Российская академия наук
Сибирское отделение
Институт систем
информатики

им. А. П. Ершова

Л.В. Городняя

СИНХРОН – ЯЗЫК ОЗНАКОМЛЕНИЯ С МИРОМ ПАРАЛЛЕЛИЗМА

Препринт

№

Новосибирск 2025

Препринт посвящен проекту разрабатываемого в ИСИ СО РАН языка обучения программированию СИНХРОН, предназначенному для начального ознакомления с понятиями и проблемами взаимодействия процессов и управления вычислениями. На этапе перехода к многопроцессорным архитектурам возрастает актуальность обучения параллельному программированию, что требует развития языково-информационной поддержки введения в программирование, включая гибридные методы работы с данными, показывающие разнообразие прагматики систем программирования. Язык СИНХРОН ориентирован на начальное обучение параллельному программированию школьников младших и средних классов, а также студентов младших курсов и непрофессионалов. Обучение опирается на опыт управления взаимодействием игрушечных роботов, перемещающихся на клетчатой доске. При иллюстрации методов представления программ, использующих параллелизм, приведены небольшие примеры, решения которых представлены табличной формой, позволяющей нагляднее проявлять специфику параллельных процессов. Показан эксперимент по использованию полиморфного синтаксиса, показывающий независимость синтаксиса от семантики и прагматики языка программирования, что может помочь профилактике известного эффекта второй системы. Препринт может быть интересен для программистов, студентов и аспирантов, специализирующихся в области системного и теоретического программирования, для всех, кто интересуется проблемами современной информатики, программирования и информационных технологий, особенно проблемами параллельных вычислений, суперкомпьютерами, распределёнными системами, мобильными устройствами и вообще применением многопроцессорных комплексов.

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

L.V. Gorodnyaya

SYNCHRON - PROGRAMMING LANGUAGE FOR ACQUAINTANCE WITH A WORLD OF PARALLELISM

Preprint

—

Novosibirsk 2025

The preprint is devoted to the SYNHRON programming language project, intended for initial acquaintance with the basic concepts of process interaction and computation management. At the stage of transition to multiprocessor architectures, the relevance of teaching parallel programming increases, which requires the development of language and information support for introduction to programming. The SYNHRON language is focused on initial teaching of parallel programming to primary and secondary school students, as well as junior students and non-professionals. The experience of controlling the interaction of toy robots moving along a checkered board is mainly used. To illustrate programs that use parallelism, small examples are given, the solutions of which are accompanied by a tabular form for a visual manifestation of the specifics of parallel process. The preprint is of interest to programmers, students and graduate students specializing in the field of system and theoretical programming. It can be useful to those who are interested in the problems of modern computer science, programming and information technology. First of all, this refers to the problems of parallel computing, supercomputers, distributed systems, mobile devices and, in general, the use of multiprocessor systems.

ВВЕДЕНИЕ

Пришло время изучать информатику и программирование, начиная с мира параллелизма. Ещё в начале 1960-ых при создании языка APL его автор, Кеннет Айверсон (*Kenneth Iverson)*, утверждал, что истинное программирование — это организация параллельных процессов [1]. Сколь ни сложен этот мир, системе подготовки программистов предстоит его освоить и создать методику полноценного ознакомления с его не очевидными явлениями, что и побудило к эксперименту по разработке учебного языка программирования СИНХРОН [2]. Алгоритмы становятся параллельными, программы — много-поточными, компьютеры, включая мобильные устройства, — многопроцессорными. Очередное действие может начаться до завершения предыдущего, средства управления кроме логических значений используют события и многое другое [3-5], что приводит к размежеванию синтаксиса и семантики параллельных вычислений на уровне выражений и допускает полиморфный синтаксис, мало влияющий на семантику [6, 7]. И всё это происходит в много-языковой обстановке — мощность пространства языков программирования уже перевалила за десятки тысяч [8, 9]. Язык СИНХРОН мультипарадигмальный с доминированием функционального программирования (ФП), обладающий полиморфным синтаксисом для выбора форм, способствующим снижению стартового барьера на первых шагах обучения и профилактике эффекта второй системы.

Первая цель опережающего ознакомления с понятиями и явлениями параллелизма — профилактика жесткого привыкания к принципам самого распространённого одно-процессорного, последовательного, императивно-процедурного программирования, осложняющего овладение новыми средствами, требующими знакомства с другими парадигмами программирования [10, 11]. Более широкое понимание программирования необходимо для специализации в области разработки распределенных информационных систем, а также приложений для суперкомпьютеров, многопроцессорных конфигураций, мобильных устройств и графических процессоров. Следует признать, что программирование и его техническая основа в последние десятилетия претерпели значительные изменения [12]. Дополнительной целью является приобретение опыта применения механизмов, актуальность которых растёт в связи с усложнением ИТ и расширением классов сложных задач, обременённых трудно удостоверяемыми требованиями, такими как живучесть, надёжность и безопасность программных инструментов.

Изложение начинается с общих положений (раздел 1) и обоснования решений по созданию и описанию идей учебного языка программирования языка СИНХРОН (раздел 2), затем описан рассматриваемый класс программ и в рамках полиморфного синтаксиса предложена табличная форма, делающая представления много-поточных программ более наглядными

(раздел 3). Далее приведена учебная метафора языка СИНХРОН, способствующая ознакомлению с параллельными взаимодействующими процессами в форме управления роботами, приведены примеры разных категорий учебных задач, пригодных для демонстрации проблем, возникающих при наивных попытках организации параллельных процессов (раздел 4). После этого дано краткое описание диалектов Асинхр, Синпар и Трансформ, соответствующих разным целям и уровням обучения (раздел 5) и приведены примеры, в представлении которых использованы четыре семантически эквивалентных формы синтаксиса, допускаемых определением языка (раздел 6). В заключении приведены выводы по образовательным проблемам параллельного программирования. В приложении 1 описана семантика и прагматика языка СИНХРОН в виде системы команд ядра языка, в приложении 2 описан синтаксис языка СИНХРОН в виде ряда усложняющихся диалектов, соответствующих этапам обучения, а в приложении 3 описан проект реализации языка СИНХРОН.

1. ОБЩИЕ ПОЛОЖЕНИЯ

Прежде всего, следует прояснить основные вопросы:

- Насколько и всегда ли изменяется и усложняется постановка задачи при переходе к параллельным алгоритмам и многопроцессорным комплексам?
- В какой мере при постановке задачи следует предварительно учитывать и выбирать модель параллельных вычислений и парадигмы параллельного программирования?

Ответим сразу, что постановка задачи в таком случае использует более сложную систему понятий, что может приводить к изменению методов решения задач и расширению пространства допустимых процессов выполнения программ. Благодаря этому появляется возможность выполнять усложнённые требования к решению важных задач. Нередко это может упрощать постановку задачи соответствием реальности и интуитивным пониманием процессов.

Предварительный выбор модели параллельных вычислений может оказаться неудачным из-за стремительного развития элементной базы и ИТ, особенности которого не всегда .удаётся предвидеть. Поэтому полезно предусматривать предстоящее развитие или уточнение выбора модели.

Более тридцати лет назад инициатива академика А. П. Ершова по обучению школьников информатике несколько обидела профессионалов, полагавших, что программирование — это занятие для людей с высшим образованием [13, 14]. Теперь детское программирование не редкость [15-17] и при оценке образовательного значения парадигм программирования признают базовыми императивно-процедурное, функциональное, логическое и ООП, фундаментальными – функциональное, параллельное и императивно-процедурное, а логическое, мета-программирование и ООП

рассматривают как дополнительные парадигмы, требующие знаний из областей приложения [18, 19]. Опережающее освоение параллелизма пока не получило признания, хотя реализация такой идеи возможна на базе учебных и новых языков программирования благодаря присущей им мультипарадигмальности [20].

Как это типично для учебных языков программирования, язык СИНХРОН поддерживает базовые и фундаментальные парадигмы программирования, включая параллельное программирование и мета-программирование. Использование разных парадигм приводит к разным формам представления много-поточных программ, что методически обусловлено профилактикой привыкания к одной форме, которая может тормозить переход к новым формам (Эффект второй системы) [21]. Программы в императивно-процедурной парадигме и ООП представляются в операторной форме, в функциональной парадигме и мета-программировании — в виде выражений, парадигма логического программирования использует предикатные формы и сопоставление с образцом, для параллельного программирования здесь предлагаются табличные формы, позволяющие зрительно разносить независимые потоки для много-поточных вычислений[1]. При синтаксическом различии эти формы семантически эквивалентны и прагматически равномощны, они порождают одно и то же семейство процессов.

Для представления много-поточных программ в операторной, функциональной и предикатной форме достаточно обычных синтаксических формул БНФ, хотя они слабо отражают семантику параллельных вычислений. Кое-что дает использование синтаксических функций и индексных грамматик, удобных для решения задач мета-программирования. Для повышения наглядности в процессе ознакомления с параллелизмом предлагается БНФ дополнить табличной формой, позволяющей использовать двумерные таблицы, приспособленные для наглядного представления много-поточных программ.

В табличной форме кроме обычных разделителей-символов рассматриваются клетки, записи и столбцы, допускающими контекстные зависимости. В клетке размещается неделимый фрагмент много-поточной программы, используемый целостно от начала до конца. В столбцах располагаются однородные очереди действий или фрагментов, соответствующих определённым семантическим мета-понятиям языка программирования. Записи бывают разных видов. Заглавная запись, первая в таблице, может содержать обозначения семантических мета-понятий, размещаемых в соответствующем столбце. Заглавных записей может быть несколько. При отсутствии заглавной записи по умолчанию предполагается, что первый столбец — это спусковые условия или имена, последний — комментарии, остальные столбцы — очереди используемых действий. Обычные записи представляют собой наборы действий, одновременно выполняемых в произвольном порядке. Для наглядности используются пустые записи, разделяющие слои многопоточной программы (Таблица 1).

1 Можно использовать текстовые редакторы для конвертации текстов в таблицы и обратно.

Таблица 1

Дополнение к БНФ для использования двумерных табличных форм.

Заглавная запись	Семантика предикатов	Семантики фрагментов в столбцах		Неисполняемый текст
Обычная запись	Условие	Исполнимое действие	. . .	Комментарий
Пустая запись	Разделитель между слоями программы			
Столбец	Имя	Фрагмент, соответствующий семантике столбца	. . .	Располагается вертикально, может содержать имена.
Клетка	Целостно используемый фрагмент			

Парадигма ООП в языке СИНХРОН необходима для решения задач с группами роботов. Она применяется без иерархии определений, допускает разложение решения задачи на шаги, использует механизм размеченных множеств, что несколько усложняет спектр структур данных, элементы которых могут быть представлены в рамках любых парадигм, любыми формами. Это требует хороших навыков абстрагирования, иллюстрируемых на задачах конструирования исполнителей без специальных форм.

Для разных семантически эквивалентных, порождающих одно и то же семейство процессов, в разных формах представления решений задач, приведённых в примерах раздела 6 будут использоваться четырёх клеточные таблицы, размещённые в матрице 2 на 2, подобно представлению решений задач по программированию в материалах для подготовки к ЕГЭ по информатике (Таблица 2).

