DOI 10.37882/2223-2966.2026.03.40

ТЕХНИКО-ФУНКЦИОНАЛЬНЫЙ АНАЛИЗ ГЕНЕРАТИВНЫХ ВЗАИМОСВЯЗЕЙ БИБЛИОТЕК-РАСШИРЕНИЙ «CELL-AUT.LSP» И «PIANOSYN.LSP — PIANO-NOTE» ЯЗЫКА ПРОГРАММИРОВАНИЯ NYQUIST В СИСТЕМЕ ИМИТАЦИОННОГО МОДЕЛИРОВАНИЯ АЛГОРИТМИЧЕСКИ-КОМБИНАТОРНЫХ АУДИОСИГНАЛОВ С ПРИМЕНЕНИЕМ ПРИНЦИПА КЛЕТОЧНЫХ АВТОМАТОВ

Таран Василий Васильевич
кандидат культурологии,
Лаборатория компьютерного дизайна
и прикладной информатики SPLASHLab
allscience@lenta.ru

Аннотация. В статье рассматриваются аспекты программно-генеративных взаимосвязей библиотек-*расширений* «cell-aut.lsp» и «pianosyn.lsp» (в контексте фрагмента «piano-note») языка программирования Nyquist с точки зрения технико-функционального анализа для повышения эффективности процедур имитационного моделирования аудиосигналов, а также для составления алгоритмически-комбинаторных композиций. Расшифрованы основные детали программного кода библиотеки-*расширения* «cell-aut.lsp» во взаимосвязи её функций с фрагментом библиотеки «pianosyn.lsp», отвечающей за синтез звуков фортепиано с имитационным моделированием различных метрико-акустических характеристик. В качестве механизма для получения новой фактуры полифонико-синтетических аудиоматериалов был выбран принцип клеточных автоматов. За основу модели, определяющей структурную топологию поведений элементарных клеточных автоматов, были взяты два *правила* Вольфрама 35 и 95. Раскрыты и на экспериментальной основе уточнены базовые свойства топологии поведения клеток (*ячеек*) на холсте клеточного автомата по определённым в настоящем исследовании *правилам* Вольфрама при получении новых синтезируемых образцов синтетических аудиокомпозиций. Получены новые экспериментальные результаты в области алгоритмизации и аудиопрограммирования, способствующие приращению новых инженерных и научных знаний в области аудиоинформатики и компьютерных наук. В заключении сформулированы итоги проведённого автором исследования.

Ключевые слова: язык программирования Nyquist, NyquistIDE, аудиопрограммирование, аудиоинформатика, компьютерные науки, параллельные вычисления, аудиосинтез, имитационное моделирование, алгоритмически-комбинаторные аудиосигналы, генеративные взаимосвязи, клеточные автоматы, *правила* Вольфрама — «35/95», библиотеки-*расширения* — «cell-aut.lsp» и «pianosyn.lsp (функция piano note)».

TECHNICAL AND FUNCTIONAL ANALYSIS OF THE GENERATIVE INTERRELATIONS OF THE LIBRARIES-EXTENSIONS «CELL-AUT.LSP» AND «PIANOSYN.LSP — PIANO-NOTE» OF THE NYQUIST PROGRAMMING LANGUAGE WITHIN THE SYSTEM OF IMITATION MODELING OF ALGORITHMIC-COMBINING AUDIO SIGNALS USING THE CELLULAR AUTOMATA PRINCIPLE

V. Taran

Summary. This paper examines the aspects of programmatically generated interrelations between the extension libraries «cell-aut.lsp» and «pianosyn.lsp» (within the context of the «piano-note» fragment) of the Nyquist programming language from the perspective of technical and functional analysis. The goal is to enhance the efficiency of procedures for imitation modeling of audio signals and to facilitate the development of algorithmically combinatorial compositions. Key details of the source code of the «cell-aut.lsp» extension library are decoded, revealing its functional relationship with the «pianosyn.lsp» library fragment, which is responsible for piano sound synthesis with imitation modeling of various metrical and acoustic characteristics. The cellular automata principle was adopted as the mechanism for generating new textures of polyphonic synthetic audio materials. The model, which defines the structural topology of the behaviors of elementary cellular automata, is based on two Wolfram rules: 35 and 95. The fundamental properties of cell behavior topology on the cellular automaton grid, according to the specific Wolfram rules, are elucidated and experimentally refined during the synthesis of new synthetic audio samples. New experimental results have been obtained in the fields of algorithmization and audio programming, contributing to the advancement of engineering and scientific knowledge in audio informatics and computer science. The paper concludes with a summary of the research conducted by the author.

Keywords: nyquist programming language, NyquistIDE, audio programming, audio informatics, computer science, parallel computing, sound synthesis, imitation modeling, algorithmically combinatorial audio signals, generative interrelations, cellular automata, Wolfram rules — «35/95, extension libraries — «cell-aut.lsp» and «pianosyn.lsp (piano note function)».

Пакет языка программирования Nyquist содержит необходимое количество библиотек-*расширений*, функционально обогащающих практику инженерной обработки, управления и программирования звука. Выбранная инженерами-проектировщиками тактика к реализации пакета NyquistIDE (Nyquist Integrated Development Environment) объясняется прежде всего тем, что технические манипуляции в сфере модификации аудиоматериалов, имеющих сложную неоднородную структуру, требуют определённой специфики подхода, так как такой тип аудиоматериала, в отличие от своего однородного аналога, имеет смешанные мультичастотные акустические характеристики, усложняющие процедуры цифрового многозадачного аудиопроцессинга [1]. Пакет NyquistIDE имеет программно-технические связи с редактором Audacity®, что по средствам *приглашения* позволяет запускать код на языке Nyquist внутри архитектуры данного редактора [2]. Эта опция позволяет вести инженерное редактирование звука программным путём с возможностью его дальнейшей визуальной интерпретации [3]. Концептуально это хорошо вписывается в бурно растущее технологическое развитие Всемирной паутины, поскольку в будущем традиционные программы всё сильнее и сильнее будут интегрированы в WEB-оболочку [4]. Известно, что классические библиотеки и библиотеки-*расширения*[1] в языках программирования играют *существенную* вспомогательную роль при построении различных архитектур программного обеспечения разноформатной направленности. Nyquist в этой связи не является исключением, его библиотеки призваны снизить нагрузку на разработчика (оператора рабочей станции, компьютера) при формировании прототипов, предполагающих повторяющиеся (*циклические*) операции, имеющие рутинные свойства. В этой статье речь пойдёт о библиотеках-*расширениях* cellaut.lsp и pianosyn.lsp, которые в контексте применения клеточных автоматов направлены на обеспечение утилитарных генеративных функций при решении задач, затрагивающих различные аспекты организационно-технической деятельности по производству новых *уникальных* фактур аудиоматериалов и инновационных методов аудиоисинтеза. Автор продолжает анализировать и экспериментировать на тему применения *правил* Вольфрама (Wolfram rules) для построения различных топологий поведения клеточных автоматов. Такой анализ необходим для расширения *теоретической* и *прикладной* исследовательской базы *аудиоинформатики*, а также для внедрения передовых практик инженерной обработки аудиоматериалов в обиход специалистов, занимающихся *аудиоинженерией* и инновационными техниками разработки специальных компьютерных программ [5,6,7,8,9,10,11,12]. В качестве модели генеративных алгоритмов[2] выбраны *правила* 35 и 95 с возможностью их базовой реализации в LISP-структуре [13,14]. LISP-структура универсальна для исполнения неявных команд внутри тела программы, так как Nyquist является её частью (диалектом), поэтому справедливо отметить, что *интерпретация* указанных *функций*[3] может быть также важна и для других проектов[4] [15,16,17,18,19]. Эти модели призваны продемонстрировать получение уникальных образцов аудиоматериала и могут быть использованы в качестве программных прототипов для составления различных алгоритмических конструкций

[1] Прим. автора. Библиотеки-***расширения*** (в информатике) — это составная часть концепции языков программирования. Они предназначены для многократного воспроизведения ряда технологических операций, опционально направленных на выполнение рутинных задач и предусматривают заранее заготовленные инструменты для осуществления циклов и адаптации внутрикодовой нотации при написании программ со сложной архитектурой. Привязка «***расширения***» означает, что та или иная библиотека языка программирования, соответствующая конкретному профилю решения задач, содержит более глубокие структуры кода, не присутствующие в оригинальной схеме и нотационной разметке любого программно-лексического средства. Такие расширения наполняют языки программирования дополнительным инструментарием, как правило, необходимым для оптимизации логических взаимосвязей кодовой нотации и различных итерационных последовательностей. Строго говоря, библиотеки нужны для упрощения выполнения определённых задач, с возможностью добавления новых функций или интеграции сторонних технологий в ту или иную программно-языковую среду (по определению автора, В.В. Таран — 2026.).

[2] Прим.автора. Генеративные алгоритмы — это специальный тип алгоритмов, предназначенных для создания новых данных, структурных комбинаций, объектов или решений на основе заданных правил, логических моделей или обучения. Такой тип алгоритмов, как правило, использует определённые параметры статистических и обученных моделей, с целью автоматического генерирования новых структур, таких как звуки, изображения, тексты и другие виды данных (по определению автора, В.В. Таран — 2026.).

[3] Прим.автора. Интерпретация функций в LISP — это процесс определения значения функции при её вызове и выполнении соответствующих вычислений. В LISP интерпретация включает в себя поведение программной схемотехники, а также выполнение выражений, где функции могут быть определены пользователем или встроены в структуру языка. Это позволяет LISP работать с ***динамической типизацией*** и возможностью обработки кода как данных, что делает его мощным инструментом для разработки и исследования различных программных архитектур (по определению автора, В.В. Таран — 2026.).

[4] Прим.автора. Универсальность проектов на LISP — это способность данного языка адаптироваться к различным задачам и областям имитационной деятельности благодаря разнообразию его диалектов. LISP обладает гибкостью, позволяющей использовать его для искусственного интеллекта, обработки сложных данных, разработки программных систем, автоматизации и многих других сфер. Его мощная концепция обработки кода как данных (***гомогенность***) и богатая поддержка ***метапрограммирования*** делают LISP ***универсальным*** инструментом, который можно ***настроить*** и ***расширить*** под ***конкретные потребности*** проекта в рамках различных диалектов, таких как Common Lisp, XLisp, Scheme, Nyquist и другие. При этом интерпретационная последовательность определения значения той или иной функции будет схожей, независимо от диалекта.

