максим Викторович *– к.т.н., заведующий отделом информатики, Украинский языково-информационный фонд НАН Украины, Киев, Украина. E-mail: maxkrb@gmail.com.*

Сфера научных интересов: структурная, прикладная, математическая и компьютерная лингвистика, лексикография, когнитивные информационные технологии, лексикографические системы, системы профессионального взаимодействия в лингвистике, информационно-поисковые системы, онтологические системы и их применение в процессах поддержки принятия решений.