Таблица 2

Четыре формы .представления одного и того же решения задачи

Табличная форма

Условие	Действия	Примечания
a = 4	**! мон**	*сначала прием монетки*
	? шок	*потом выдача шоколадки*

Операторная форма

! мон ;
? шок

Функциональная форма (Выражение)

(! мон ; ? шок)

Предикатная форма

! мон → ? шок

Проект учебного языка СИНХРОН нацелен на проявление или формирование интуитивных моделей при ознакомлении с основными явлениями и моделями параллельного программирования, встречающимися в учебно-методических и научных материалах [3, 22, 23], языках высокого

уровня (ЯВУ) [24, 25], языках высокопроизводительных вычислений (ЯВПВ) и механизмах сетевых информационных сервисов, образующих инструменты современной ИТ-индустрии. Такие модели необходимы для успешного обучения в любой практической области, в частности, для перехода к производственной деятельности и обеспечению надёжности и безопасности ИТ [26-28]. Следует обратить внимание, что при обучении не всегда более простую сущность изучить легче, чем более сложную. Реально простым воспринимается знакомое и привычное, а новое и непривычное кажется более сложным, даже если оно математически много проще.

Основные понятия языка СИНХРОН — фрагменты выражений, действий, вычислений и данных, включая процессоры, программы, языки программирования или системы. Многие понятия при переходе к параллельному программированию вызывают необходимость в дополнительных понятиях или их уточнении, а также в расширении синтаксических форм, используемых при определении языка параллельного программирования и представлении программ, работающих в более широком пространстве, чем обычные программы. Параллельные программы обладают более длительным жизненным циклом, чем последовательные. В учебном процессе язык предстаёт как череда диалектов, соответствующих фазам обучения. На уровне языка СИНХРОН предлагается следующая трактовка основных понятий:

Схема определяет варианты **управления** действиями в много-поточной **программе,** предназначена для параметризации барьеров, фрагментов и исполнителей **потока** (процессоров), а также предполагает наполнение **фрагментами**.

Управление – механизм принятия решений о возможности выполнения тех или иных **действий**, учитывающий **условия готовности** и расстановку **барьеров** в потоках..

Программа — это **комплекс/сценарий** из фрагментов, выполняющийся в заранее заданной **обстановке/контексте**, возможно с выделением значимых барьеров-синхронизаторов. Невыделенные барьеры не влияют на синхронизацтю потоков.

Поток — это очередь **действий**, возможно синхронизованных по одноименным барьерам с действиями из других потоков. Должен существовать **контекст**, в котором выполнимы все действия потока.

Фрагмент — представление любой сущности — действия, выражения, данного или произвольной их части, приспособленной для встраивания в схему управления программой.

Действия бывают двух видов — выражения и директивы, выражения используя память не изменяют её, а директивы могут память изменять.

Условия готовности означают, что определённое действие можно выполнить, не обязательно сразу, но когда-нибудь оно будет выполнено — проблема справедливости.

Барьер – метка для выделения позиций потока, в которых поток может ожидать взаимодействия с другими потоками.

Комплекс – сложная структура данных, позволяющая при необходимости именовать элементы и указывать кратность их вхождения в структуру или обстановку.

Обстановка — реализация контекста, содержащего перечень используемых объектов и исполнителей (процессоров) программы.

Набор состоит из действий, порядок выполнения которых не задан, но допускает одновременное выполнение на разных процессорах.

Контекст – бывает локальный контекст потока, защищённый от доступа из других потоков, и глобальный, реализуемый как общая **память**, доступная разным потокам и используемая при их взаимодействии.

Память - устройство для хранения программы и данных для её выполнения, включая поддержку взаимодействия потоков.

Обход дает возможность определять ветвящиеся процессы, подобен IF без ELSE.

Событие — данное, характеризующее изменение статуса выполняемых в программе действий или изменение обстановки выполнения программы.

Включение обеспечивает использование общих фрагментов потока или исполнителей или схемы.

Назначение позволяет распределять работу программы по процессорам.

Итерация обеспечивает многократность выполнения потока.

Рецепт — хранимый вместе с копией контекста сценарий, ждущий своего времени исполнения — замыкание функции.

По умолчанию обстановка содержит систему программирования (СП) на входном языке. Должен существовать контекст, в котором синхронизация набора потоков корректна, иначе программа выполняется до обнаружения невыполнимого действия и приостанавливается.

2. ВЫБОР РЕШЕНИЙ

Определение языка СИНХРОН потребовало заметного числа не очевидных решений в разных направлениях.

2.1. **Скрытая грамматика деятельности.** Актуальность обучения параллельному программированию требует не только развития языковой и системной поддержки учебных систем информатики, но ещё и пересмотра методик обучения, включения в них этапа формирования скрытой грамматики деятельности по организации процессов, для проявления которой предпринято создание и разработка языка начального обучения программированию СИНХРОН, предоставляющего средства экспериментирования и оперирования взаимодействующими параллельными процессами [29]. .

2.2. **Диалекты по целям обучения.** Учитывая большой диапазон уровней системы понятий параллельного программирования и длительность жизненного цикла много-поточных программ в проекте языка СИНХРОН выделены диалекты, соответствующие разным целям и фазам обучения:

- ознакомление с феноменами параллелизма (Мульти);
- приобретение навыков подготовки много-поточных программ (Асинхр);
- изучение методов представления взаимодействующих процессов (Синпар);
- эксперименты по многопроцессорному программированию (АПК);
- опыт мета-программирования на разных фазах обучения (Трансформ).

Описание синтаксиса языка, разделённое по диалектам, соответствующим этим целям (Синпар, Асинхр, Мульти, Трансформ), а также общая сводка, приведены в Приложении 2, вслед за Приложением 1, содержащим описание машины АПК.

В результате для решения разных классов учебных задач язык СИНХРОН допускает выделение подсистем разного уровня, таких как ядро или виртуальная машина (АПК), оболочки для представления отдельных категорий параллельных вычислений (Синпар, Асинхр, Мульти) и мета-преобразователь программ (Трансформ). Подсистемы могут функционировать автономно и совместно. Диалекты могут быть реализованы независимо, как отдельные системы программирования.

2.3. **Мультипарадигмальность** Сложность перехода от ранее сложившихся навыков императивно-процедурного программирования к организации параллельных процессов в значительной мере обусловлена разрывом между природными процессами формирования интуитивных моделей в реальной деятельности человека и системой обучения, требующей все упорядочивать, выстраивать в однозначные, безальтернативные, последовательные построения, без выражения зависимости программируемых решений от динамики выбора структур данных и порядка действий по их обработке. Языки XXI века, а также долгоживущие и учебные языки как

правило, поддерживают базовые и фундаментальные парадигмы программирования на ЯВУ, что позволяет программировать решения задач с разным уровнем изученности и с разной длительностью жизненного цикла постановки задачи в рамках единой языковой обстановки и получать навыки работы на всех фазах полного жизненного цикла программ в контексте одной системы программирования. Стихийно сформированные интуитивные модели обычно успешны при обучении императивно-процедурному и объектно-ориентированному программированию, подобному движению по одному маршруту. Они менее удобны при обучении функциональному, параллельному и логическому программированию, подобным обходу графа в ширину или движению сразу по всем возможным маршрутам, потому, что в обычной жизни прецеденты применения таких парадигм встречаются нечасто и обычно рассредоточено по времени — нет компактности для интуитивного проявления закономерностей. Возникают и новые парадигмы при появлении особо сложных, трудно решаемых задач [5, 30, 31].

Язык СИНХРОН поддерживает череду диалектов, соответствующих целям и фазам обучения при ознакомлении с параллелизмом и выполнении учебно-производственной практики для перехода к профессиональному программированию. Определение языка содержит диалекты Мульти, Трансформ, Асинхр, Синпар и АПК, соответствующие ознакомлению, преобразованиям, много-поточности, синхронизации и многопроцессности.

2.4. **Приоритет параллелизму.** Параллельные вычисления показали себя трудной задачей, требующей особой парадигмы, а такие парадигмы требуют предварительного освоения иллюстративного материала для целенаправленного формирования скрытых, интуитивных моделей новой парадигмы деятельности, отвечающей на вызовы новых проблем (см. 2.1.) [1, 6, 7, 12, 24]. Такое формирование поддержано диалектом Мульти, который можно рассматривать как язык оперирования передвижением фишек на клетчатой доске, похоже на языки Logo, Karel, Робик и систему Шпага.

2.5. **Функциональные языки параллельного программирования.** Языки параллельного программирования занимают видное место среди функциональных языков. Одна из причин заключается в том, что функциональные программы свободны от побочных эффектов и ошибок, непредсказуемо зависящих от реального времени. Это существенно упрощает отладку программ, что при обучении даёт поощрение выполнению ученых заданий. Кроме того, принципы универсальности, само-применимости и равноправия параметров помогают достигать удобочитаемость и лаконизм при подготовке программ. Это позволяет использовать разные решения по областям видимости имён, допускать выбор между статикой и динамикой, так было в языке Рапира.

История языков программирования накопила ряд работоспособных идей по эффективному представлению естественного параллелизма при серийной обработке сложных структур данных (APL Algol-68, Ada, БАРС, Sisal, Норма, mpC,)[2]. Для многих таких ЯВУ характерно включение в

2 Ссылки на материалы по языкам программирования не приведены, т. к. их не трудно

семантику языка одной конкретной модели параллелизма. Так, функциональный язык APL нацеливает на обобщение скалярных операций до обработки однородных векторов произвольной размерности. Algol-68 поддерживает управление процессами в терминах семафоров и критических участков, позволяющих параллелизм сводить к привычным последовательностям. Язык Ada предлагает механизм «рандеву», требующий для обмена данными одновременного существования процессов. Язык БАРС предлагает взаимодействия процессов представлять с помощью синхросетей над комплексами данных и выражений. Языки БАРС и Поляр используют сети управления процессами и программируемые дисциплины доступа к данным [24, 32-36]. Функциональный язык параллельного программирования Sisal предоставляет ряд средств формирования пространств итераций для параллельного исполнения циклов и для свертки полученных параллельно значений в общий результат [37, 38]. Язык Норма сосредоточен на проблеме представления вычислительных алгоритмов. Интересен язык mpC, нацеленный на представление программ для многопроцессорных конфигураций при выполнении много-поточных программ, допускающих синхронизацию с помощью барьеров и перераспределение нагрузки процессоров. Производственные инструменты MPI и Open MP связаны с определенными моделями параллелизма, поддержанными на уровне аппаратуры, что чревато трудоемкостью известной проблемы мобильности программ. Многие идеи организации процессов на уровне операционных систем слишком ограничены механизмами параллельной обработке на базе очередей и векторов [24].

Основные идеи ФП поддержаны диалектом Асинхр, учитывающим принципы универсальности функций, равноправия параметров, самоприменимости, гибкости ограничений, неизменяемости данных и строгости результата.

2.6. **Разнообразие архитектур.** Проблемы обучения параллельному программированию осложнены дистанцией между уровнем абстрактных понятий, в которых описываются решения сложных задач, и уровнем конкретных аппаратных средств и архитектурных принципов управления параллельными вычислениями (потактовая синхронизация, совмещение действий во VLIW-архитектурах, сигналы, семафоры, буферы со временем ожидания сообщения, прерывания и т. п.). Проблемы подготовки параллельных программ для всех столь разных моделей обладают общностью, но есть и существенная специфика, требующая понимания разницы в критериях оценки программ и информационных систем для различных применений. Разнообразие архитектур параллельных процессов на ЯВУ поддерживает диалект Синпар, дополняющий Асинхр средствами синхронизации с помощью барьеров и воздействиями на внешнюю память. Виртуальная машина АПК эту задачу решает как ЯНУ, отчасти допускающих императивное управление взаимодействиями процессов.