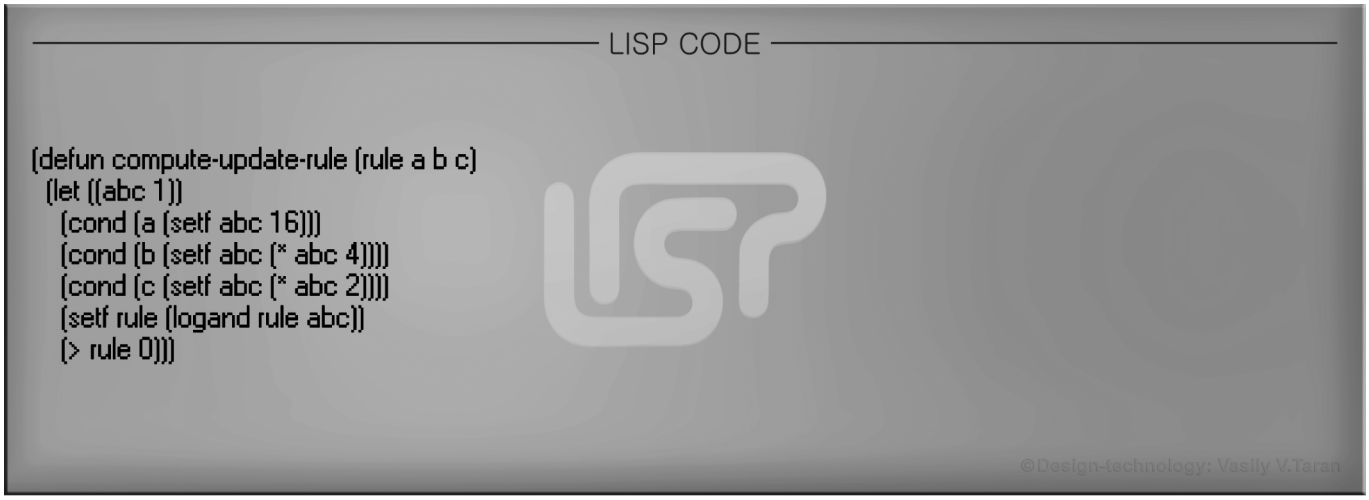

Рис. 1. Пролог, фрагмент кода функции cell-aut для генерации звуковых комбинаций на принципе клеточных автоматов (первичная логика обновления состояний ячеек, основанная на битовых масках)

и развития прикладного программирования в области научных исследований, связанных с аудиоинженерией [20].

Библиотеки-*расширения* cell-aut.lsp и pianosyn.lsp служат инструкциями для функционального отклика языка Nyquist. Чтобы глубже понять их работу, а также исследовать их опциональные функции, требуется провести технико-функциональный анализ[5] генеративных и программно-структурных взаимосвязей.

Для этого представим код функции cell-aut *фрагментарно*, с некоторыми пояснениями, а затем проанализируем её программную взаимосвязь с участком кода близкой по функционалу библиотеки pianosyn.lsp, отвечающего за воспроизведение различных метрико-акустических характеристик piano-note. Первый фрагмент — пролог программы, характеризующий *правило* обращения с битами (*битовыми масками*)[6] внутри библиотеки, рисунок №1.

Правило является восьмибитным (состоит из 8 *бит*). Здесь a, b, c используются для индексации битов *правила* и определения результата. Для этого необходима правильная позиция, например, если a = 0, b = 1, c = 1, то индекс равен 011 = 3, и нам нужен битовый шаблон с «1» в позиции 3: 00001000. Обратите внимание, что это 2 ^ 3 = a* 2 ^ 4 + b * 2 ^ 2 + c * 2 ^ 1.

Итак, мы можем проверить a, b, c и умножить на 16, 4, 2, чтобы получить правильную *последовательность* битов. Тогда эту же операцию мы можем сделать с помощью *правила*, чтобы в результате получить либо ноль, либо ненулевое значение. Расшифруем пролог *детальнее*:

Определение функции.

Имя функции: compute-update-rule

Аргументы: rule, a, b, c (все — *переменные*, передаваемые при вызове).

Локальная переменная.

В let создается локальная переменная abc, для которой по умолчанию присваивается значение «1».

Изменение abc в зависимости от *условий*:

cond — условная конструкция. Она проверяет усло-

[5] Прим.автора. Технико-функциональный анализ — это метод оценки (исследования) системы или объекта с точки зрения его ***технических*** характеристик, ***функциональных*** возможностей, структурных элементов и взаимосвязей между ними. Такой подход в научных исследованиях позволяет выявлять сильные и слабые стороны системы, определять её эффективность, надёжность и области для оптимизации или совершенствования. В контексте программных систем и технологий ***технико-функциональный анализ*** включает изучение архитектуры программного обеспечения, алгоритмов, закономерностей внутрикодовых операций, процедур верификации данных, интерфейсов, ресурсов и процессов, направленных на повышение производительности и качества работы компьютерных систем (по определению автора, В.В. Таран — 2026.).

[6] Прим.автора. Битовая маска — это число, в котором каждый бит (***битовая позиция, обращение с битами***) используется для хранения информации о каком-либо признаке или состоянии. Обычно битовая маска применяется для управления флагами, настройками или состояниями в программировании. Например, у нас есть 8 признаков, каждый из них можно представить одним битом в байте (8 бит). Значения 0 или 1 в отдельных битах показывают, включен ли признак или нет. Пример использования битовой маски: «00000001» — только первый признак включен, «00000101» — включены первый и третий признаки. Чтобы проверить, включен ли признак, используют побитовые «И» (&). Чтобы включить признак, используют побитовый «ИЛИ» (|). В области программирования это достаточно эффективный способ компактно хранить и обрабатывать множество булевых значений.

вие, и если оно истинно, выполняет соответствующий блок.

1.| (cond (a (setf abc 16)))[7]

[7] Прим.автора. В статье фрагменты программного кода библиотек-***расширений*** «cell-aut.lsp», «pianosyn.lsp (piano-note)» представлены без ***индентации****. Такой подход обусловлен ***удобством чтения кода*** в контексте обширных технико-функциональных (***текстовых***) комментариев. Фрагменты кода (***за исключением строк***) пронумерованы в целях оперативного обращения к тому или иному элементу языка программирования (литералы, переменные, списки, объявления, выражения, директивы, функции, процедуры и т.п.). Нумерация в структуре кода имеет специальный разделитель «|», позволяющий исключить ошибки при прочтении числовых массивов (целостность числовых массивов важна для осуществления операций счёта). Символ «–» — в ***комментариях***, обозначает строковой «***неразрывный***» пробел** (по Unicode U+00A0) присущий каждой строке в отдельности без автоматического переноса текста. Такое допущение необходимо для поддержания литературного стиля статьи. Символ «|#» — в комментариях, используется как начало блока комментариев, разделителя или метки для выделения определенного участка кода либо данных. Он служит маркером начала раздела комментария или примечания, это часть пользовательского стандарта — внутреннего соглашения. Символ введён для многострочных комментариев, таким образом «#|» — это начало комментария, а «|#» — его конец. Всё, что находится внутри #| ... |#, игнорируется интерпретатором. Следует помнить, что ***индентация*** в LISP не носит строго регламентированного характера и зависит от различных условностей построения кода, однако в некоторых диалектах её стоит придерживаться. В нашем случае код библиотеки-***расширения*** написан на диалекте LISP — (AutoLisp) с инкапсулированными вставками Common Lisp. Как и во многих других диалектах LISP хороший тон использования ***индентации*** равен 2-4 пробела для вложенных форм с левой стороны, но главное это сохранность скобочной структуры (она обязательна для всех диалектов LISP). В целях формирования удобочитаемости и разборчивости семантики LISP рекомендуем придерживаться следующих правил ***оформления*** кода:

1) Скобки должны располагаться в начале выражения.

2) Внутренние ***списки*** или ***вложенные выражения*** отступают вправо относительно ***родительского выражения***.

3) Обычно каждый элемент внутри ***списка*** пишется на ***новой строке*** с одинаковым уровнем отступа.

4) Величина отступа — обычно 2 или 4 пробела, в зависимости от стилевых рекомендаций или личных предпочтений программиста.

LISP является ***регистрозависимым*** языком. Это означает, что в нём различаются идентификаторы, написанные в верхнем и нижнем регистре. Например, переменные или функции с именами «foo», «Foo» и «FOO» считаются разными объектами. Для Common Lisp созданы некоторые инструменты форматирования (например, SLIME или SLY) которые подразумевают определённые правила оформления исполнимой символьной структуры в целях корректного ***парсинга*** кода. SLIME (Superior Lisp Interaction Mode for Emacs) — это специальный режим для Emacs, обеспечивающий глубокую интеграцию с различными LISP-реализациями. Он включает команду C-c C-f (или slime-format-defun), которая автоматически ***форматирует*** текущую функцию или выражение. Это большой плюс с точки зрения дизайна кода, так как: можно настраивать различные правила форматирования в автоматическом режиме, использовать стандартные правила LISP для отступов, обеспечить автоматическую расстановку скобок, отступов, произвести выравнивание элементов. SLY (Sylvester Lisp-y) — это аналог SLIME, также предназначен для Emacs, является современной модернизацией SLIME с расширенными функциями, имеет облегчённый и современный интерфейс, также позволяет форматировать код. Следует помнить, что при вводе закрывающей скобки или вызове команды TAB либо C-M-\ Emacs (по умолчанию) автоматически выравнивает текущее выражение. Если такой подход не срабатывает, применяем следующее выражение (define-key lisp-mode-map (kbd "TAB") ' indent-for-tab-command). Переменная lisp-indent-offset определяет размер отступа в LISP. По умолчанию — 2 пробела. Чтобы изменить на четыре пробела, применяем следующую конструкцию (setq lisp-indent-offset 2) → (setq lisp-indent-offset 2). Emacs — это мультифункциональный и настраиваемый текстовой редактор с богатой историей и большим сообществом пользователей. Он широко используется для программирования, редактирования текстов, написания сценариев и автоматизации рутинных задач. Реализация новых функций, а также разработка модулей возможна благодаря встроенной поддержке языка Emacs Lisp (ELisp).

* ***Индентация*** (в информатике, ***компьютерных науках*** — программировании) — это форматирование кода с помощью пробелов или табуляции, которое используется для: обозначения структуры программы (например, вложенности циклов, условий, функций), повышения общей читаемости и разборчивости сложных конструкций кода, наглядного отображения логической иерархии. Следует помнить, что в языках программирования, где отступы строго регламентированы (например, Python), некорректная ***индентация*** вызывает ошибки. В профессиональных практиках программирования существует термин ***акцидентная индентация*** — это ***непреднамеренные*** или случайные отступы в коде, которые появляются из-за ошибок либо несогласованности в оформлении. Это не часть стиля оформления кода, а результат случайных ошибок вызванных неправильной работой редактора, а также некорректным подходом к автоматическим процедурам ***дизайн***-формирования кода. Это редкое понятие в информатике, однако, с ним можно столкнуться при некорректном изложении кода, к примеру, при единовременном использовании разных стилей отступов (пробел+табуляция), а также в условиях процедур автоматической ***индентации*** с ошибками (неправильный перенос строк и т.п.).

** Неразрывный пробел используется для того, чтобы избежать ***автоматического*** переноса текста в нежелательных местах, например, между числом и единицей измерения, именами, ссылками и т.п. В LISP есть возможность использовать функцию make-string с кодом символа, выглядеть это будет следующим образом: (format nil "Nyquist~u" #x00A0) → (format nil " Nyquist~u" (code-char #x00A0)), в непечатных кавычках «""» указан пример использования неразрывного пробела. В статье (и в ***продолжение разработки темы*** в других исследованиях) автором приняты следующие ***условные обозначения***:

«⟶» — обозначение прямой (горизонтальной) логической взаимосвязи строк кода; «↔» — прямая и обратная (горизонтальная) логическая взаимосвязь строк кода; «↓» — прямая (вертикальная) взаимосвязь строк кода, а также некоторых вычислительных элементов используемых в мультиблоковой нотации; «↕» — прямая и обратная (вертикальная) логическая взаимосвязь строк кода; «…» — знак продолжения (цикл, итерация, любая логическая и математическая операция); «⟵» — обозначение прямой (горизонтальной) логической взаимосвязи строк кода, используется для указания обратной последовательности; «|» — разделитель нумерации строк кода («||» — двухуровневый раз-

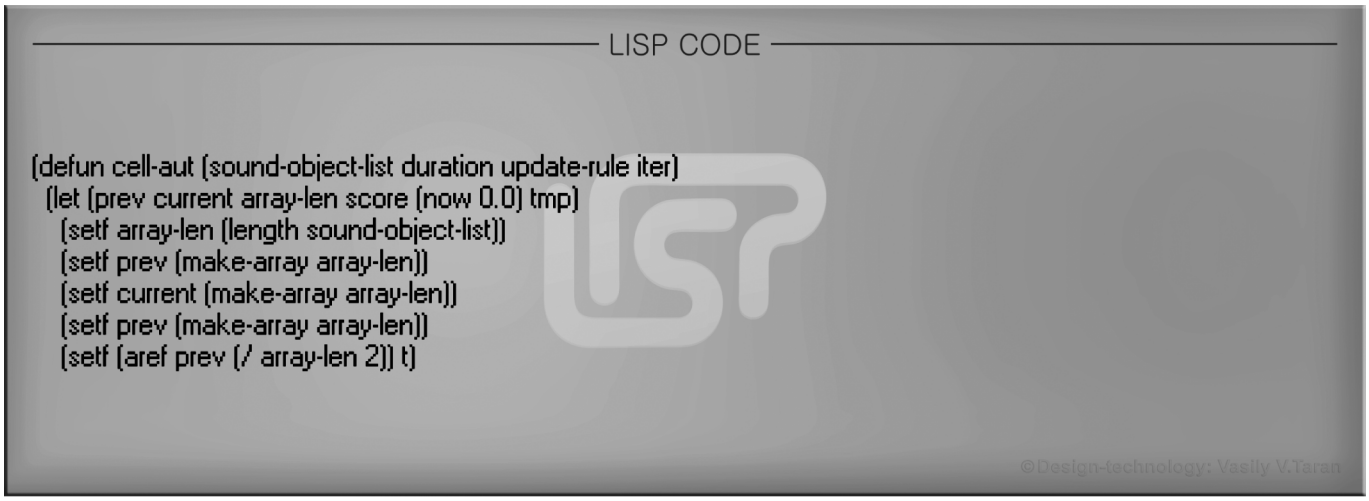

Рис. 2. Фрагмент кода функции cell-aut для генерации звуковых комбинаций на принципе клеточных автоматов (моделирование клеточного автомата)

— Если «a» *истина* (не nil), abc устанавливается в значение 16.

2.| (cond (b (setf abc (* abc 4))))

— Если «b» *истина*, abc умножается на 4.

3.| (cond (c (setf abc (* abc 2))))

— Если «c» *истина*, abc умножается на 2.

Обратите внимание: cond в LISP без:else (или t) — это проверка условий. В данном случае, поскольку условия просто a, b, c, то они рассматриваются как логические выражения. Если a, b, c — nil, условие *ложно*, иначе — *истинно*.

Обновление rule:

(setf rule (logand rule abc))

rule обновляется с помощью побитовой операции logand (логическое «И» по битам) между текущим значением rule и вычисленным abc.

Возврат значения:

(> rule 0)

Возвращается логическое выражение: истина (t), если rule больше 0, иначе — nil.

В зависимости от булевых элементов a, b, c, переменная abc принимает разные значения (1, 16, 64, 32). Затем rule обновляется с помощью побитового «И» с этим значением. В конце проверяется, является ли rule больше нуля, и возвращается t или nil. В прологе реализуется часть логики для обновления *правил* и состояний ячеек, основанных на битовых масках, с учётом условий a, b, c. Следующий раздел кода отвечает за моделирование структуры клеточных автоматов при инициализации массивов состояния, устанавливает локальные зоны ближайшей активной клетки в середине массива (рисунок №2).

Объявление функции:

cell-aut — название функции.

Аргументы:

1) sound-object-list — список «объектов» (может быть массив или список, зависит от использования).
2) duration — длительность (для модели времени).
3) update-rule — *правило* обновления (может быть либо функцией, либо данными).
4) *iter* — число итераций или цикл.

Объявление локальных переменных:

1) prev, current — массивы (осуществляют хранение топологии перехода состояний — текущее состояние, предшествующее состояние).
2) array-len — длина массива.
3) score — переменная для подсчёта или оценки.
4) (now 0.0) — начальное значение с плавающей точкой.
5) tmp — временная переменная.

Инициализация:

1) array-len устанавливается равным длине sound-object-list.
2) Создаются два массива prev и current с длиной array-len.

Повторное создание prev:

делитель вложенной нумерации строк кода, «|*|» — разделитель обратной нумерации строк кода). Непечатные кавычки используются только в структуре кода «"»». Печатные кавычки (литературные кавычки) используются по тексту статьи для поддержания норм и структуры русского языка, а также в целях выделения цитат особых (ключевых) слов, выражений и авторских смысловых конструкций.

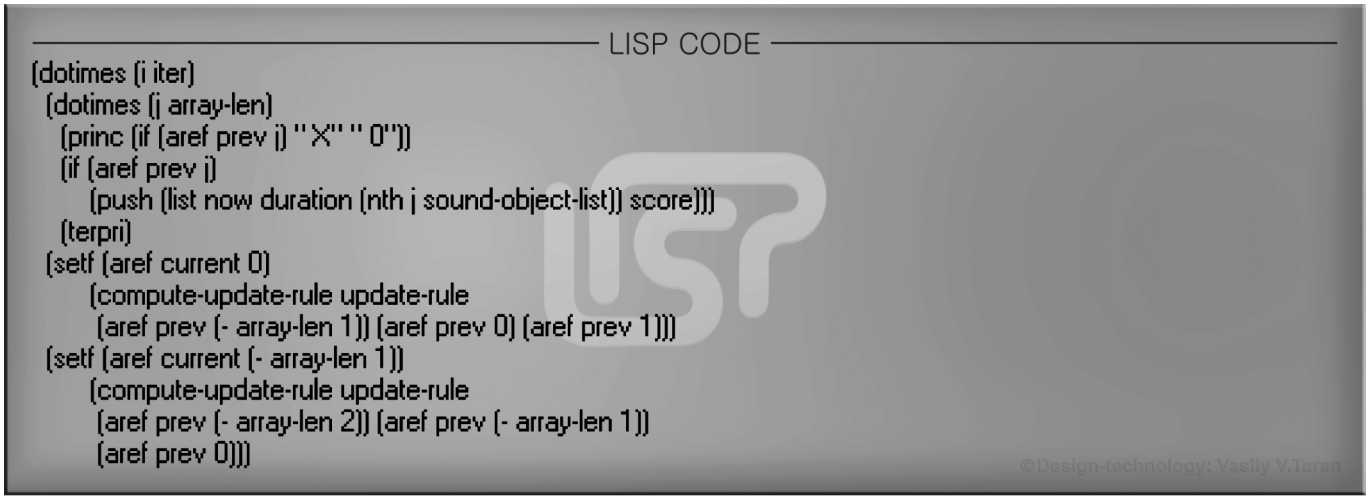

Рис. 3. Фрагмент кода функции cell-aut для генерации звуковых комбинаций на принципе клеточных автоматов (часть эмуляции клеточного автомата с циклическими границами, которая одновременно ведёт лог событий, например, учёт нот для музыкальной системы)

В строке (setf prev (make-array array-len)) создаётся новый массив и присваивается prev (*предыдущий* массив). Здесь есть повторное присваивание тому же *prev*[8].

Инициализация центрального элемента:

(aref prev (/ array-len 2)) — обращение к *центральному* элементу массива *prev* → (setf ... t) — устанавливает его в t (истина).

В целом это подготовительный фрагмент функции, которая моделирует клеточный автомат на этапах *инициализации* массивов состояния и установки начальной (*живой*) клетки посередине массива. Следующий фрагмент кода библиотеки, рисунок № 3, эмулирует клеточный автомат с циклическими границами.

Этот фрагмент кода — часть функции, которая реализует обновление состояния клеточного автомата. Она выполняется в цикле dotimes, повторяющемся *iter* раз, и внутри него есть вложенный цикл dotimes по индексам j, который проходит по элементам массива. *Внешний цикл*:

(dotimes (i iter) — повторяет весь блок *iter* раз, выполняя обновление состояния.

Внутренний цикл:

(dotimes (j array-len) — *итерируется* по индексам массива prev (current), длиной array-len.

Внутри цикла.

Вывод текущего состояния элемента:

(princ (if (aref prev j) « X» « 0»)) — Проверяет, истина ли элемент prev[j].

Если да — выводит «X».

1) Если нет — выводит «0» (пробел + ноль).

Далее следует добавление «ноты» в список (для характеристики звука):

(if (aref prev j)

(push (list now duration (nth j sound-object-list)) score))

Если prev[j] истинно, то добавляет в список *score* элемент вида:

list — из текущего времени now, длительности duration и объекта из *sound-object-list* по индексу j. Используется для записи событий (например, нот), связанных с *активными* клетками. Переход на новую строку (*новую строку вывода*):

(terpri)

Переходит на *новую строку* в *выводе*. После завершения внутреннего цикла следует обновление краевых элементов массива current:

```
1.| (setf (aref current 0)
2.| (compute-update-rule update-rule
3.| (aref prev (— array-len 1)) (aref prev 0) (aref prev 1)))
4.| (setf (aref current (— array-len 1))
5.| (compute-update-rule update-rule
6.| (aref prev (— array-len 2)) (aref prev (— array-len 1))
7.| (aref prev 0)))
```

Код выполняет обновление элементов массива current на основе значений массива prev и функции compute-update-rule.

[8] Прим.автора. Скорее всего, это лишний вызов — повторный вызов setf перезапишет ***первое*** обращение, и первый массив может ***утечь*** (если он не сохранён в другой локации). Вероятно, здесь удобнее использовать связку prev и current, а повторное создание может привести к замедлению операции.