2.7. **Методическая обусловленность введения понятий.** Обычно

найти в Интернете

учебные языки программирования форсируют изучение базовых понятий поощрением первых успехов и форсируют переход от абстрактной алгоритмизации к практике отладки программ на производственных системах, в которых нет традиции выделять диалекты по уровням квалификации. При проектировании предлагаемого здесь языка СИНХРОН, точнее диалекта Мульти, учтены идеи наиболее известных, популярных учебных языков программирования, таких как Basic, Pascal, Logo, Grow, Karel, Робик, A++, Oz. Особенно важен опыт методически обусловленного введения понятий программирования в языке начального обучения программированию Робик [39]. Тщательная методически обусловленная проработка понятий в этом языке может служить базисом для любой профориентации учебного процесса по информатике, программированию, информационным технологиям и допускает естественное развитие для перехода к представлению взаимодействия асинхронных процессов. Следует отметить, что изначально семантика языка Робик содержала резерв для изучения мира параллелизма. Программа могла включать и выключать разных исполнителей, обладающих своими системами команд, и назначать исполнителей отдельных действий. Этот резерв не был востребован в конце 1970-х годов, но в наши дни он обретает актуальность.

2.8. В**заимодействующие процессы.** Чуть позже, в конце 1980-х, увидел свет перевод на русский язык превосходной книги блестящего авторитета Энтони Хоара (*Antony Hoare)* «Взаимодействующие последовательные процессы» [22], провозгласившего, что параллелизм — это вызов интеллекту человека. В своей книге на простых задачах управления автоматами Т. Хоар показал, что параллельные композиции внешне, на уровне синтаксиса, не сложнее последовательных, если не акцентировать внимание на том, что отладка, т. е. семантика параллельных процессов, много сложнее. Кроме того, такие внешне похожие композиции выражений срывают разницу в прагматике — мощности семейства допустимых процессов, что ниже показано на примерах (табл. 1 и 2). Именно таким образом простота пользовательских интерфейсов или удобство синтаксиса маскируют истинную сложность программ и процессов вычислений.

Модель Т. Хоара чувствительна к обнаружению достаточно тонких ошибок при создании сложных программ, её нередко используют в системах верификации свойств программ, что способствует и раннему ознакомлению с техникой верификации программ, распознавания их свойств. Поэтому здесь примеры Т. Хоара дополнены рассмотрением вопросов отладки и методов минимизации ее объёма с помощью выделения типовых, многократно используемых фрагментов, реализуемых с учётом синтаксической правильности результата подстановки фрагментных переменных [40-43], что можно рассматривать как задачу выбора форм для представления и организации семейств допустимых параллельных процессов.

Разработка программ для организации взаимодействия процессов

отличается от подготовки обычных программ на весьма глубоком уровне, что можно показать на модели интерпретирующего автомата для языка управления процессами [25]. Такой автомат требует реализации дополнительной таблицы событий и структуры данных для очередей, регулирующих доступ к данным. Асинхронные много-поточные программы представляются в рамках диалекта Асинхр, некоторые возможности синхронизации предоставляет диалект Синпар и виртуальная машина АПК.

2.9. **Расширяемость языков и систем программирования.** При определении реализации языка СИНХРОН, использованы идеи языков Lisp, F#, C#, допускающих динамическую обработку кода программ и в рамках парадигмы ФП выполнять переход к парадигме рефлексивного программирования, полезной для создания программ для мобильных устройств. Это позволяет обеспечить лаконизм представления программ, облегчить отладку программ и упростить реализацию интерпретатора, компилятора и виртуальной машины языка.

Следует отметить, что полная парадигма программирования представляет собой расширяющийся ряд отдельных парадигм по мере появления новых трудно решаемых задач. Это ставит проблему расширяемости языка и системы программирования по мере развития средств и методов параллельных вычислений, обусловленную высоким темпом прогресса в области элементной базы и информационных технологий в целом. Многие понятия языка программирования — схемы управления программой и образующие программу действия, вычисления и данные — при переходе к параллельному программированию претерпевают изменения, и возникает необходимость в дополнительных понятиях [44]. Возможность расширения программ и диалектов языка СИНХРОН поддержана диалектом Трансформ.

2.10. **Ленивые вычисления и мемоизация.** Результативность вычислений можно повышать с помощью специально устроенных данных — так называемых «рецептов» или замыканий функций. Рецепт предназначен для отложенного исполнения фрагмента программы и представлен как пара из фрагмента и контекста для его выполнения, называемая замыканием, а процесс так организованных вычислений позволяет переходить к «мемоизации», освобождающей от повторного выполнения функций над одними и теми же данными. Выполнение параллельных процессов часто требует независимости порядка вычислений от последовательности представления действий в программе [45, 46].

2.11. **Трансформационная семантика.** Общие решения по обеспечению лаконизма, универсальности, конструктивности и расширяемости приводят к трансформационной семантике языка программирования, определяющей семейства допустимых преобразований программы, что является естественным полигоном для мета-программирования и практики смешанных вычислений как основы для оптимизации программ, признаваемой уже как основная проблема разработки компиляторов [47]. Эти задачи и задачи управления отладкой и испытаниями программ поддерживает диалект Трансформ.

2.12. **Просачивание операций и функций.** Основные методы лаконичного представления массовых вычислений связаны с использованием неявных циклов, позволяющих избежать выписывания однотипных схем над стандартными структурами данных типа многомерных векторов. Так, например, результат операции над скалярами в языках параллельного программирования обычно автоматически распространен на произвольные однородные структуры данных. Список операций, допускающих такое распространение или просачивание, определен реализацией. Обычно это арифметические операции (+ - * /). Этот механизм в СИНХРОН распространяется на технику применения функций, что повышает лаконизм выражений [1, 36,-38].

2.13. **Фильтрация данных.** Кроме того, из бинарных операций можно конструировать фильтры. Результат фильтрации исчезает из аргумента — он переносится в другую структуру данных, подобно обработке множеств в языке теоретико-множественного программирования SETL [48, 49], что удобно при подготовке программ алгоритмов перебора. При необходимости предварительно сохраняют обрабатываемое значение. Структура из фильтров создает аналогичную структуру из результатов их применения к одному и тому же аргументу.

2.14. **Фрагменты программ и синтаксическое подобие.** При определении функций кроме обычных переменных используются так называемые «фрагментные», используемые для параметризации схем управления и наполнения их подходящим содержанием, возможно требующим контроля вида фрагмента, что полезно для мета-программирования. Подстановка фрагментов, кроме простого сцепления строк подобно макротехнике, использует синтаксические макросы, вид параметров которых задаётся как синтаксическое подобие вхождению переменной в заданную строку согласно синтаксису используемого языка. Для выражения синтаксического подобия введена бинарная операция «~~», задающая вид первого аргумента с помощью строки, являющейся вторым аргументом. Выражение «v ~~ svt» понимается как «переменная “v” имеет вид, заданный её вхождением в строку “svt”». Это означает, что при подстановке значения “v” в “svt” должна получаться правильная конструкция используемого языка. Правильность означает существование строк Pref и Suff, таких, что сцепление строк «Pref», «s», «v», «t» и «Suff» является допустимой программой в используемом языке. Все вхождения фрагментной переменной во фрагмент или схему должны быть синтаксически эквивалентны её вхождению в строку, задающую вид переменной [50]. Работа на уровне фрагментов как уточнение макротехники поддержана диалектом Трансформ.

2.15. **Мета-программирование для преобразования программ.** В случае много-поточных программ возможно преобразование потоков, нацеленное на сведение к однородной системе, однозначно отображаемой на заданный комплекс процессоров размещением потоков по процессам или назначением процессоров для выполнения потоков. Достаточно простые преобразования сети потоков позволяют варьировать схемы потоков и многие

конструкции языка программирования сводить к взаимодействию простых потоков. Обратимость преобразований и чувствительность их результата к информационным связям между фрагментами программы требуют умения формализовать критериев применимости трансформаций и выбора подходящего варианта.

Мета-программирование при решении задач преобразования программ в основном использует макротехнику и символьную обработку, дополненную средствами проверки синтаксической и/или семантической эквивалентности [51]. Для представления сквозных понятий, имеющих различия в диалектах языка, в диалекте Трансформ введено обозначение вида «Понятие.Диалект». При определении перевода программы в иную форму можно конкретизацию отдельного понятия в разных синтаксических позициях представлять в виде «Понятие-Номер», что расширяет возможности использования разных ЯП.

2.16. **Взаимодействие локальной и общей памяти.** Работа с данными при организации параллельных вычислений и предельно распределённых систем из набора потоков, взаимодействующих в терминах доступа к значениям переменных, возможно расположенных в общей памяти, потребовала дополнительных гибридных решений. На уровне ядра языка СИНХРОН предложен механизм взаимодействия локальной и общей памяти отчасти смягчающий проблемы освобождения памяти для много-поточных программ, более подробно описанный в Вестнике НГУ [52]. Обычная система команд виртуальной машины АПК в языке СИНХРОН пополнена для решения проблем работы с общей памятью и внешними устройствами, особенности которых проявляются на командах воздействия на общую память, взаимодействия общей и локальной памяти, доступа к устройствам и организации параллельных процессов [52]. Это команды пересылок данных и обмена данными в общей памяти много-процессорного комплекса, нужные для исключения временных интервалов между взаимосвязанными присваиваниями в общей памяти. Возможен побочный эффект присваивания, допускающий при необходимости восстановление прежних значений переменных. Кроме того, предложена гибридная реализация императивной синхронизации взаимодействия локальной и общей памяти, совмещающая возможности «сборки мусора» и счетчика указателей на память. Для этого используются паспорта данных и пары запрос и его двойник, выполняемые неразрывно в стиле рандеву языка Ada. Технические детали определения этих и остальных команд приведены в статье [52]. Система команд АПК приведена в Приложении 1.

2.17. **Статика и динамика.** Память удобно считать статической конструкцией, в которой лишь изредка, время от времени меняются отдельные элементы. Для программ, допускающих параллелизм, приходится признавать целесообразность учёта динамики протекания процессов, совмещающих свою внутреннюю жизнь с воздействием на на внешний мир. Это исключает предпочтение статической или динамической области видимости имён, требуют программирования их взаимодействия и сосуществования.

2.18. **Побочные эффекты и неизменяемость данных.** Никакое удобство и надёжность чистого функционального программирования не отменяют того

обстоятельства, что существуют задачи, методы решения которых наиболее естественно выражаются как изменения состояний памяти и циклы. Приведение таких решений к чисто функциональной форме, конечно, интересное упражнение для ума, но утрата естественной формы решения может приводить и к снижению продуктивности программирования, и к потере правильности решений. Возможно, транзакционный стиль работы с изменяемой общей памятью является достаточно практичным компромиссом подобно базам данных или введению такого механизма в Clojure, новый диалект языка Lisp.

2.19. **Клетчатое поле.** Большинство учебных задач решаются в обстановке, выглядящей как клетчатое поле, по которому можно перемещать фишки, что способствует пониманию алгоритмов над матрицами и формированию интуитивной грамматики параллельных вычислений. При выборе постановок учебных задач на уровне ознакомления с феноменами параллелизма на базе диалекта Мульти дано предпочтение задачам, допускающим иллюстрацию решений в терминах перемещения фишек по клетчатой доске.

2.19. **Относительное описание языка программирования.** Описание языка представляет собой ряд диалектов, каждый из который определён как дополнение к предыдущему диалекту, возможно поддерживающему другой комплект парадигм программирования. Таким образом осуществлен эксперимент по относительному определению языка программирования, в котором чётко выделена разница с другим языком, и по раздельному представлению парадигм в мультипарадигмальных ЯП. При необходимости разница может быть размещена в суммарном описании очередного диалекта.

3. МНОГО-ПОТОЧНЫЕ ПРОГРАММЫ

«Не верь глазам своим!»

Конкретный свод решений связан с определением класса рассматриваемых много-поточных программ.

3.1. Без иерархии и статической типизации. Язык СИНХРОН приспособлен к демонстрации небольших **много-поточных программ** с целью изучения основных явлений и моделей параллелизма методом показа типичных проблем организации параллельных процессов, заметных при выполнении простых этюдов в форме создания учебных игр, сюжеты для которых можно найти и среди математических задач [53]. Язык не поддерживает статической иерархии определений и областей действия имен. Локализация имен используется лишь при определении исполнителей, потоков и функций.