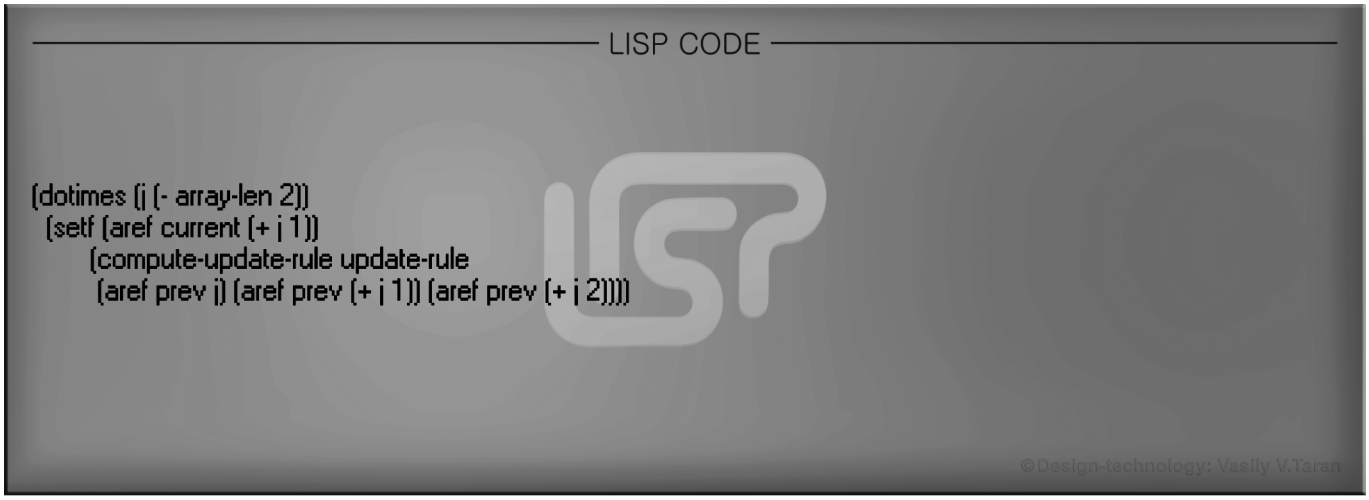

Рис. 4. Фрагмент кода функции cell-aut для генерации звуковых комбинаций на принципе клеточных автоматов (обновление внутренних элементов массива current (не крайние), основываясь на тройках соседних элементов массива prev с помощью *правила* compute-update-rule)

Обозначим:

1) current[i] — элемент массива current в позиции i;
2) prev[i] — элемент массива prev в позиции i;
3) len — длина массива array;
4) compute-update-rule (update_rule, a, b, c) — функция, применяемая к тройке значений.

Анализ фрагмента кода.

Первый элемент массива current (позиция 0):

current [0] =compute-update-rule(update_rule,prev[len −1],prev[0],prev[1])

Последний элемент массива current (позиция len-1):

current[len−1] =compute-update-rule (update_rule, prev[len−2],prev[len−1],prev[0])

Общий вид, учитывая, что индекс «-1» — это последний элемент массива, «-2» — предпоследний, и так далее.

Таким образом, итоговая конструкция будет выглядеть так:

$$\begin{cases} current[0] = \text{compute}- \text{update}- \\ -\text{rule}(update\_rule, prev[\text{len}-1], prev[0], prev[1]) \\ current[\text{len}-1] = \text{compute}- \text{update}- \\ -\text{rule}(update\_rule, prev[\text{len}-2], prev[\text{len}-1], prev[0]) \end{cases}$$

Обновляются крайние элементы current [0] и current[array-len-1]. Значения для них вычисляются с помощью функции compute-update-rule:

1) *для* current [0]: использует prev элементы: последний (prev[array-len-1]), первый (prev [0]), второй (prev[1]);
2) *для* current[array-len-1]: использует prev элементы: предпоследний (prev[array-len-2]), последний (prev[array-len-1]), первый (prev [0]).

Это делается для реализации циклических границ, то есть, массив воспринимается как кольцо (*цикл*). В каждом *iter*-ном шаге:

1). Выводятся текущие активные клетки (prev), отображая их как «X» или «0».
2). Для активных клеток добавляются записи в список score, связывая их с текущим временем и объектами.
3). Обновляются крайние клетки массива current, используя *правило* compute-update-rule и значения с «краёв» массива prev.

Фрагмент кода выше, рисунок № 4, производит обновление внутренних (не крайних) элементов массива current, основываясь на тройках соседних элементов массива prev с помощью *правила* контекстной функции compute-update-rule[9].

Данный фрагмент кода — часть обновления массива current внутри цикла, который реализует *правила* клеточного автомата с циклическими границами.

Цикл dotimes:

j[10] анализирует значения от 0 до array-len — 3 (всего array-len — 2 итераций).

Обновление current:

[9] Прим.автора. В диалектах LISP, функция compute-update-rule — это, как правило, пользовательская (контекстная функция), которая используется для вычисления правила обновления в алгоритмах машинного обучения, оптимизации и некоторых других итеративных процессах.

[10] Прим.автора. То есть, j — индекс, который идет по массиву prev, исключая последние два элемента.

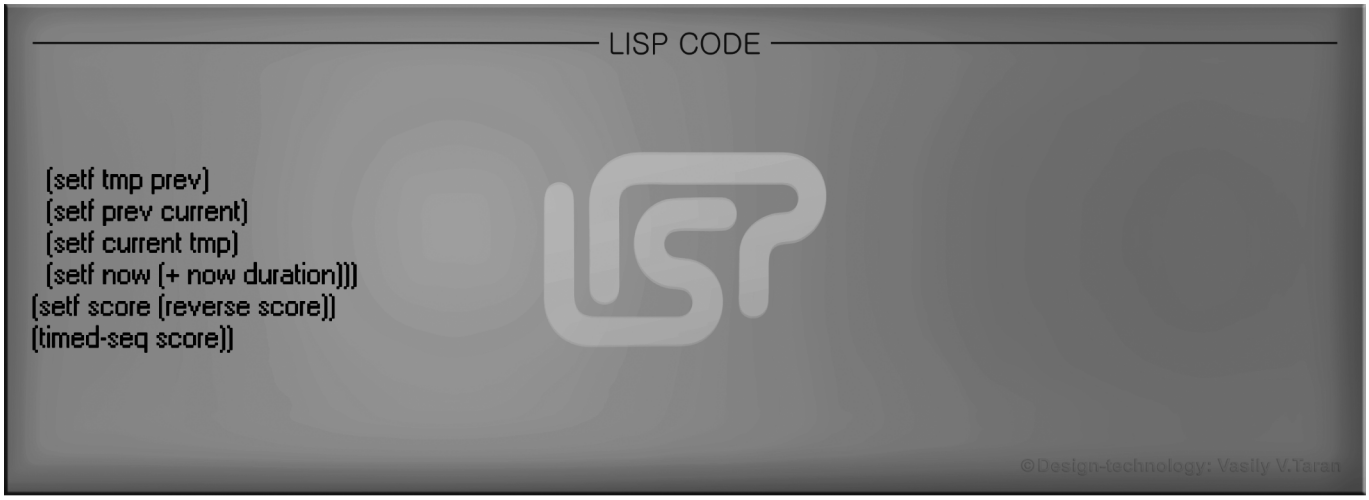

Рис. 5. Фрагмент кода функции cell-aut для генерации звуковых комбинаций на принципе клеточных автоматов (завершение итераций симуляции/генерации последовательностей и подготовка данных для дальнейшей обработки или воспроизведения)

Для каждого j обновляется элемент current[j+1].

Значение присваивается *результату* вызова функции compute-update-rule, которая применяет *правило обновления*.

Параметры compute-update-rule:
1). update-rule — *правило* (параметры *правила*);
2). prev[j] — текущий элемент массива *prev* на позиции j;
3). prev[j+1] — следующий элемент;
4). prev[j+2] — элемент следующий далее […2].

Это сдвиг или обновление элементов массива current, начиная со второго элемента (*индекс* 1), потому что current[j+1] — это позиция в массиве current[11]. Этот раздел кода обновляет внутренние элементы массива current (*не крайние*), основываясь на тройках соседних элементов массива *prev* с помощью *правила* compute-update-rule.

В данном случае это типичный подход при моделировании клеточных автоматов или подобных систем, где новые состояния элементов определяются их соседями.

Данный фрагмент кода (рисунок 5) — это часть *функционального цикла*, который управляет *обновлением состояний* и подготовкой данных для дальнейшей обработки или воспроизведения. Обмен ссылками на массивы:
1). tmp — временная переменная;
2). prev — предыдущий массив состояния;
3). current — текущий массив состояния.

Шаги:
1). tmp — сохраняет ссылку на prev;
2). prev — переназначается на current;
3). current — становится равен tmp (то есть, старому значению prev).

Вышесказанное — классический приём в области программирования для обмена значениями без копирования, чтобы подготовить массивы для следующего итерационного шага. Обновление времени:

(setf now (+ now duration))

Обновляет текущее время now, увеличивая его на duration, для формирования отсчёта прогресса времени в симуляции или последовательности. Следующая строка *переворачивает* список score:

(setf score (reverse score))

Score собирается в обратном порядке (с помощью push), и перед возвратом нужно его развернуть, чтобы получить правильный *временной* порядок событий. Далее осуществляется вызов функции timed-seq:

(timed-seq score)

Передаёт финальный список score. Функция timed-seq создаёт последовательность или объект, который использует временные метки и связанные с ними события (например, ноты, сигналы или другие события). Под-

[11] Прим.автора. В цикле идет обработка «треугольников» из трёх соседних элементов prev: [j], [j+1], [j+2]. Значения этих трёх элементов используются для вычисления нового значения для current[j+1]. Почему j идет до array-len - 2? Потому что j+2 не должен выходить за границы массива. В итоге последний j — это array-len - 3, а j+2 = array-len - 1 — последний индекс массива. Таким образом, весь диапазон покрывает все тройки соседних элементов по массиву prev, и для каждого из них вычисляется новое значение.

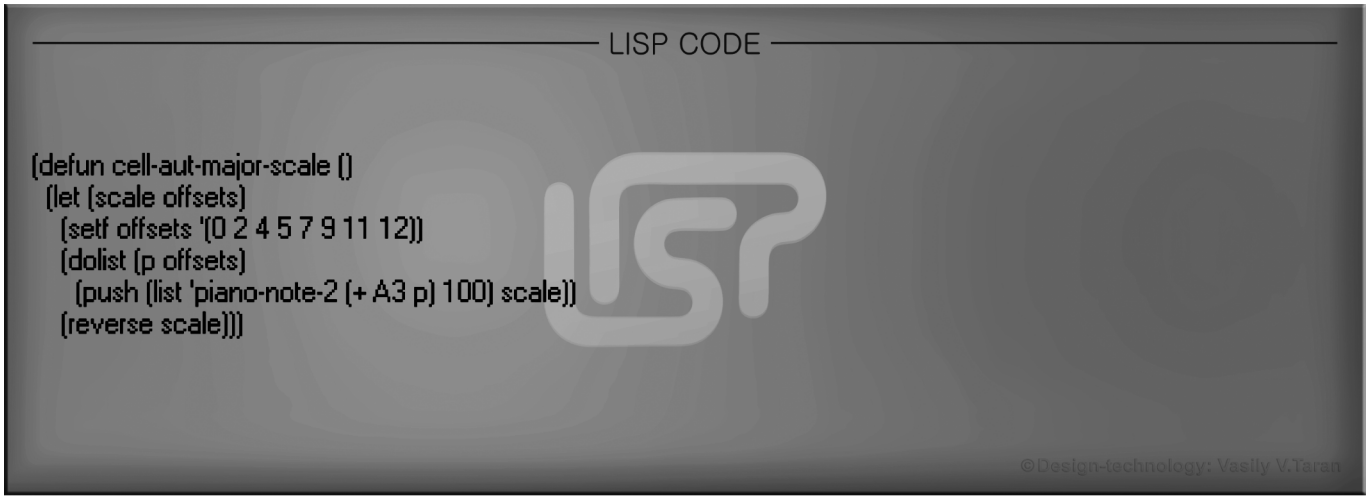

Рис. 6. Фрагмент кода функции cell-aut для генерации звуковых комбинаций на принципе клеточных автоматов (работа с нотами)

ведём промежуточный итог по части данного кода. Итак, данный блок выполняет *следующие действия*:

— Обменивает указатели на массивы, чтобы подготовить новые состояния для следующего цикла.
— Обновляет текущее время.
— Разворачивает список score, чтобы сохранить правильный порядок событий.
— Возвращает результат вызова timed-seq с этим списком.

Весь перечень действий необходим для завершения итераций *симуляции/генерации* последовательностей и подготовки данных для дальнейшей обработки или воспроизведения.

Код на рисунке № 6 определяет функцию cell-aut-major-scale, которая создает последовательность нот, соответствующую *мажорной* шкале, и возвращает их в виде *списка*. Разберём его более подробно:

1.| (defun cell-aut-major-scale ()
2.| (let (scale offsets)
3.| (setf offsets '(0 2 4 5 7 9 11 12))
4.| (dolist (p offsets)
5.| (push (list 'piano-note-2 (+ A3 p) 100) scale))
6.| (reverse scale)))

1) offsets = (0, 2, 4, 5, 7, 9, 11, 12) — фиксированный набор интервалов;
2) для каждого p в offsets создаётся нота: piano-note-2 (+ A3 p) 100;
3) итоговый список scale — это все подобные ноты в обратном порядке.

Это можно записать в виде следующей рекурсивной формулы:

$$scale = reverse(\{piano\text{-}note\text{-}2(A3 + p,100) | p \in (\{0,2,4,5,7,9,11,12\}\})$$

Работа функции.

Объявление локальных переменных:
1) scale — список для хранения нот.
2) offsets — список интервалов, соответствующих интервалам мажорной гаммы относительно тоники.

Инициализация offsets:

(setf offsets '(0 2 4 5 7 9 11 12))

Эти числа — полутона, соответствующие ступеням мажорной шкалы:

1). 0: тоника
2). 2: секунда мажорная
3). 4: терция
4). 5: кварта
5). 7: квинта
6). 9: секста
7). 11: септима
8). 12: октава

Генерация нот:
(dolist (p offsets)
(push (list 'piano-note-2 (+ A3 p) 100) scale))

Для каждого интервала p из offsets:
Создаётся список (list 'piano-note-2 (+ A3 p) 100)
'piano-note-2' — тип ноты (*вызов функции*).

(+ A3 p) — высота ноты, где A3 — *базовая нота*[*] (A3), а p — интервал.

«100» — *громкость* (в верхнем регистре длина ноты).

[*]*Эта нота добавляется в scale с помощью push.*

Обратный порядок:

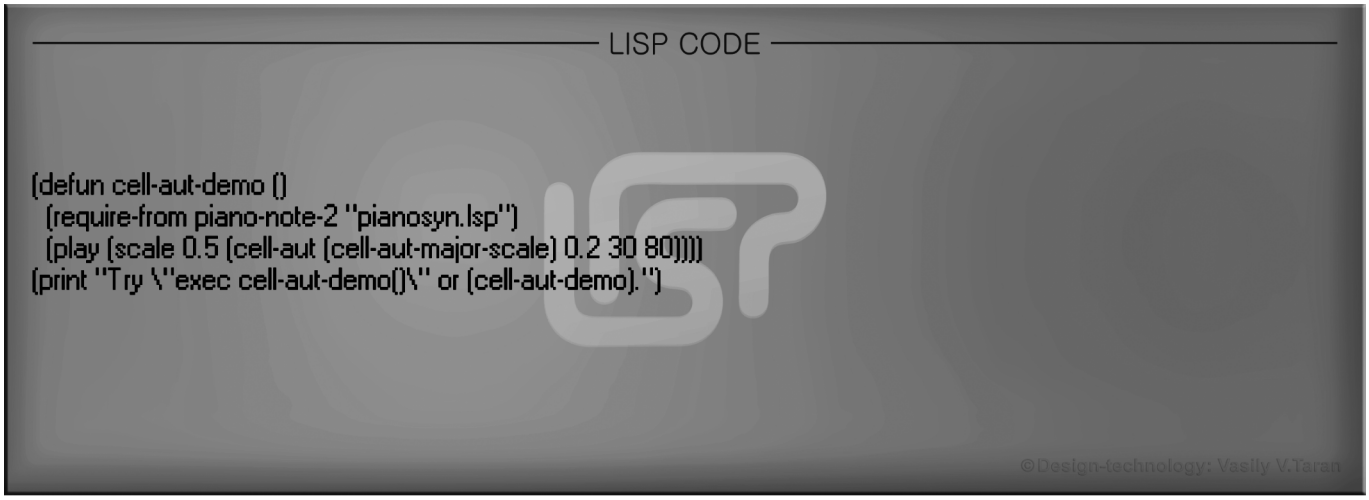

Рис. 7. Фрагмент кода функции cell-aut для генерации звуковых комбинаций на принципе клеточных автоматов (генерация музыкального паттерна с помощью клеточного автомата, основанного на мажорной гамме)

(*reverse* scale) — в конце возвращается список нот в правильном порядке (от тоники к октаве). В целом данный блок кода создаёт мажорную гамму, начиная с ноты A3, и возвращает список нот, представляющих эту гамму, готовых для воспроизведения или дальнейшей обработки. Выше, на рисунке № 7 показан фрагмент структуры кода, генерирующий музыкальный паттерн с помощью клеточного автомата, основанный на мажорной гамме.

1. *Объявление функции* cell-aut-demo

(defun cell-aut-demo ()

Здесь определяется функция с именем cell-aut-demo. Когда вы вызываете (cell-aut-demo), выполняется весь вложенный в неё код.

2. *Загрузка внешней библиотеки*

(require-from piano-note-2 «pianosyn.lsp»)

Эта строка загружает внешний файл pianosyn.lsp из источника piano-note-2. В ней содержится код, связанный с синтезом звука и работой с музыкальными нотами.

3. Воспроизведение музыкальной последовательности

(play (scale 0.5 (cell-aut (cell-aut-major-scale) 0.2 30 80)))[12]

Воспроизведение проходит несколько этапов:

- cell-aut-major-scale — генерирует начальную конфигурацию или паттерн для клеточного автомата, основанную на мажорной гамме.
- cell-aut — запускает клеточный автомат с этой конфигурацией и параметрами 0.2, 30, 80.
- scale 0.5 — масштабирует громкость или интенсивность звуковой последовательности, делая ее в два раза тише.
- play — воспроизводит полученную музыкальную последовательность.

4. *Вывод подсказки пользователю*

(print «Try \»exec cell-aut-demo()\» or (cell-aut-demo).»)

Печатает сообщение для пользователя, чтобы он знал, как запустить функцию.

В целом данный фрагмент кода определяет функцию, которая:

1). Загружает нужную библиотеку для синтеза звука.
2). Генерирует музыкальный паттерн с помощью клеточного автомата, основанного на мажорной гамме.
3). Воспроизводит его с *уменьшенной* громкостью.
4). И после этого *выводит сообщение*, как запустить функцию.

Теперь функционально рассмотрим фрагмент кода библиотеки pianosyn.lsp, обратимся непосредственно к piano-note (рисунок 8).

Код определяет функцию piano-note, которая создаёт и воспроизводит *музыкальную ноту* на пианино с заданной продолжительностью, высотой (тоническим интервалом) и динамикой (громкостью). Функционально это выглядит так:

Объявление функции:

(defun piano-note (duration pitch dynamic)

Функция принимает три аргумента:

1). duration — длительность ноты (например, целое число или символ, который позже интерпретируется).

[12] Прим.автора. Эти параметры управляют скоростью, количеством итераций и другими аспектами автоматики.

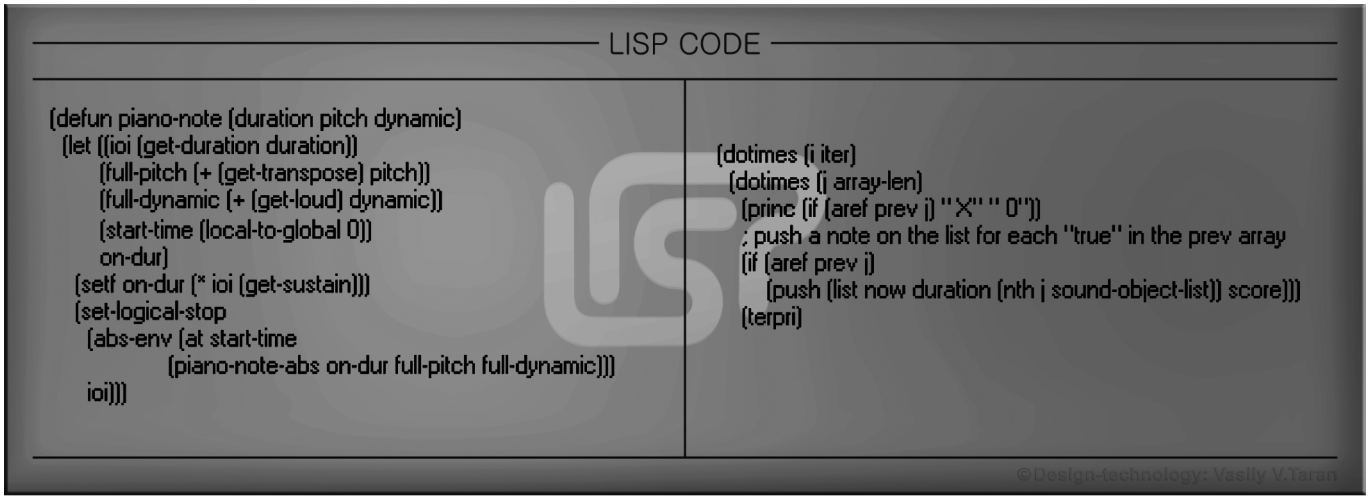

Рис. 8. Фрагмент кода библиотеки pianosyn.lsp для генерации звуковых комбинаций фортепьянных звуков по нотам — функция piano-note на принципе клеточных автоматов

2). pitch — высота ноты относительно базового тона.
3). dynamic — громкость или интенсивность исполнения.

Конструкции внутренних переменных (let):

```
1.| (let ((ioi (get-duration duration))
2.| (full-pitch (+ (get-transpose) pitch))
3.| (full-dynamic (+ (get-loud) dynamic))
4.| (start-time (local-to-global 0))
5.| on-dur)
```

Программная схемотехника переменных:

1) ioi = get_duration(*duration*)
2) full_pitch = get_transpose() + *pitch*
3) full_dynamic = get_load() + *dynamic*
4) start_time = local_to_global(0)
5) on_dur = значение, связанное с on-dur (не определено явно)

Расшифровка конструкции:

1). ioi — рассчитывает интервал между нотами (длительность) из аргумента duration, вызывая get-duration.
2). full-pitch — высота ноты, которая складывается из транспонирования (get-transpose) и переданного pitch.
3). full-dynamic — громкость, которая складывается из глобальной громкости (get-loud) и локальной *dynamic*.
4). start-time — время начала исполнения ноты, преобразованное из локального в глобальное (local-to-global 0).
5). on-dur — переменная для хранения продолжительности «звучания» ноты.

Расчёт продолжительности звучания ноты:

```
(setf on-dur (* ioi (get-sustain)))
```

В данном случае on-dur устанавливается как произведение ioi и get-sustain(). Это указывает, какая часть ноты будет звучать, основываясь на факторе сустейна (удержания). *Установка времени остановки ноты*:

```
1.| (set-logical-stop
2.| (abs-env (at start-time
3.| (piano-note-abs on-dur full-pitch full-dynamic)))
4.| ioi)
```

Опционально-технические характеристики:

1). at start-time — вызывает функцию в момент start-time.
2). piano-note-abs — создаёт абсолютное описание ноты (часто включает параметры высоты, громкости и длительности).
3). abs-env — оборачивает ноту в окружение, контролирующее её абсолютные параметры.
4). set-logical-stop — задаёт метрические параметры для завершения нот, основываясь на ioi.