3.2. **Синхронизация потоков программы.** Много-поточная программа представляется как **набор** потоков и может быть размечена барьерами на **слои**. Каждый поток состоит из очереди вычисляемых, возможно, помеченных **барьерами**, действий. Барьеры нужны для управления синхронизацией потоков, они выполняют роль внутренних точек **синхронизации**, разбивающих программу на слои. Синхронизация потоков заключается в

согласовании времени выполнения помеченных барьерами действий. Определение порядка вычислений отличается от порядка действий, образующих в программе слой. Каждый слой начинает выполняться одновременно и завершается до выполнения следующего слоя. Выражения одного слоя из разных, представленных независимо, потоков с одинаковой пометкой выполняются одновременно, точнее — независимо. Кроме внутренних точек синхронизации можно задавать внешнюю очередь барьеров, определяющую порядок реального выполнения помеченных барьерами слоёв. Внутренние барьеры, не входящие во внешнюю очередь барьеров, на синхронизацию не влияют. [54].

3.3. **События и условия готовности.** При выполнении много-поточной программы известно, из каких потоков она состоит и каков ряд точек синхронизации потоков, разбитых на слои. Достижение барьера рассматривается как **событие**. Объявление события позволяет выполняться ждущим его действиям, обладающим готовностью. Событие не сбрасывается, пока все **ждущие его потоки** не будут готовы или их выполнение не будет оценено как невозможное. Синхросети позволяют независимые описания процессов связывать в терминах разметки барьерами. Узлы с одинаковой синхронизирующей разметкой срабатывают одновременно, если они упомянут во внешней очереди барьеров. Полное представление об асинхронных процессах, их эффективности и проблемах организации дают работы по сетям Петри [55, 56].

3.3. **Варьирование порядка асинхронных действий.** Текст программы формируется как композиции фрагментов. Правило композиции влияет на упорядочение действий при их выполнении, возможно, отличающееся от порядка вхождения в сценарий игры или программу. Программа на языке СИНХРОН строится в определённой обстановке или контексте из действий – выражений или директив, использующих данные из контекста, выполняющего роль общей памяти. Выражения могут использовать размещённые в общей памяти данные, но не изменяют их. Директивы могут изменять хранимые в общей памяти данные [6, 7].

3.4. **Отделение порядка вычислений от порядка выражений.** Параллелизм приводит к независимости порядка выполнения действий и получения их результатов от порядка записи действий в программе и размещения результатов в памяти. Для развития навыков сознательного выбора решений, влияющих на изменение данных в общей памяти, предложено в представлении структур данных учебного языка разнести смысл скобок и разделителей. Скобки означают порядок доступа к элементам структур данных в памяти, а разделители — порядок вычисления элементов при исполнении программы. В результате структуры данных наполняются элементами, порядок вычисления которых может отличаться от порядка вхождения в программу [6, 7]. Примеры синтаксического отделения порядка вычислений от порядка доступа к результатам в структурах данных приведены в таблице 3.

Таблица 3

Порядок вычислений не зависит от порядка доступа к данным.

№	*Выражение*	*Пояснение*
1	'('Выр (**';'** Выр)…')'	Список последовательно вычисляемых элементов, заполняемый в порядке записи в программе.
2	'('Выр (**','** Выр)…')'	Список, заполняемый элементами в порядке записи в программе, вычисляемыми асинхронно, возможно в другой последовательности.
3	'['Выр (**','** Выр)…']'	Вектор, заполняемый элементами в порядке записи в программе, вычисляемыми асинхронно, возможно в другой последовательности.
4	'['Выр(**';'** Выр)…']'	Вектор, заполняемый последовательно вычисляемыми элементами в порядке записи в программе.

Умение учитывать такое различие полезно для понимания оптимизирующих преобразований программ, связанных с разнообразием многопроцессорных комплексов. Различие проявляется в мощности семейства допустимых процессов вычисления, что можно видеть на примерах (табл. 2), демонстрирующих такую разницу на простых выражениях со списками и векторами, воспринимаемыми зрительно как равнозначные, содержащие одинаковые элементы, но семантически они различны.

Таблица 4

Разница в числе допустимых процессов вычисления выражений с разными структурами данных.

№	*Выражение*	*Процессы вычисления*	*Пояснение*
1	(a; b)	{ a' b' ↓b ↓a }	*Один допустимый процесс*
2	(a, b)	{ a' b' ↓b ↓a | b' a' ↓b ↓a | b' ↓b a' ↓a }	*Три варианта допустимых процессов*
3	[a; b]	{ a' b' 2:↓b 1:↓a | a' 1:↓a b' 2:↓b | a' b' 1:↓a 2:↓b }	*Три варианта допустимых процессов*
4	[a, b]	{ a' b' 2:↓b 1:↓a | a' 1:↓a b' 2:↓b | a' b' 1:↓a 2:↓b | b' 2:↓b a' 1:↓a | b' a' 1:↓a 2:↓b | b' a' 2:↓b 1:↓a }	*Шесть вариантов допустимых процессов*

где x' – вычисление выражения x,
↓x – размещение полученного значения x в памяти,
n:↓x – размещение значения x как элемента вектора с номером n,
{ … }– множество допустимых процессов,
| – разделитель элементов множества процессов.

В первом примере согласно порядку записи в программе вычисляется a’, затем b’. После этого в памяти размещается ↓b, получается список (b), затем в этом списке размещается ↓a и получается список (a b). Во втором примере b’ может быть вычислено раньше, чем a’, и его результат может быть сразу или поэже размещён ↓b в списке (b) до вычисления a’ и размещения полученного результата ↓a. Это даст такой же список (a b). Выражения «(a, b)» и «(a; b)» функционально эквивалентны, а мощность пространств допустимых процессов вычислений для них отличается.

В третьем примере размещение 1:↓a первым элементом вектора может произойти как раньше вычисления b’, так и позже, результат которого размещается вторым элементом вектора 2:↓b, а размещение 1:↓a может быть выполнено после размещения 2:↓b. В четвёртом примере допускается возможность вычисления b’ раньше вычисления a’. Результат совпадает с результатом третьего примера, различна лишь мощность пространств допустимых процессов вычисления. Выражения «[a; b]» и «[a, b]» функционально эквивалентны. При ясном понимании теоретической разницы, на лабораторной практике многие студенты учитывают допустимые процессы лишь первого и третьего примеров, глаза воспринимают текст как последовательность независимо от разделителя.

В задачу разработки языка СИНХРОН входит поддержка лабораторных работ для формирования навыков учёта полных семейств допустимых процессов. Здесь важно учесть многоканальность восприятия, включающую не только зрение и слух, но и ручные манипуляции (см. 3.8). Много-поточная программа допускает асинхронное выполнение потоков и действий, что означает в соответствии с принципом **равноправия параметров** функций независимость порядка вычисления выражений от порядка их вхождения в структуры данных, рассматриваемые как функции над этими выражениями.

3.5. **Навыки предвидеть проблемы.** Часть проблем взаимодействия потоков алгоритмически не разрешима, поэтому в задачи ознакомления с параллелизмом входит научиться обнаруживать и предвидеть такие опасности, перестраивать и отлаживать программы при обнаружении неудачных взаимодействий потоков. Наполнение много-поточной программы может развиваться независимо от схем управления вычислениями в отдельных потоках, а схемы можно реорганизовывать без дополнительной отладки наполнения в соответствии с **принципом факторизации** на автономно развиваемые модули, вытекающим из гибкости ФП. Схемы работают подобно макросам, но с контролем соответствия параметров объявленным видам фрагментов. **Д**ирективa при благополучном исходе обработки памяти даёт результат подобно выражению.

3.6. **Условия выполнения действий.** При отладке сценариев можно задавать ориентировочное время срабатывания отдельных **действий**. При выполнении программы могут использоваться внутренние, системные команды, недоступные для внешнего управления. Все команды допускают безусловное и **условное выполнение**. Последнее предназначено для эффективного

реагирование на события и условия. По мере выполнения действия формируются сообщения о **готовности** к выполнению, т. е. началу, о собственно выполнении и о **завершении действия**. Условное выполнение команды приводит к сбросу соответствующего сигнала. Действия, связанные с изменением состояния памяти, подчинены механизму **транзакций**, т. е. признание их безуспешными влечёт **восстановление памяти** в состояние, предшествующее этому действию.

3.7. **Множество исполнителей.** Ведущее понятие языка СИНХРОН — «исполнитель»[3], способный выполнять команды, причем исполнителей может быть много, и они могут обладать разными системами команд [2, 10, 39, 61, 62]. Определение исполнителя в диалекте Мульти выглядит как программа, задающая сценарий его функционирования, запускаемый при вызове исполнителя. Каждый исполнитель представлен как готовая к выполнению программа. Переключение исполнителей можно сделать как смену режимов при подготовке изображений или вёрстки текста. Программы в СИНХРОН строятся из потоков, а потоки — из действий, выполнение которых может быть обусловлено ожиданием времени (пауза) или другого сигнала, любым предикатом или вероятностью срабатывания. Последнее означает, что при исполнении ведется учет частоты выполнения вероятностных действий как в системах для разработки компьютерных игр. Кроме того, можно объявлять планируемую длительность выполнения действий.

3.8. **Ручное оперирования исполнителями.** Постановка учебной задачи для экспериментов с параллельными алгоритмами на диалекте Мульти языка СИНХРОН формулируется в терминах ручного оперирования исполнителями, включая автоматизацию взаимодействующих процессов с помощью мультипрограмм, определяющих их работу как автоматов [39]. Кроме обычных команд из меню системы команд исполнителя и средств синхронизации процессов в терминах барьеров — событий, происходящих в разных процессах одновременно, в диалекте Трансформ языка СИНХРОН есть специальные мета-команды для управления составом исполнителей и их редактирования при определении контекста исполнения программы, нацеленные на формирование навыков мета-программирования. При организации сложных данных и действий используются общие структуры или средства, такие как списки, вектора, варианты и множества, обеспечивающие представление отношений «после» и «одновременно», причём последовательность вычисления компонентов может не зависеть от порядка их размещения в программе или в памяти. Механизмы применения функций и операций рассматриваются как операции, допускающие просачивание.

3.9. **Дисциплины доступа к данным.** Действия в языке СИНХРОН приспособлены к варьированию дисциплины доступа к данным и схемы управления процессами обработки комплексов с помощью схемы обхода, выглядящей подобно оператору IF без ELSE. Потоки могут отлаживаться и выполняться независимо друг от друга в предположении, что каждый из них

---

3 Унаследовано от языка Робик и школьного курса информатики. Используется как общий синоним терминов «автомат», «процессор», «компьютер», «робот», «субъект», «агент» и т.п..

можно располагать отдельно, их могут выполнять разные процессоры..

При подготовке примеров для демонстрации феноменов параллельного программирования были приняты следующее ограничения:

- взаимодействия потоков выполняются только через синхронизацию;
- одновременно исполняемые потоки могут обмениваться данными через общую память;
- при взаимоисключении каждый поток работает в своей копии контекста, скопированного из общей памяти;
- поддерживается список для восстановления многократно выполняемых фрагментов.

4. ОЗНАКОМЛЕНИЕ С ПАРАЛЛЕЛЬНЫМИ ПРОЦЕССАМИ

«В ученье трудно — легко в бою!»