Эта функция создаёт и воспроизводит ноту на пианино с заданной длительностью, высотой и динамикой, учитывая транспонирование, громкость и сустейн (удержание). Она управляет точкой начала и конца ноты, а также параметрами звука. Далее следует определение функции *piano-note*, которая создаёт и воспроизводит музыкальную ноту на пианино с заданной продолжительностью, высотой (*тоническим интервалом*) и динамикой (громкостью). *Объявление функции*:

```
(defun piano-note (duration pitch dynamic)
```

Функция принимает следующие три аргумента:

1). duration — длительность ноты (например, целое число или символ, который позже интерпретируется).
2). pitch — высота ноты, относительно базового тона.
3). dynamic — громкость (интенсивность исполнения).

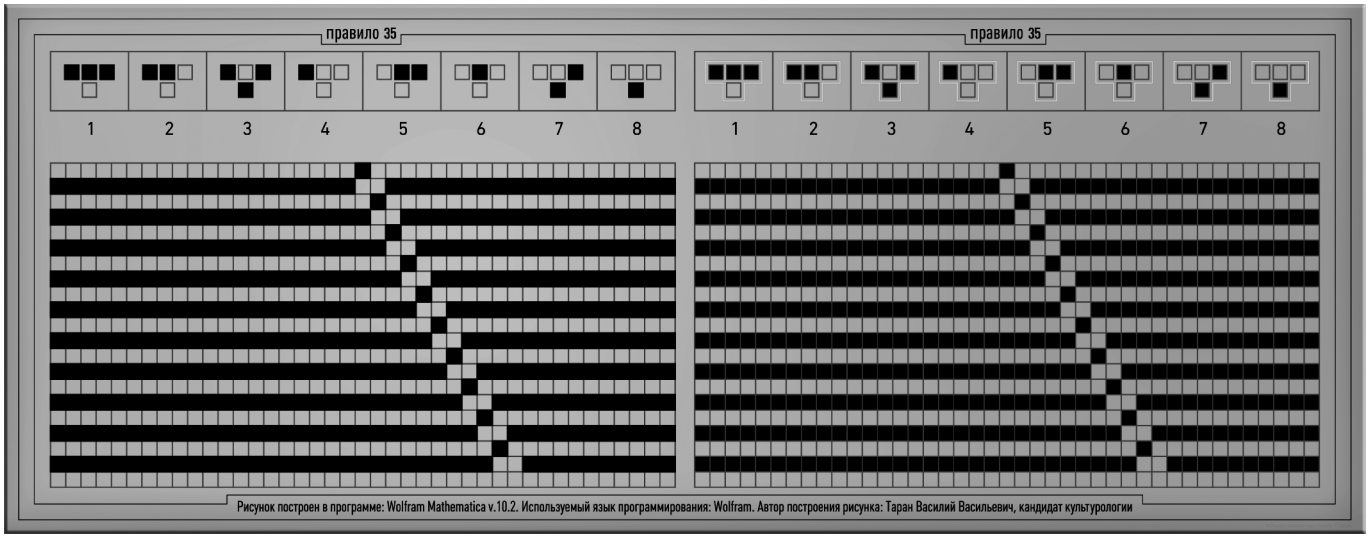

Рис. 9. Реализация воспроизведения аудиофрагмента, синтезирующего звуковую комбинацию по *правилу* 35 на принципе однородного[13] клеточного автомата в контексте обращения к коду библиотеки-*расширения* pianosyn.lsp

Внутренние переменные (let):

```
1.| (let ((ioi (get-duration duration))
2.| (full-pitch (+ (get-transpose) pitch))
3.| (full-dynamic (+ (get-loud) dynamic))
4.| (start-time (local-to-global 0))
5.| on-dur)
```

1) ioi = get_duration (*duration*)
2) full_pitch = get_transpose() + *pitch*
3) full_dynamic = get_loud() + *dynamic*
4) start_time = local_to_global(0)
5) on_dur = незначительно, *так как не определено явно*

Опционально-технические характеристики:

1). ioi — рассчитывает интервал между нотами (или длительность) из аргумента duration, вызывая get-duration.
2). full-pitch — высота ноты, которая складывается из транспонирования (get-transpose) и переданного pitch.
3). full-dynamic — громкость, которая складывается из глобальной громкости (get-loud) и локальной dynamic.
4). start-time — время начала исполнения ноты, преобразованное из локального в глобальное (local-to-global 0).
5). on-dur — переменная для хранения продолжительности «звучания» ноты.

Расчёт продолжительности звучания ноты:

```
(setf on-dur (* ioi (get-sustain)))
```

Здесь on-dur устанавливается как произведение ioi и get-sustain(). Это указывает, какая часть ноты будет звучать, основываясь на факторе сустейна (удержания).

Установка времени остановки ноты:

```
1.| (set-logical-stop
2.| (abs-env (at start-time
3.| (piano-note-abs on-dur full-pitch full-dynamic)))
4.| ioi)
```

Опционально-технические характеристики:

1). at start-time — вызывает функцию в момент start-time.
2). piano-note-abs — создаёт абсолютное описание ноты (включает параметры высоты, громкости и длительности).
3). abs-env — оборачивает ноту в окружение, контролирующее её абсолютные параметры.
4). set-logical-stop — задаёт метрические параметры для завершения нот, основываясь на ioi.

Подытоживая фрагмент структуры раздела данного кода, отметим, что он воспроизводит ноту на пианино с заданной длительностью, высотой и динамикой, учитывая транспонирование, громкость и сустейн (удержание). Она управляет точкой начала и конца ноты, а также параметрами звука. Проведя технико-функциональный анализ основной структуры программного кода, подключаемых библиотек-*расширений* языка программирования Nyquist, перейдём к изложению *базовой* логики работы клеточных автоматов по *правилам* 35 и 95, взаимодействующих на фрагментарном уровне с кодовыми конструкциями cell-aut.lsp и pianosyn.lsp. Вычисли-

[13] Прим.автора. Правило 35 — это правило обновления для однородного клеточного автомата, где все клетки используют одинаковое правило обновления, которое задаётся числом 35 (обычно по системе нумерации правил Вольфрама). Термин «однородный клеточный автомат» подразумевает автомат, в котором все клетки следуют одному и тому же правилу обновления. Правило 35 — это однородное клеточное правило, которое обновляет состояние каждой клетки в зависимости от тройки соседних клеток, согласно заданной двоичной схемы.

Рис. 10. Реализация (авторского) базового программного кода клеточного автомата с пиктограммно-блоковой легендой по *правилу* 35

тельная логика (рисунок № 9) и неявная программная схемотехника (рисунок № 10), иллюстрируемая нижеследующим разделом настоящей статьи, наглядно демонстрирует распределение топологий звуковых сигналов в виде клеток для получения новых уникальных образцов полифонических аудиоматериалов.

На рисунке 9 показана логика поведений аудиосигнала по *правилу* 35 в соответствии с функцией piano-note (рисунок № 8), обращение к этой функции позволяет в автоматическом режиме имитировать звучание фортепианной струны в зависимости от назначенной ноты. Параметры поведения ноты в однородном клеточном автомате регулируются фазой затухания, продолжительностью звучания, а также шириной диапазона частот. Ниже в таблице 1 приводятся общие координаты для построения *правил* поведения и топологического распределения аудиосигнала в клеточных автоматах.

Логика поведения аудиосигнала по *правилу* 35 будет выглядеть следующим образом:

Для одномерного клеточного автомата с тремя клетками (*левая*, *центральная*, *правая*) предполагается 8 конфигураций (таблица 2):

Представляем перечисленные конфигурации в *двоичном* коде.

Таблица 1.

Координаты вариативного построения клеточных автоматов по указанным в таблице *правилам**

Правило 35	Правило 95
Алгебраическая форма конструкции	
(p, q, r) ↦ ((1 + q) (1 + p + p r)) mod 2	(p, q, r) ↦ (1 + p r) mod 2
Форма булевых значений	
(p, q, r) ↦ (NOT q) AND (r OR (NOT p))	(p, q, r) ↦ NOT (p AND r)
Текстовая форма	
(p, q, r) ↦ ¬q ∧ (r ∨ ¬p)	(p, q, r) ↦ ¬ (p ∧ r)

** координаты используются для универсального построения базовой логики клеточных автоматов вне зависимости от языка программирования.*

Объединяем три бита (левый, центральный, правый) в один — в порядке слева направо (таблица 3):

Правило задаётся числом 35[14].

Переведём число 35 в двоичный вид с 8 битами (таблица 4):

[14] В двоичной системе 35 — это «00100011». Эта двоичная последовательность показывает, какие конфигурации приводят к следующему состоянию — «1», а какие — к «0».

Таблица 2.

Возможные конфигурации соседей

Левый сосед		Центральная клетка	Правый сосед
1.	1	1	1
2.	1	1	0
3.	1	0	1
4.	1	0	0
5.	0	1	1
6.	0	1	0
7.	0	0	1
8.	0	0	0

Таблица 3.

Конфигурации в двоичном виде

Конфигурация		Биты	Двоичное число	Десятичное число
1.	111	1 1 1	111	7
2.	110	1 1 0	110	6
3.	101	1 0 1	101	5
4.	100	1 0 0	100	4
5.	011	0 1 1	011	3
6.	010	0 1 0	010	2
7.	001	0 0 1	001	1
8.	000	0 0 0	000	0

Таблица 4.

Число 35 в двоичном виде с 8 битами

Бит позиции	7	6	5	4	3	2	1	0
Значение	0	0	1	0	0	0	1	1

Конфигурации:

1) 111 (позиция 7) — 0
2) 110 (6) — 0
3) 101 (5) — 1
4) 100 (4) — 0
5) 011 (3) — 0
6) 010 (2) — 0
7) 001 (1) — 1
8) 000 (0) — 1

Таблица 5 отображает для *каждой* конфигурации соседей новое состояние центральной клетки (*см*. таблицу 2).

Вычисление вероятности состояния центральной ячейки:

Таблица 5.

Таблица состояний по *правилу* 35

Конфигурация соседей		Бинарное представление	Новое состояние по *правилу* 35
1.	111	111	0
2.	110	110	0
3.	101	101	1
4.	100	100	0
5.	011	011	0
6.	010	010	0
7.	001	001	1
8.	000	000	1

new_state=(¬left∧¬center) ∨(center∧right)

1). left — состояние левой соседней ячейки.
2). center — текущее состояние центральной ячейки.
3). right — состояние правой соседней ячейки.
4). ¬ — отрицание (NOT).
5). ∧ — логическое «И» (AND).
6). ∨ — логическое «ИЛИ» (OR).

Вычисление:

1). Первая часть (¬left∧¬center) даёт 1, если оба соседа слева и центр равны 0.
2). Вторая часть (center∧right) даёт 1, если центр и правый сосед равны 1.
3). Общее выражение истинно (результат = 1), если истинна хотя бы одна из частей.