Учебно-ознакомительный диалект Мульти языка СИНХРО использует метафору «фабрика разнородных роботов для конструирования программно-управляемых игрушек», управляемых подобно передвижению фишек на клетчатой доске. По этой метафоре конструктор игрушки заранее строит определённую **обстановку** для комплекса взаимодействующих **роботов**. Например, простая обстановка может быть реализована как клетчатая доска, по которой могут двигаться роботы и на которой можно располагать разные предметы, доступные роботам. Роботы могут обладать разными системами команд и сценариями. Кроме ручного управления роботами возможен их вызов из сценариев. Определение команд роботов выполняется средствами разных диалектов языка СИНХРОН (или средствами других языков) в зависимости от особенностей класса изучаемых задач и цели обучения (Мульти, Трансформ, СинПар, Асинхр, АПК).

4.1. Одна их сложностей вербализации и визуализации интуитивных образов параллельного программирования связана с разнообразием индивидуальных интересов и успешного опыта в обычной жизни. Конструирование игрушек рассматривается как создание системы из ядра и оболочки, поддерживающих оперативную настройку на созвучные, привлекающие, знакомые из жизни или игр, образы [39, 61, 62]. Ядро выбирается как можно более простым, подобно абстрактной машине или базовой семантике языка программирования. Оболочка создаёт расширение ядра для конкретной игры подобно абстрактному синтаксису языка программирования, дополненному, подобно встроенным библиотекам и бенчмаркам, пользовательским интерфейсам с информационным наполнением: картинки, тексты и вспомогательные процедуры по сюжетам, выбранные в соответствии с индивидуальными предпочтениями. На фабрике имеется база данных для информационного наполнения и оно может пополняться в учебном процессе.

4.2. **Ядро системы команд исполнителя.** Любой робот может выполнять небольшой набор общих команд языка СИНХРОН – система команд исполнителя. Команда реализуется как МикроРобот, обрабатывающий

только простые данные. Описания команд обычно представляют как переход из одного состояния МикроРобота в другое, что можно рассматривать как вариант предикатной формы, использующей сопоставление с образцом. Подобно тому как язык Logo часто называют языком управления черепашкой, диалект Мульти можно называть языком низкого уровня для управления раскраской клетчатой доски или перестановкой фишек. Динамика раскраски может выглядеть как движение. Раскраску выполняют исполнители-роботы, передвигающиеся на доске. Роботов может быть несколько, они могут работать по очереди или вместе, создавая иллюзию одновременности. На доске могут быть расположены предметы разных классов. В зависимости от класса роботы могут двигать, забирать с собой или преобразовывать часть предметов.

Команды робота могут быть внешними, доступными через кнопки для ручного оперирования, или специальными, внутренними, используемыми в сценариях, созданные для автоматизации поведения роботов. Каждый робот имеет свои универсальные команды ПУСК, ШАГ и СТОП. Отдельные **категории роботов** могут обладать специальной **системой команд**, включая регистры — это переменные и шкала **сигналов о событиях и условиях**, на которые робот может реагировать по ходу игры. Базовый механизм **взаимодействия роботов** представляется как **объявление и ожидание событий**, показывающих свойства клеток на рабочей доске. Конкретный робот может иметь своё **имя** и один или несколько **сценариев** поведения, задающих спектр допустимых команд. Сценарий робота — это линейный участок с возможностью обхода отдельных действий, допускающий самоприменение. Взаимодействия роботов выполняются исключительно как реагирование на свойства клеток доски. На доске в клетках могут быть размещены разные предметы, обладающие свойствами, которые известны и понятны роботам. Клетки характеризуются классами размещённых на них предметов или исполнителей. При включении каждого робота в обстановку появляются три внешних кнопки управления этим роботом: **пошаговое ручное оперирование,** пуск **автоматически выполняемого сценария** и прекращение работы робота. Конструктор игры или представитель сервисной службы могут настраивать роботов и необходимый для их работы **инвентарь или реквизит** на **стартовое состояние** до начала игры при формировании обстановки. Реквизит или инвентарь отличается от робота отсутствием кнопок управления [63]. Создание роботов и реквизита выполняется диалектом Трансформ.

4.3. Сценарная оболочка. При создании игрушки конструктор может сценарии робота уточнить, включая изменение системы команд. Сценарии представляют собой схему управление поведением робота, включающую в себя действия – команды и выражения. Схему управления и действия одного робота удобно представлять в операторной форме, а при переходе к взаимодействию группы роботов предлагается использовать табличную форму [61-63]. Функциональные и предикатные формы появятся при переходе от ручного оперирования к программированию на других диалектах или

конструированию новых роботов.

Дань приоритету функционального программирования при ознакомлении с параллелизмом можно обеспечить роботами, работающими без изменения реквизита на рабочем поле — принцип неизменяемости данных. Гибкость границ можно демонстрировать на досках разных размеров (масштабируемость). Роботы могут сообщать о неподходящей обстановке (универсальность). Обход доски и оперирование работами поддерживается в любом порядке (равноправие параметров). Робот может вызывать сам себя (само-променимость). Формируется сигнал об успешном завершении сценария (строгий результат). Так может работать сторож, подсчитывающий хранимые драгоценности, хозяйка, проверяющая на месте ли вещи после детских игр, робот Karel, включающий-выключающий уличные фонари, сборщик фруктов, вычисляющий сколько надо припасти корзин, маляр, планирующий количество краски для покраски забора, и т. п.

4.4. **Конструирование новых исполнителей.** В рамках диалекта Трансформ можно фиксировать изменение системы команд робота, реализуемое фабрикой как запуск производства нового типа роботов. Обстановка выполнения учебно-игровых программ может быть устроена как МакроРобот, который схемоподобен роботу, только роль команд выполняют вовлечённые в игру роботы, а роль данных — **фрагменты** сценариев или протоколов игр, допускающие редактирование, и инвентарь, имеющий фиксированное число состояний памяти, переключение между которыми подчинено определённой **дисциплине** [61-63].

Работа диалекта Трансформ похожа на работу препроцессора в производственных системах программирования и может быть представлена в предикатной форме как сопоставление с образцом. Работа команд любого робота может быть достаточно точно определена в стиле абстрактной или виртуальной машины. Можно использовать предикатную форму, использующую сопоставление с образцом как систему переходов над четырьмя простыми регистрами, подобно абстрактной машине:

П С У Д ⟶ П' С' У' Д', используя следующие обозначения:
П — протокол или результат работы команды,
С — обозначение выполняемой команды,
У — управляющий сигнал, разрешающий выполнять команду,
Д — данное, используемое при выполнении команды.

Общая схема команд робота (_ - произвольная команда) имеет вид:

_ : (_ _ 1 **знач**) → (**знач** _ 0 **знач**)

\\ *команды работают на общих регистрах*

Робот.П += В_К += ком(Д)

Специальные команды и операции Трансформ позволяют воздействовать на всех включенных роботов и редактировать фрагменты обстановки и программы, за исключением своих команд, своего сценария, а также команд и сценария МикроРоботов и ядра абстрактной машины.

Простейшие действия — это выражения и команды. Более сложные действия строятся как композиции, использующие фрагменты и операции редактирования, выполняемые при формировании обстановки техникой мета-программирования, возможно использующей ещё один регистр Б, содержащий базу данных для хранения рисунков, текстов и других фрагментов, многократно используемых при конструировании роботов и их функционировании: П С У Д Б ⟶ П' С' У' Д' Б'.

4.5. **Взаимодействия и спецификация.** Каждый робот выполняет свою работу в отдельном потоке, на своём процессоре. Возможны связи по времени между роботами и фрагментами программы, они задаются специальной операцией указания роботу выполнить данный фрагмент, что допускает участие нескольких роботов при выполнении одного потока много-поточной программы. Роботы могут различаться по системе команд и другим характеристикам. В таком случае необходимо, чтобы представление потока соответствовало системе команд робота и комплекту используемого инвентаря, точнее сценарий был выполним заданным роботом. Спецификация робота должна хранить нужные характеристики, в частности, относительное время выполнения действий.

4.6. **История выполнения.** Каждый робот имеет встроенный самописец, регистрирующий ход процесса выполнения его части много-поточной программы в **протоколе.** Роботы генерируют протокол, внося в него сообщения о статусе выполняемых команд. Безусловные команды вносят в протокол события Н В К (начало выполнение конец), условные команды – Н В К или Н К в зависимости от истинности **ключевого сигнала.** МакроРобот после редактирования сценариев и формирования обстановки вносит в протокол имена включаемых роботов, объявляемые ими события и готовность к **реагированию на события**. Кроме того, попутно собирается статистика **частоты выбора вероятностных ветвей**. Выраженные как дроби в текстах сценариев вероятности приводятся к общему знаменателю.

4.7. **Оперирование на уровне виртуальной машины.** Для оперирования роботами выделяются команды уровня МикроРобота, устроенные как функции без параметров над простыми данными из общей памяти — виртуальная машина, описываемая в предикатной форме, использующей сопоставление с образцом.

Каждого робота сопровождает своя легенда, поясняющая в какой игровой **обстановке** робот действует и каков смысл его системы команд для игры. Кроме того, известно его **изображение** и вид кнопок управления роботом. Робот может обладать встроенным **сценарием** – программой автоматизации его поведения. Игра управляется много-поточной программой, выполняемой комплексом роботов, включая МакроРобота, созданного с помощью диалекта Трансформ.

4.8. **Пример робота «Кнопка».** В качестве примера рассмотрим механизм работы кнопки для команды «ком», изображаемой на одноклеточном поле как буква «К».

Кнопка К: П С У Д → П' С' У' Д'

Нажатие кнопки К приводит к появлении 1 в регистре К.У, что даёт команде «ком» из регистра С право выполнения. Не важно каким был до этого протокол, теперь в него записывается «ком(Д)»[4] — команда над данным Д, в регистр управления заносится 0 как сигнал о выполнении команды, данное сохраняется до завершения команды.

_ ком 1 Д → ком(Д) ком 0 Д

// в протокол записалось «ком(Д)» и сбросился сигнал исполнения нажатия
// робот до выполнения команды запишет в протокол Н_К.

Робот.П += В_К += ком(Д)

// робот после выполнения команды запишет в протокол — З_К.

Робот.П += «З_К»

4.9. **Пример робота «Переменная»**. Аналогично можно описать механизм обычной переменной как робота, который на двух-клеточном поле объявляет, инициирует, читает и обновляет её значение. Команды (**→, ←, ↑, !, ↓, :=**) работают на общих регистрах МикроРобота.

Переменная V: П С У Д → П' С' У' Д'
!: (_ → 1 имя) **→ (**П' → 0 имя) *\\ объявление робота*
Р.Д += имя *// в одной клетке поля появляется имя робота*

Робот.П += В_К += ком(Д) *// событие — команда выполняется*
\\ робот до команды записал Н_К, после запишет — З_К.

←: (_ ← 1 имя) **→** (П' ← 0 имя) \\ удаление робота
Р.Д -= имя // из поля удаляется имя робота
Робот.П += В_К += ком(Д)
// в протокол внесена команда удаления робота
\\ робот до команды пишет Н_К, после — З_К.

ком : (_ Робот.С.ком 1 Робот.Д.имя)
// своя работа с переменными у каждого робота
// передача значения от другого робота - инициирование
→ (В_К **_** 0 **Робот.Д.ком**) *// результат идёт в данные*

↑: (_ ! 1 имя) → (Р.Д.имя ! У0 имя) *\\ вызов значения робота ()*
Робот.П += В_К += ком(Д)
\\ робот до команды пишет Н_К, после — З_К.
\\ значение располагается на внешнем поле, например, в окне вывола
\\ перепись значения в протокол команды присваивания

4 Без пробела, чтобы это был неразделимый текст.

ком : (_ ком 1 **знач**) → (**знач :=** 0 **знач**)
\\ команды работают на общих регистрах

↓ **:= : (знач** С У имя) → (П' С' У' Д') *\\ обновление значения робота*
Робот.Д.имя = знач
Робот.П += В_К += ком(Д)
\\ робот до команды пишет Н_К, после — З_К.