Ниже представлен *базовый программный код* клеточного автомата, воспроизводящий *правило* 35[15] (рисунок 10). Код обусловливает основную структуру клеточного автомата, его технические параметры и масштаб холста.

На рисунке 11 показана эволюция клеточного автомата по *правилу* 35.

Теперь обратимся к *правилу* 95, которое, по сути, представляет собой линейный двоичный автомат и характеризуется тем, что оно является простым и склонным к быстрой эволюции, распространяя единицы по линейной экспозиции, часто используется для моделирования процессов динамического (ускоренного) распространения, а также роста и захвата чего-либо

[15] Прим. автора. Поскольку логика изложения программного кода здесь схожа с конструкциями правил 30 и 90, то в целях экономии места, мы опускаем расшифровку кода, имеющего похожую структуру, поскольку его конструктивная часть подробно изложена в первой части статьи. Далее мы проиллюстрируем только логические операции правил работы клеточных автоматов.

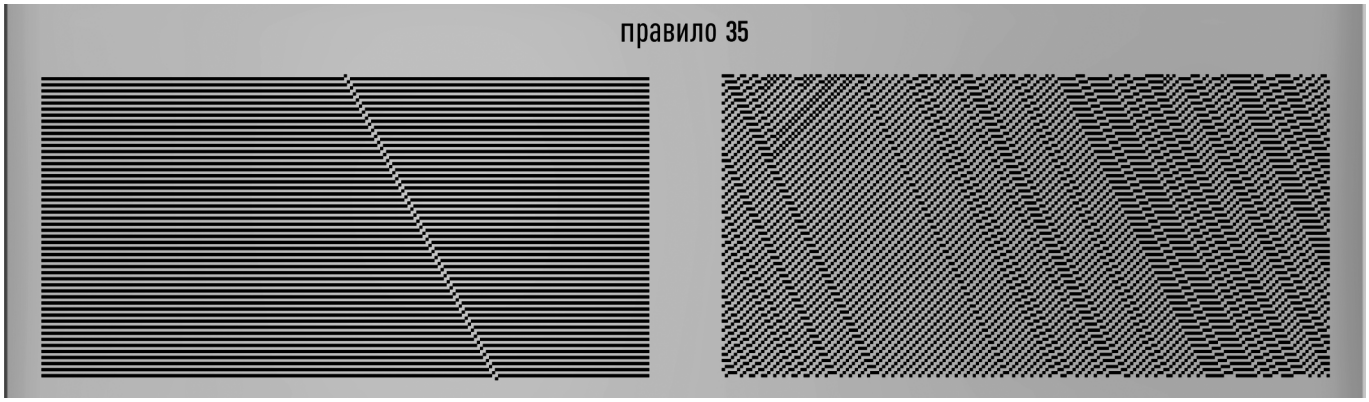

Рис. 11. Эволюция клеточного автомата по *правилу* 35

в различных системах. *Правило* 95 задаёт обновление положения клетки на основе её текущего состояния и локации соседних клеток. В контексте классических одномерных клеточных автоматов *правило* 95 определяется как *правило*, при котором новая клетка равна логической *дизъюнкции* ИЛИ (OR)[16] текущего состояния клетки и состояния её правого соседа. Вкратце, его логике можно придать следующий вид:

новое состояние = *текущее* состояние *или* «OR» состояние *правого соседа*

Правило 95 в теории клеточных автоматов применимо к моделированию систем, где важна динамика распространения, роста или захвата, а также в задачах, связанных с распространением информации или воздействием внешних факторов. В частности, его применяют для моделирования следующих ситуаций:

1) Моделирование распространения различных эпидемий:
— В медицине, биологии и информатике
- *Правило 95 моделирует быстрое распространение вируса по цепи, так как оно характеризуется высокой скоростью захвата и распространения единичных состояний (1) по линии.*

2) Моделирование процессов распространения информации:
— В информатике (кибермедицине, биоинформатике)
- *Используется для имитации ситуаций, когда наличие одного «носителя» информации быстро приводит к экспрессивной экспансии в ближайшей зоне локализации (распространению по всей системе).*

3) Моделирование процессов роста и захвата:
— В химии (биохимии, молекулярной биологии), кристаллографии
- *В системах, где важна тенденция к захвату или заполнению пространства единичными состояниями, например, моделирование роста кристаллов, распространение огня или других процессов, где «захват» происходит по принципу ИЛИ.*

4) Теоретические исследования и учебные цели:
— физико-математические науки, технические науки, биофизика, экспериментальная информатика;
- *используется для изучения свойств автоматов, таких как скорость распространения, устойчивость и переходные процессы.*

5) Моделирование процессов в теории сложности и самоорганизации:
— физико-математические науки, технические науки, теоретическая информатика, междисциплинарные и полидисциплинарные исследования;
- *правило 95 помогает понять, как локальные взаимодействия приводят к глобальным эффектам. Например, как локальные правила приводят к быстрому захвату всей системы.*

В аудиоинформатике *правило* 95 может использоваться в теории звука (моделирование звуковых волн) а также в некоторых прикладных исследованиях для создания уникальных звуковых эффектов, паттернов и генеративных композиций, благодаря своим уникальным свойствам распространения и динамики. Вот некоторые примеры использования данного *правила*:

1) Генерация ритмов и паттернов:
- *Используя правило 95, можно моделировать создание ритмических паттернов, где «захваченные» клетки (состояние 1) быстро распространяются по ритмической последовательности, создавая оригинальные, динамичные и непредсказуемые ритмы.*

2) Создание случайных или эволюционных звуковых текстур:
- *Автомат можно применять для генерации случайных последовательностей, которые затем преобразуются в звуковые волны или MIDI-сообщения, создавая уникальные звуковые текстуры, шумы или мелодии.*

3) Моделирование эффектов распространения звука:
- *Правило 95 можно использовать для моделирова-*

[16] Прим.автора. Новое состояние клетки определяется логической операцией OR (ИЛИ) между её текущим состоянием и локацией её правого соседа.

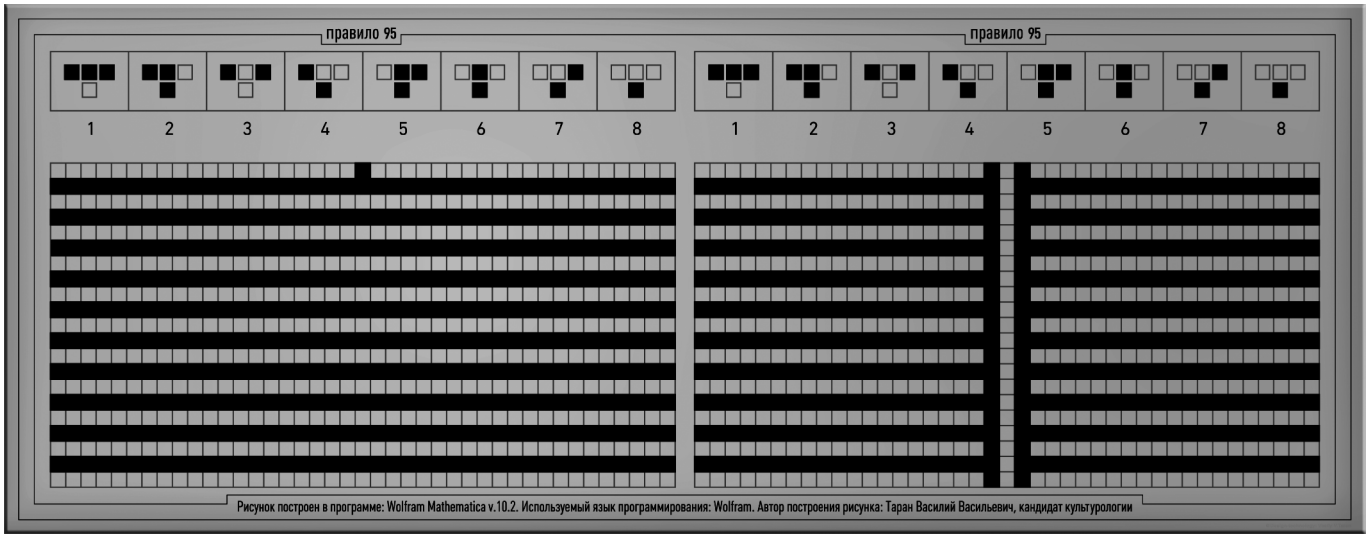

Рис. 12. Принцип работы клеточного автомата по *правилу* 95

ния распространения звуковых волн или эффектов «захвата» в акустических моделях. Например, для имитации эхообразных откликов или реверберации, где звуковые сигналы «захватывают» соседние области.

4) Генерация алгоритмических композиционных структур:

- *Можно создавать алгоритмы, в которых музыкальные события (ноты, аккорды) активируются и распространяются по цепочке в соответствии с автоматом, создавая сложные и не повторяющиеся паттерны.*

5) Визуально-звуковое представление:

- *Использовать автомат для визуализации процесса распространения в реальном времени, а звуки воспроизводить в такт с изменениями визуальных паттернов, создавая мультимедийные инсталляции.*

Вышеуказанные пункты показывают, насколько эффективно можно применять это *правило* для организации имитационных моделей в сфере программной звукотехники.

Логика поведения аудиосигнала по *правилу* 95 будет выглядеть следующим образом:

— Определим вероятные конфигурации топологий соседствующих объектов, таблица 6.

В таблице 6 определена структурная конфигурация соседствующих друг с другом элементов (*ячеек*). Это также одномерный клеточный автомат с тремя клетками (левая, *центральная*, правая) и здесь по-прежнему возможны только восемь конфигураций.

— Представим *правило* 95 в двоичной форме[17], таблица 7.

[17] Прим.автора. Число 95 в двоичной системе: 01011111.

Таблица 6.

Возможные конфигурации соседствующих объектов (ячеек)

Левая соседствующая клетка		Центральная клетка	Правая соседствующая клетка	Двоичное число	Десятичное число
1.	1	1	1	111	7
2.	1	1	0	110	6
3.	1	0	1	101	5
4.	1	0	0	100	4
5.	0	1	1	011	3
6.	0	1	0	010	2
7.	0	0	1	001	1
8.	0	0	0	000	0

Таблица 7.

Расположение битов в соответствии с топологией вероятных конфигураций соседствующих объектов

Конфигурация		Индекс (от 0 до 7)	Бит в двоичной записи	Значение
1.	111	7	0	0
2.	110	6	1	1
3.	101	5	0	0
4.	100	4	1	1
5.	011	3	1	1
6.	010	2	1	1
7.	001	1	1	1
8.	000	0	0	0

Таблица 8.

Таблица переходов, определяющих состояние активных клеток на ячейках холста

Конфигурация соседних клеток		Бинарное представление	Новое состояние по *правилу* 95
1.	111	111	0
2.	110	110	1
3.	101	101	0
4.	100	100	1
5.	011	011	1
6.	010	010	1
7.	001	001	1
8.	000	000	0

— Сформируем таблицу переходов для верификации состояний активных клеток на холсте, таблица 8.

Для каждой конфигурации соседствующих элементов таблица 8 показывает, какое новое состояние должна принять центральная клетка по *правилу* 95.