Реквизит исполнителя или инвентарь робота приспособлен к установке стартовых сигналов МакроРоботом, инициированию данных и их изменению роботами, входящими в спецификацию инвентаря.

5. УЧЕБНО_ПРОИЗВОДСТВЕННЫЕ ДИАЛЕКТЫ

После ознакомления с проблемами параллелизма возникает задача приобретения навыков отладки программ на языках высокого уровня, начиная с использования парадигмы функционального программирования. Для этих целей на разных уровнях работают диалекты Трансформ, Аспар, Синпар и АПК.

Во всех диалектах любая конструкция программы может быть именована, если предстоит её применять многократно. Именованная конструкция может использоваться и как безымянная. Программа формируется как взаимодействие схемы управления с наполняющими её действиями над памятью, не меняющими память выражениями и средствами формирования структур данных. При формировании схемы управления вычислениями вводятся определения данных, функций и схем фрагментов, используемые в иерархии блоков, функционирующих над заранее заданным контекстом. Отдельно выделена синтаксическая позиция выбора планового результата системы потоков.

Формат комментария на несколько строк не имеет принципиального значения. Комментарий хранится в коде программы, на случай использования при отладке, его можно размещать в программе и/или его адрес в её коде. Он не исчезает после компиляции.

Программы выполняется в определённом **контексте** и представляют собой конструкцию из **блоков**, завершаемую обработкой полученных **результатов**, необходимость которой обусловлена много-поточностью. Возможно явное объявление результатов, иначе результатом является последовательность результатов потоков в порядке их вхождения в программу.

5.1. Программное конструирование учебных игр (Трансформ)

«Мастер — ломастер»
«Я учиться не хочу.
Сам любого научу»
Самуил Маршак
Вольт Суслов Стихотворение «Ну и мастер»
Велика у стула ножка -
подпилю её немножко.
Велика теперь другая,
подпилю с другого края.
А теперь вот эта ножка...
Эх ошибся я немножко...

Программная поддержка учебных роботов на уровне диалекта Мульти устроена как система из ядра и оболочки [61-63] [5], подобно языкам управления заданиями, в которых обычно доступны базовые средства и возможность

5 Подразделы 5.1 — 5.9 составлены по следам дипломных работ студентов С. Демидов и Н. Шаченко.

создавать и улучшать свои сценарии, что входит в задачи диалекта Трансформ, являющегося языком сверх высокого уровня (ЯСВУ), поддерживающего преобразования программ на разных диалектах. В рамках диалекта Трансформ программа рассматривается как чередование трансформируемых и неизменных участков, представляющих контекст вычислений или набор потоков, вид которых зависит от диалекта.

5.1.1. Трансформ для ядра диалекта Мульти позволяет:

- Создавать свободное клеточное поле размером NxM, где N и M задаются пользователем.

- Размещать в произвольные клетки поля элементы (неподвижная “стена”, подвижный “ящик” и стандартный робот-исполнитель).

- Задавать каждому исполнителю предопределенную последовательность команд, которую можно изменять в процессе игры.

5.1.2. Для оболочки, соответствующей нажатию кнопок или выполнению программ взаимодействия процессов, Трансформ допускает редактирование протоколов хода игры или выполнения программы. Оболочка является буфером между вводом команд и работой ядра, облегчающим процесс управления игровыми объектами и позволяющим в понятной человеку форме визуализировать происходящее. Имеется возможность запоминания истории действий и произвольно «передвигаться» внутри истории действий, произошедших на клеточном поле.

5.1.3. На уровне Мульти можно выделять специальные виды объектов, например, «Ящик» - это объект, который занимает одну клетку и не имеет своих активных действий, но его можно двигать, а объект «Стена» неподвижен. Задаются не только размер поля, специальные объекты и их координаты, роботы-исполнители, их команды, но и условия выполнения команд. Каждый уровень имеет свои условия успешного прохождения в соответствии с постановкой задачи, например, успешного собирания роботами всех объектов определённого типа.

5.1.4. Самой важной функцией является возможность создания новых действий для роботов на основе более простых (атомарных) с возможностью последующего использования, включая совместные действия.

5.1.5. Диалект **Трансформ** является языком сверх высокого уровня, обогащённым средствами мета-программирования, расширяющими возможности макрогенерации. При представлении программ для этого диалекта используется понятие «составное имя» в двух случаях. При определении преобразования могут быть нужны определения из других диалектов и конкретизации понятий, отличающихся в разных синтаксических позициях отдельного правила. Имена сквозных понятий представляются в виде «Имя.Диалект», а конкретизации понятия в виде «Имя-Номер», где «Номер» означает позицию вхождения понятия в правило. Такими обозначениями можно описывать результаты преобразования программ для отладки, управления ходом исполнения, выполнения частичных вычислений, выделения фрагментов, производства протоколов, сравнения программ, выяснения расхождений в версиях программ, их оптимизации и другое, включая определение границ

эквивалентности преобразуемых программ, **измерения** характеристик с помощью встраиваемых измеряющих функций, сравнения производительности улучшаемых программ.

Трансформации участков сводятся к применению макросов, замен фрагментов программы на другие, эквивалентным преобразованием или переводом фрагмента на другой диалект. Определение макроса выглядит подобно определению функции с объявлением видов параметров в расчёте на выполнение символьной подстановки в тело определения. Предполагается, что результат подстановки является синтаксически правильным текстом. Остальные трансформации используют сопоставление с образцом. Такие определения содержат два образца. Первый выполняет роль усложнённого списка параметров, получающих значение при сопоставлении фрагмента программы с образцом. Второй используется при формировании результата, размещаемого в программе вместо сопоставленного фрагмента. Для преобразований и переводов требуется установление функциональной эквивалентности, что выходит за пределы содержания данного препринта. Перевод в качестве результата вырабатывает программу на другом диалекте. Образцы строятся из слогов, определение которых приспособлено к автоматизации конструирования обработчиков программ. Диалект наследует идеи БНФ, YACC, LEX, mpC, Clang-LLVM.

Трансформ позволяет для диалекта Мульти:

- Задавать и изменять размеры поля.
- Изменять количество роботов и объектов.
- Изменять виды роботов доступных на данном уровне.
- Создавать собственных роботов и задавать их возможные команды.
- Задавать собственные команды и добавлять их в стандартное меню команд.
- Сохранять созданный уровень в списке стандартных уровней.
- Определять и добавлять условие завершения уровня (успех или провал).

5.1.6. **Ядро игр на клетчатой доске.** Простейшая система команд для роботов-исполнителей на клетчатой доске содержит в диалекте Мульти базовые действия, такие как:

1) ВВЕРХ - Передвинуться вверх
2) ВЛЕВО - Передвинуться влево
3) ВПРАВО - Передвинуться вправо
4) ВНИЗ - Передвинуться вниз
5) ПАС - Пропуск хода

Диалект Мульти позволяет опробовать на задачах движения по доске — пройти по диагонали, обойти периметр, дойти до заданной клетки, подсчитать число предметов на доске и т. п. Прорисовку предметов и персонажей можно

изменять на уровне Трансформ. Часто изображения персонажей размещают в одну клетку, но можно делать и крупные изображения персонажей, реализуя их прорисовку как одновременное изменение вида комплекта жёстко связанных клеток. Меню можно сделать бестекстовым, в виде общепонятных знаков на отдельных кнопках. При желании можно фон игры сделать достаточно зрелищным и динамическим, а командам дать текстовые варианты в соответствии со сценарием игры.

5.1.7. **Пример игры «Философы».** Например, можно сделать игру «Философы», на которой в теории параллелизма демонстрируют разные тонкие эффекты организации взаимодействующих процессов. Пример обедающих философов является одной из классических иллюстраций проблем взаимодействия параллельных процессов. Он примечателен простотой и наглядностью концепции и наличием большого числа возможных тонких решений, изученных на уровне теории организации процессов. Задание поведения философов через встроенный текстовый редактор позволяет вручную настраивать количество философов, а также последовательность действий из следующего набора команд:

– ШАГ_Е- Сдвиг на одну клетку в направлении стола.
– СВБ - Нужная клетка свободна.
– СЛВ - Если левая вилка свободна - взять её.
– СПВ - Если правая вилка свободна - взять её.
– ЕДА - Приступить к еде.
– ВЛВ - Вернуть левую вилку.
– ВПВ - Вернуть правую вилку.
– ШАГ_Д- Сдвиг на одну клетку в направлении двери.

```
(ЦИКЛ ШАГ_Е  [ ПОКА СВБ ] ПОВТОР ШАГ_Е)
                    // Около стола движение прекращается.
СЛВ СПВ ЕДА
ПАУЗА 10 // каждый ест по 10 минут
ВЛВ ВПВ
(ЦИКЛ ШАГ_Д  [ ПОКА СВБ ] ПОВТОР ШАГ_Д)
```

Пример 1. Сценарий на случай, что философы у своих дверей на границе доски, а стол со спагетти в центре доски. До стола не менее одной клетки.

При выборе действия исполнитель пытается его осуществить в течение неограниченного количества времени, поэтому иллюстрация дедлока получается наглядной. Оператор-игрок имеет возможность отмены хода, в результате поле возвращается к предыдущему состоянию, а сценарий игры можно подкорректировать средствами Трансформ.

5.1.8. **Демонстрируемые проблемы.** Подобным образом удаётся демонстрировать такую проблему параллелизма как состояние гонки (race condition): Это гонка между двумя роботами за обладание соседней клеткой. Два робота разделённые ею, пытаются захватить её. Так как объекты исполняют свои действия в случайном порядке, то нельзя предугадать, какой из них займёт

клетку. В результате, кто-то достигнет своего, а кто-то нет. Возможны варианты взаимной блокировки (deadlock), реализованной в виде двух множеств роботов, каждое из которых хочет занять соседние клетки, занятые другим множеством. В таком примере клетки рассматриваются как ресурсы. Эксперимент с одним философом заодно показывает, что истинный параллелизм начинается с двух философов [61, 62].

5.1.9. Когнитивные ошибки. Одной из проблем при создании игр детьми является неточность представления и понимания сложных логических и вероятностных условий управления действиями персонажей. Эта проблема обусловлена противоречием между интуитивной и формальной оценкой истинности логических и вероятностных формул. Интуитивно истинность связки «&&» оценивается выше, чем истинность составляющих, а реально она ниже. Противоречие, имеющее лингвистический характер, замечено в исследованиях нобелевского лауреата Д. Канемана [57].

Формат и примеры встроенных команд «Философов» в предикатной форме имеет вид:

<Условие → Изменение_разметки_ игрового_поля>

На языке программирования это можно выразить в форме условной команды:

движение_от_Двери_к_Столу (int i, j) =
{ (i < N/2 && д [i][j]==2 && д [i+1][j]==0)
→ (д [i+1][j]=2 ; д [i][j]=0) ,
// Для Ф2 если нужная клетка свободна, то Ф2 идёт вверх

(i > N/2 && д [i][j]==2 && д [i-1][j]==0)
→ (д [i-1][j]=2 ; д [i][j]=0)
// Для Ф1 если нужная клетка свободна, то Ф1 идёт вниз
}

Несколько понятнее можно написать более традиционное условное выражение:

если (i < N/2 && д [i][j]==2 && д [i+1][j]==0) то (д [i+1][j]=2 ; д [i][j]=0)

Возможно, ещё понятнее или надёжнее будет техника таблиц решений, на которую в начале 1970-х обратил внимание Дж. Шварц при создании языка теоретико-множесвенного программирования SETL. По аналогии с ней можно представлять таблицы решений, в которых левая колонка содержит простые выражения, участвующие в выборе решений, а в остальных столбцах располагаются значения этих выражений, соответствующие каждому выбору. Есть наблюдение, что в табличной форме когнитивных ошибок встречается меньше.

Таблица 5. Пример таблицы решений

если	*Ф2: ему нужна клетка сверху, если она свободна, то Ф2 идёт вверх*	*Ф1: ему нужна клетка снизу, если она свободна, то Ф1 идёт вниз*
i < (N/2-1)	истина	–
i > (N/2+1)	–	истина
д [i][j]	**2**	**2**
д [i+1][j]	0	–
д [i-1][j]	–	0
то	(д [i+1][j]=2 ; д [i][j]=0)	(д [i-1][j]=2 ; д [i][j]=0)

Линеаризация таблицы решений для размещения в тексте программы может иметь вид текста, выравнивание колонок в котором достигается табуляцией, а записи выделяются концами строк, что позволяет текстовому редактору конвертировать текст в таблтцу.

```
РЕШЕНИЕ         Ф2          Ф1
(i < (N/2-1))   истина      _
(i > (N/2+1))   НЕТ         ДА
(д [i][j])       2          2
(д [ i+1 ][ j ])  0         _
(д [ i-1 ] [ j])  _         0

ф1 =  ( д [ i-1 ][ j ]=2 ; д [i][j]=0 )
ф2 =  ( д [ i+1][ j ]=2 ; д [i][j]=0 )
```

Такой формат выглядит более свободным от ложных лингвистических ассоциаций, реже провоцирующим на ошибки.

5.2. Асинхронные много-поточные программы

Диалект Асинхр, язык высокого уровня (ЯВУ), поддерживает парадигму функционального программирования, характеризуемую принципами универсальности функций, равноправия параметров, само-применимостью, гибкостью ограничений, неизменяемости данных и строгостью результата. На основе таких принципов асинхронные программы на уровне диалекта Асинхр, допускают представление выражений, значение которых не всегда определено или вычислимо в текущий момент, но при отладке можно рассчитывать на будущее значение в продолжающем процессе или восстановление предыдущего значения при поддержке восстановления памяти для шагов рекурсии-цикла. Возможен выбор между статикой и динамикой при взаимодействии областей видимости имён, допускать явное указание, так было в языке Рапира. Кое-что может дать представление недоопределённостей в тексте программы или диагностическое расширение грамматики с переводом и специальный язык доопределения шаблона программы.

При организации структур данных можно использовать триплеты с

пустыми границами, понимаемыми как бесконечные списки. При формировании структур используются перечисления и предикаты. Для организации контроля типов данных в представлении аргумента и результата можно использовать примеры произвольных значений заданного типа. Можно использовать таблицы вычисленных результатов и библиотечные функции. Для равноправия потоков нужна свёртка их результатов с помощью мульти-функций, оперирующих произвольным или заданным числом параметров. Мульти-функция введена для свёртки заданного числа результатов в одно данное (список, вектор, сумма, произведение, минимум, максимум, среднее, последний и др.). При компиляции может быть установлено число аргументов мульти-функции.

В дополнение к чисто функциональному программированию, допускается чтение из внешней памяти. Доступны протоколы других прогонов программы для парного или сравнительного исполнения программ. Поддержано опережающее выполнение функций для профилактики порожнего функционирования отдельных процессоров. Возможно внешнее решение проблемы остановки из независимых процессов, прерывание или освобождение процессора, возобновление прерванных или отложенных вычислений. Возможен учёт границ вычисленности функций и объявление величины допустимого времени счёта или числа шагов цикла. При вычислении формируется статус функции: ОК, новое, изменённое, макет, спецификация. Может быть задано число процессоров или осуществлён запрос нового процессора. Выполняется разделение активных и фоновых вычислений (про запас), замеряется производительность многократно используемых программ, собирается статистика по сеансам отладки. Ранние версии программы могут быть прототипами для переход к более производительной версии.

Средства диалекта Аспар с Трансформ достаточны для выполнения декомпозиции программы на систему функций, приведение функций к чисто функциональной форме для отладки и улучшения. Можно представлять неограниченный параллелизм и пошаговый переход к производительной форме с общей и внешней памяти, включая учёт фактического числа процессоров. Для оптимизации может быть полезен анализ перемещаемости участков или функций, систематизация используемых элементов: чистые, общая память, устройства и сеть. Средства управления вычислениями образуют ряд: комплект автономных процессов, взаимоисключение, обход отдельных участков, рекурсия или цикл для многократно повторяемых участков, вызов многократно используемых фрагментов, а также схем, нагружаемых разными фрагментами.

5.3. Много-поточные программы над общей памятью.

Диалект Синпар (**син**хронизованный **пар**аллелизм), язык сверх высокого уровня (ЯСВУ), поддерживает более полную парадигму параллельного программирования, наследует идеи языков Lisp 1.5, Pure Lisp, Setl, БАРС, Sisal, Clisp, mpC и других, менее известных, языков программирования [1], ориентированных на реализационную прагматику символьных вычислений,

абстрактных машин, управления операционными системами, базами данных, редактированием и визуализацией, используемыми при подготовке и отладке программ. Он создан для поддержки экспериментов с много-поточными программами над общей памятью. Их разработка затруднена сложностью и многообразием моделей, понимание которых необходимо при организации параллельных вычислений на современной аппаратуре. Реализационная прагматика диалекта Синпар ориентирована на механизмы символьных вычислений, абстрактных машин, управления операционными системами и базами данных, редактирования и визуализации, используемыми при подготовке и отладке программ.

Более конкретно, от языка Lisp 1.5 наследуется выделение семантического базиса (Pure Lisp), множественность определений функций с хранением вариантов символьных и кодовых форм, универсальность и разнообразие категорий функций, включая специальные функции с нетипичными моделями вычислений, мульти-функции над произвольным числом аргументов, мета-программирование, равноправие средств ввода-вывода с функциями языка, сохранение внешних форм и свойств элементов программы на период её исполнения, управляемость средств отладки программ, двойственность системной поддержки (интерпретатор и компилятор), разрешение при экспериментах изменять семантику практически любых конструкций языка (кроме атома Nil), что позволяет моделировать разные парадигмы программирования и их расширенные варианты, приспособленность реализации языка к методике раскрутки, позволяющей реализацию начинать с уровня ядра, а затем её пошаговым образом расширять. Наследование таких особенностей языка Pure Lisp позволяет выделить в проекте языка Синпар компактное высокоуровневое ядро системы программирования. Лаконизм необходим для ознакомления с ключевыми идеями до перехода к практике.

Язык теоретико-множественного программирования Setl поддерживает, кроме обычной процедурной техники определения функций, возможность использовать множества кортежей для хранения соответствия аргументов и результатов функции, что открывает перспективу мемоизации как средства оптимизации многократно используемых библиотечных модулей, особенно при реализации алгоритмов с высокой сложностью вычислений. Обработка структур данных допускает пересылку элементов с исчезновением их из исходной структуры. Используется вычисляемая левая часть присваиваний и множественность реализации структур данных, поддерживающая автоматическую реорганизацию ради максимально возможного продолжения вычислений и повышения производительности. Наследование таких возможностей позволяет загружать простаивающие процессоры опережающими вычислениями и уменьшить нагрузку на механизм контроля типов данных.

При разработке языка БАРС (Базовый язык проекта МАРС), предпринятой в начале 1980-ых годов, была провозглашена концепция выделения подъязыков вычислений, обработки памяти, сетевого управления асинхронными процессами и комплексации сложных данных. Подъязык

вычислений использует механизм просачивания скалярных операций, унаследованный от языков APL, АЛЬФА, VAL, на структуры данных. Наследование такой структуры определения языка позволяет избегать сложности одновременного освоения разнородных моделей. Предложенный в этом языке механизм синхросетей позволяет представлять оперативные решения проблемы синхронизации на уровне оболочки системы программирования, а уровень ядра ограничить моделями синхронизованных потоков. Обработка памяти включает программируемый механизм дисциплины доступа, что полезно при программировании работы с неоднородной многоуровневой памятью. Средства комплексации позволяют представлять кратность вхождения в структуры данных элементов, включая процессоры.

Появившийся в конце 1980-ых функциональный язык параллельного программирования Sisal даёт удобное решение по организации распараллеливаемых циклов. Предложено выделение синтаксических позиций для представления пространства итерирования и отдельно для свёртки результатов выполнения всех итераций цикла. Чётко выделена роль однократных присваиваний и границ наследования значений переменных. Выражения могут вырабатывать более одного результата. Имеется специальный подъязык управления вводом-выводом данных. Каждый тип данных содержит своё значение ERROR. Наследование таких особенностей даёт перспективу расширения методов распараллеливания и распространения их на более широкий класс устройств.

Язык Clisp, наиболее практичный язык функционального программирования, теперь обладает особо эффективной реализацией CMUCL. В этом языке поддержан гладкий переход от академических средств чисто функционального программирования к производственным средствам других парадигм с помощью включения в систему программирования эквивалентных функций, обладающих другой реализационной прагматикой, включая организацию автономных процессов.

Дань популярности языкам семейства «C» представляет язык параллельных вычислений mpC (multi-programming C). В этом языке используется явное управление процессорами с помощью сетей и синхронизации в терминах барьеров. Имеется разнообразие видов памяти — размазанные переменные и их реплики, что дополняет привычные типы данных возможностью программировать дисциплину доступа к ним. Предложен механизм измерения производительности процессоров с автоматизацией перераспределения потоков по процессорам. Наследование таких решений на уровне ядра языка и его абстрактной машины позволяет формировать сетевые много-поточные программы над неоднородными процессорными комплексами.

Диалект Синпар, поддерживая самую популярную парадигму императивного программирования работы с общей памятью, несколько ослабляет её в связи со спецификой функционального программирования и много-поточности средствами восстановления данных. Очередное действие потока начинается строго после начала предыдущего, но может не дожидаться его завершения. Между двумя соседними действиями потока может быть

выполнено действие другого потока. Такое смягчение компенсируется возможностью объявлять непрерывные критические интервалы как в языке Algol-68. Кроме того, в механизме комплексации сложных данных предложена прагматика разделения процесса выделения памяти от процесса вычисления размещаемых в памяти данных. Можно сказать, что программа строится как императивная схема, допускающая взаимодействие процессов в терминах сообщений и общей памятью, на этапе отладки наполняется функциональными фрагментами вычислений без действий над общей памятью, от которых по мере отладки возможен гладкий переход через замену чистых функций на процедурные аналоги с воздействиями на память и синхронизацией потоков с помощью барьеров по мере необходимости.

В Приложении 1 представлена система команд абстрактного многопроцессорного комплекса (АПК), в Приложении 2 приведена общая сводка синтаксиса языка, занимающая примерно одну страницу. Это несколько больше по объёму, чем определение синтаксиса Pure Lisp, но всё-таки это не 19 страниц для языка C или 32 страницы для стандарта C++.

Некоторые понятия следует несколько уточнить.

Контекст программы — бывает локальный контекст потока, защищённый от доступа из других потоков, и глобальный, реализуемый как общая память, используемая при взаимодействии процессов, данные из которой доступны всем потокам программы, при компиляции отображаемым в процессы. Контекст программы содержит описания глобальных, возможно изменяемых, общих данных, доступных директивам потока. Существуют и локальные контексты в определениях функций и циклов, доступные только внутри определения. Внешне обработка общей и локальной памяти выглядят одинаково, что позволяет перемещать фрагменты в разные позиции. Компилятор их различает по вхождению имени в тот или иной контекст. Имена идут в память с данным или ссылкой на результат компиляции определений. Возможно хранение непосредственно определений и имён. Типы именуемых данных можно устанавливать по виду инициирующего значения или функции. Имеются распознающие предикаты ATOM, NUMBER, LIST, ARRAY и др.