Логика вычисления вероятности состояния центральной ячейки:

$$new\_state=(left\wedge\neg center\wedge right)\vee(\neg left\wedge center)\vee(center\wedge\neg right)$$

где: left — состояние левой соседней ячейки, center — текущее состояние центральной ячейки, right — состояние правой соседней ячейки, ¬ — отрицание (NOT), ∧ — логическое «И» (AND), ∨ — логическое «ИЛИ» (OR).

Вычисление:

1). Первая часть (left∧¬center∧right) даёт 1, если left = 1, center = 0, right = 1.
2). Вторая часть (¬left∧center) даёт 1, если left = 0, center = 1 (независимо от right).
3). Третья часть (center∧¬right) даёт 1, если center = 1, right = 0 (независимо от left).
4). Общее выражение истинно (результат = 1) если истинна хотя бы одна из частей.

Ниже (на рисунке 13) представлен авторский программный код для построения полноценной структуры клеточного автомата по *правилу* 95. За ним следует рисунок 14, характеризующий эволюцию топологии состояний, поэтому же *правилу*.

Ниже на рисунке 14 показана эволюция клеток по *правилу* 95.

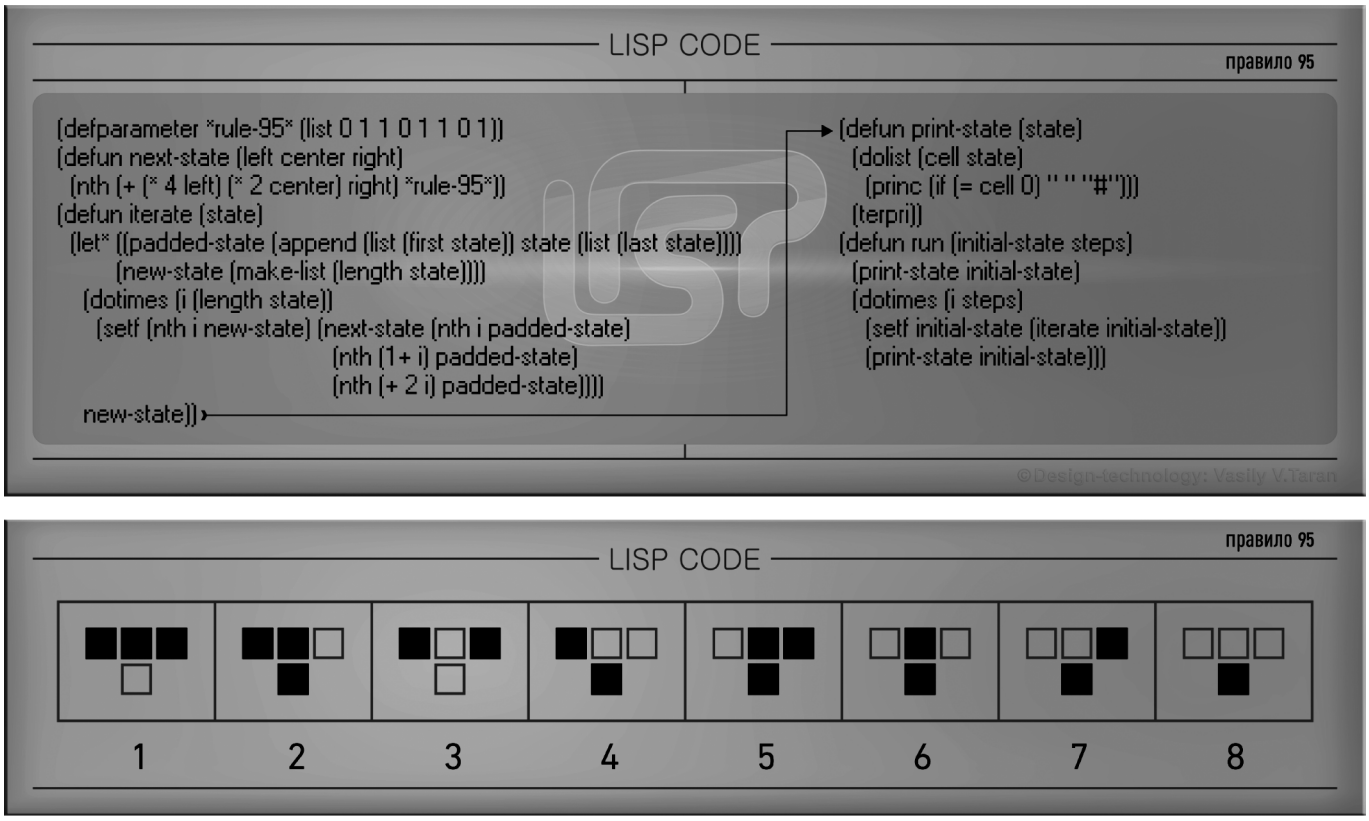

Рис. 13. Авторский программный код для построения полноценного клеточного автомата с пиктограммно-блоковой легендой по *правилу* 95

Рис. 14. Эволюция клеток (топология состояний) по *правилу* 95

В заключение отметим, что проблема создания оригинальных аудиофонических структур остаётся весьма актуальной, это показывает современное состояние развития аудиоинформатики, что также подтверждается весомыми аргументами, приведёнными автором в начале статьи. Клеточные автоматы и аудиосинтез, которые можно реализовать в соответствии с их принципами, могут усилить питательность исследовательской почвы, что, в свою очередь, откроет широкие горизонты для приращения научного знания в области компьютерных наук в целом и аудоинформатики в частности.

Проведённый технико-функциональный анализ в настоящем исследовании показывает, насколько важным является понимание хода вычислительных операций и процедур оперативной верификации данных для продолжения дальнейших научных изысканий по указанной автором теме.

Автор выражает надежду, что практики совершенствования составления алгоритмическо-программных конструкций синтетико-звуковых комбинаций, изложенные в данной рукописи, откроют новую страницу в истории программирования и вычислительной техники, обогатят научную картину и побудят различных исследователей к поиску сверхновых решений в области обработки звука.

ЛИТЕРАТУРА

1. Компьютерная программа Nyquist IDE v.3.15 / Файл директории (C:\Users\Name\nyquist) // OS: MS Windows, GNU Linux /// Полная реализация — Jesse Clark, David Hovard, David Movatt, David Deangelis, Roger B. Dannenberg. — 2002–2018. [электронный источник, автономная компьютерная программа].
2. Компьютерная программа Audacity® v.2.1.3 / Файл директории (C:\Program Files (x86) \Audacity) // OS: MS Windows, GNU Linux /// Полная реализация — Gale Andrews, Arturo «Buanzo», James Crook, Roger B. Dannenberg, Steve Daulton, Vaughan Johnson, Greg Kozikowski, Paul Licameli, Peter Sampson, Martyn Shaw, Bill Wharrie. — 1999–2017. [электронный источник, автономная компьютерная программа].
3. Официальный сайт редактора Audacity® — [электронный WEB-ресурс, дата обращения к ресурсу: 05.02.2026].
4. Таран В.В. О развитии концепции Всемирной паутины / В.В. Таран // Научно-техническая информация, серия 2. — 2019. — № 5. — С. 1–9.
5. Таран В.В. Проектирование дизайна аудиопродукции в программной среде Audacity® с применением языка Nyquist // Современная наука актуальные проблемы теории и практики. Серия Естественные и технические науки. — 2019. — №10. — С.159–171.
6. Таран В.В. Компьютерный аудиосинтез штатными средствами Audacity® с возможностью имитационного дизайн-моделирования на языке Nyquist // Современная наука актуальные проблемы теории и практики. Серия Естественные и технические науки. — 2020. — №1. — С.115–129.
7. Таран В.В. Язык программирования Nyquist: настоящее время и перспективы его развития в области компьютерной аудиоинженерии и аудиоинформатики // Современная наука актуальные проблемы теории и практики. Серия Естественные и технические науки — 2020. — №4. — С.135–153. DOI 10.37882/2223—2966.2020.04.37
8. Таран В.В. Компьютерная очистка аудиоматериала штатными средствами программы Audacity® (программно-ориентированный подход) // Современная наука актуальные проблемы теории и практики. Серия Естественные и технические науки. — 2020. — №9. — С.112–128. DOI 10.37882/2223—2966.2020.09.37
9. Таран В.В. Корректировка аудиосигнала при монтаже аудиозаписей в программной среде Audacity®, используя мультифункциональные возможности языка программирования Nyquist // Современная наука актуальные проблемы теории и практики. Серия Естественные и технические науки. — 2021. — №3. — С.155–202. DOI 10.37882/2223-2966.2021.03.32.
10. Таран В.В. Актуальные проблемы развития аудиоинженерии в России и их философско-техническое обоснование / В.В. Таран // Материалы X международной научно-практической конференции Фундаментальная наука и технологии — перспективные разработки [Технические науки], North Charleston, USA. — 2016 г., Том 1 стр. 120.
11. Таран В.В. Сравнительный анализ качества передаваемой информации форматами MP3, OGG, AIFF, WMA для оптимального выбора трансляции в Интернете / В.В. Таран // Материалы XVII международной научно-практической конференции Академическая наука — проблемы и достижения (Технические науки), North Charleston, USA. — 2018 г., Том 1, стр.78–94. [ISBN: 978–1729559000].

12. Таран В.В. Аудиомастеринг и интернет-телевидение: возможности для учреждений культуры и образования (доклад) / III Культурный Форум регионов России «Образование и культура: потенциал взаимодействия и ресурсы НКО в социокультурном развитии регионов России // Секция (мастер-класс): Цифровые технологии и дизайн в социокультурной сфере и образовании граждан, Москва — 22.10. 2017.
13. Touretzky, David S. Common LISP: a gentle introduction to symbolic computation /Carnegie Mellon University///Copyright (c) 1990 by Symbolic Technology, Ltd.//// Published by The Benjamin / Cummings Publishing Company, Inc. — 587 p. [ISBN 0-8053-0492-4].
14. Wolfram, S. A New Kind of Science. — Champaign: Wolfram Media, 2002. — 1197 p.
15. Dannenberg R.B. Nyquist Reference Manual Version 3.24 // Carnegie Mellon University — School of Computer Science / Pittsburgh, PA 15213, U.S.A. 05.03. 2025, 289 p.
16. Dannenberg R.B. Nyquist Reference Manual Version 3.16 // Carnegie Mellon University — School of Computer Science/ Pittsburgh, PA 15213, U.S.A. 2013–2020 WEB-version, URL: http://www.cs.cmu.edu/~rbd/doc/nyquist/ [дата обращения к электронному ресурсу: 05.02.2026].
17. Dannenberg R.B. Nyquist Reference Manual Version 3.15 // Carnegie Mellon University — School of Computer Science/ Pittsburgh, PA 15213, U.S.A. 11.08. 2018, 276 p.
18. Dannenberg R.B. Nyquist Reference Manual Version 3.09 // Carnegie Mellon University — School of Computer Science/ Pittsburgh, PA 15213, U.S.A. 27.12. 2014, 297 p.
19. Dannenberg R.B. Nyquist Reference Manual Version 2.36 // Carnegie Mellon University — School of Computer Science/ Pittsburgh, PA 15213, U.S.A. 05.03. 2007, 205 p.
20. Dannenberg R. Hu N. «Discovering Musical Structure in Audio Recordings» in Anagnostopoulou, Ferrand, and Smaill, eds., Music, and Artificial Intelligence: Second International Conference, ICMAI 2002, Edinburgh, Scotland, UK. Berlin: Springer, 2002. pp. 43–57.