Программа строится как иерархия блоков, возможно обладающих своим локальным контекстом и содержащих поток или комплект потоков или процедуру с определённым методом свёртки результатов. Процедура — это вызов нерекурсивной функции. Элементы комплекта функционируют автономно и можно комплект пометить барьером на случай необходимости в синхронизации. Поток устроен как ряд директив, выполняемых последовательно.

Комплект при компиляции преобразуется в элемент программы управления комплексом процессов, каждый из которых выполняется одним отдельным процессором. **Комплект** состоит из непустого конечного числа потоков, выполняемых асинхронно без взаимодействий по управлению за исключением явных барьеров-синхронизаторов и обмена сообщениями. Возможно явное объявление **результатов,** иначе результатом является последовательность результатов потоков в порядке их вхождения в программу.

Поток представляет собой непустой ряд или очередь директив-команд, возможно синхронизованных по одноименным барьерам с действиями из других потоков. Должен существовать **контекст**, в котором выполнимы все действия потока, допускающий наращивание при исполнении.

Очередная директива может начинать выполнение строго после начала **«;»** предыдущей директивы, но не обязана дожидаться её завершения**,** при необходимости ожидания используется разделитель **«;;»**. Результат директивы размещается в стеке строго после того, как разместился результат предшественника. Возможно явное объявление и реорганизация результатов, включая, что результатом является последовательность результатов потоков в порядке завершения их вычисления.

Схема — это макрос, при подстановке параметров которого формируется блок, включаемый в текущий комплект потоков или в общий контекст программы. По умолчанию схема выглядит как функция, но её формальными параметрами являются фрагменты - значения специальных фрагментных переменных, требующие до подстановки проверки фактических параметров на корректность синтаксиса. Можно проверять синтаксис результата подстановки фрагментных параметров в точке вызова специальной функцией COMSUB. При вызове фрагмента возможна его компиляция «на лету», результат которой размещается в общей памяти, а применяется в зависимости от очередных команд. Предполагается, что таким образом можно разделять представление схем управления и фрагментов вычислений, ставить эксперименты по расширению пространства используемых моделей параллельных вычислений, включая разделение порядка вычислений и последовательности воздействий на память. Схема работает как макроконструкция, рассматриваемая как часть программы, наполняемая фрагментами вычисления при определении взаимосвязей между элементами директивы.

Директива — конструкция управления действиями, подчинёнными правилу однократного присваивания. Может иметь вид любой **схемы**: ряд, строй, обход, вызов функции, цикл и даже динамически формируемая фрагментная схема, возможно с явным указанием процессора. Лишь бы все переменные получали значение только один раз. Возможно изменение локальной памяти, не влияющее на общую память. Различаются «мягкая» последовательность независимых действий с однократными присваиваниями или «строгая» последовательность рядов действий с однократными присваиваниями.

Цикл реализуется как вызов безымянной рекурсивной функции, выход из которой может быть задан разными способами, включая задание числа повторений.

Определение функции может быть задано как блоком, так и структурой данных, поэтому при **вызове функции** можно указать нужный формат определения – категория определения. Механизм кодирования вызова функции учитывает наличие-отсутствие рекурсии в определении.

Действия и выражения, входящие в директивы, различаются по отношению к работе с памятью. Выражения могут использовать размещённые в

общей памяти данные, но не изменяют их. Действия могут изменять хранимые в общей памяти данные. Директива вырабатывает результат обработки памяти при благополучном вычислении определённого выражения, имеет результат подобно выражениям. Определён нерасширяемый ряд конкретных вариантов действий над общей памятью. **Фильтр** — предикат для выбора пересылаемых данных (редукция). **Команда** — добавленное действие, заранее не предусмотренное в ядре ЯП, возникшее в конкретном приложении. Должна быть известна арность результата, подобно арности аргумента. **Выражение** представляет организацию вычислений с помощью операций по определённой формуле над заданными данными, представляющими значения, соответствующие смыслу операций. Для каждой операции известна её арность и число результатов. Новые операции можно программировать как функции.

Данные и структуры данных являются не более чем представлениями значений. Значение может быть представлено разными данными в зависимости от форматов, требуемых для выполнения операций. Данное представляет первичные, простые значения или составные, сложные, конструируемые из элементов так, что можно из составного данного извлекать его составляющие.

Процессор — разновидность данных. Всегда, по умолчанию, имеется один процессор, на котором происходит выполнение основной программы.

Структуры наполняются элементами, порядок вычисления которых может отличаться от порядка вхождений в программу. Структуры данных поддерживают разделение порядка вычислений и последовательности размещения полученных результатов в памяти. Это может пригодиться при организации преобразований программ, связанных с разнообразием многопроцессорных комплексов.

5.4. Абстрактный многопроцессорный комплекс.

Виртуальная машина АПК является диалектом языка СИНХРОН в качестве языка низкого уровня (ЯНУ), поддерживающего многопроцессорное программирование. Представления данных используют понятие комплекс, позволяющее задавать кратность элементам, сформированным из одноимённых идентичных составляющих, и в дальнейшем позволять развиваться им независимо. АПК выполняет роль абстрактной машины для определения механизмов выполнения много-поточных программ на многопроцессорных комплексах, обладающих сходной системой команд. При функционировании комплекса число процессоров может изменяться. Кроме собственно процессоров к выполнению программы можно привлекать дополнительные устройства, которые рассматриваются как специальные процессоры без заранее определённой системы команд, способные выполнять некоторые действия по командам процессора. Система команд АПК поддерживает обработку данных, их размещение и реорганизацию в общей памяти программы или в локальной памяти процесса, управление ходом выполнения процессов, включая взаимодействие процессоров и устройств, и резервирование данных для их защиты от случайных изменений (см. Приложение 1).

Программа выполняется над определённом контекстом, данные из

которого доступны всем процессорам, выполняющим программу. Контекст содержит описания глобальных, возможно изменяемых, общих данных. Всегда, по умолчанию, имеется один процессор, на котором происходит выполнение программы. На АПК он имеет номер 0. Такой процессор может не отличаться от остальных по устройству и работать без приоритета. Каждый процессор работает по шагам, соответствующим выполнению одной команды процессора, после каждого шага происходит переход к общему механизму управления многопроцессорной программой. Управление может учитывать параметр, задающий порядок перебора команд процессоров, что допускает и учёт приоритетов, и правил тасования, и временной шкалы, и случайного выбора очередного процессора. Если используется временная шкала, то предполагается, что существует дополнительный процессор, непрерывно вырабатывающий очередную метку времени, которую могут учитывать остальные процессоры как данное из общей памяти.

Процессор выполняет только один процесс. Предполагается, что программа — результат компиляции много-поточной программы с предельным распараллеливанием. Общее правило перехода может учитывать порядок перебора команд процессоров. Варьирование порядка перехода используется на этапе отладки программы. При переходе к сетевым программам подразумеваются дополнительные процессоры, выполняющие роль серверов, способных передавать, принимать и хранить сообщения и сигналы. Дополнительные процессоры могут выполнять и действия по освобождению памяти, и другие вспомогательные процессы.

Формально процессоры устроены примерно как абстрактная машина SECD П. Лендина, подробно описанная в книге по функциональному программированию П. Хендерсона, расширенная добавлением набора команд работы с памятью и внешними устройствами как у Н. Вирта в Пи-коде, команд организации взаимодействия процессоров уровня UNIX и транзакционной памятью, как в базах данных, и протоколом изменения данных, как в системах редактирования. Машина SECD работает над четырьмя регистрами: стек для промежуточных результатов, контекст для размещения именованных значений, управляющая вычислениями программа, резервная память (Stack, Environment, Control_list, Dump). Для поддержки много-поточных программ над общей памятью добавлен регистр M – Memory.

Обозначения:

M(SECD)+ - многопроцессорный комплекс над общей памятью.

Запись M(SECD)+ символизирует, что M — общая память для всех процессоров, (SECD)+ - что хотя бы одни процессор SECD обязателен, общее число процессоров произвольно и не исключено изменение их числа в динамике. По умолчанию все регистры АПК могут быть определены как стеки.

M — Memory - общая память, доступная всем процессам.

S — Stack - вычисленные результаты программы.

E — Environment - локальный контекст отдельного процесса.

C — Control_list - программа управления отдельным процессом.

D — Dump - дамп — резервная память обеспечения надежности процесса.

Состояние процесса полностью определяется содержимым этих пяти регистров:

m s e c d → m' s' e' c' d' – переход от старого состояния к новому.

@m (s e c d) → @m' (s' e' c' d') – подчёркнуто использование процессом общей памяти.

Регистры приспособлены к хранению выражений в форме атомов или списков. Система команд АПК представляет собой реализацию базовой семантики (БС) языка много-поточных программ (ЯМП), дополненную рядом системных действий по передаче параметров и защите областей действия, подразумеваемых ЯМП, но не имеющих чёткого синтаксического представления. Такое определение может быть машинно-независимым и переносимым.

Система команд АПК для языков Синпар (ядро) и СИНХРО (оболочка), как примеров ЯМП, различает следующие категории команд, представленные в Приложении 1:

– засылка значений из общей памяти M и локальных регистров C или E в стек S;
– вычисления над безымянными операндами из стека S при обработке выражений;
– пересылка значений из стека S в общую память M или регистр E с учетом локализации участков процесса;
– организация ветвлений, точнее — обходов, что удобно для кодирования программ с параллелизмом и эффективно для архитектур с разрядами управления выполнимостью команд методом прошивки 0/1 в коде команды; [6]
– организация циклов, вызовов процедур или функций, определения которых не сложнее одного комплекта потоков, с сохранением контекста в локальной обстановке E и/или в дампе D с возможностью восстановления;
– расширение комплекса процессов (рядов или комплектов) с передачей их результатов в общую память M и возможностью обмена сообщениями или ожидания событий для синхронизации.

Могут быть и другие категории команд, реализуемые как неконсервативное расширение АПК ради эффективности или решения других задач. Именно так можно рассматривать дополнительные команды, поддерживающие обработку общей памяти и взаимодействие процессов.

Программа для АПК формально считается идеальной, она строится компилятором по результатам статического анализа и на этапе экспериментальной отладки удостоверено, что все данные на устройстве ввода расположены в соответствии с запросами программы, именно те, что нужны. Состояния стеков к моменту выполнения команд сформированы вполне

6 (If... Then... без Else)

корректно, иначе команды не выполняются. Тем не менее, на этапе отладки полезен учёт аварийных ситуаций и обработка неподходящих данных.

Согласно традициям функционального программирования встраиваемые в ядро интерпретатора операции должны соответствовать стандартным правилам доступа к параметрам и размещения единственного «чистого» выработанного результата без побочных эффектов во внешнем контексте. При переходе к много-поточным программам число выработанных результатов может зависеть от числа функционирующих потоков. Переход к единственному результату выполняется специальными мульти-функциями над произвольным числом аргументов по мере необходимости.

Работа многопроцессорной программы с общей памятью при отладке должна быть поддержана в транзакционном стиле, подобно обработке записей в базах данных, т. е. каждое воздействие на память или выполняется полностью, или память восстанавливает состояние до начала выполнения не завершившейся команды. Каждый элемент памяти в любой момент времени обрабатывается только в одном процессе программы, подобно захвату-освобождению файла или устройства. Хранение данных в общей памяти сопровождается протоколом изменений, чтобы при отладке можно было видеть какой процесс внёс изменения и, при необходимости, восстановить утраченное значение.

Стек устроен традиционно по схеме «первый пришел, последний ушел». Размер стека не ограничен. Каждая команда абстрактной машины «знает» число нужных для ее работы элементов стека S и их форматы, элементы она удаляет из стека S и вместо них размещает выработанные результаты, как правило единственный для обычных команд, или ряд результатов для комплектов или потоков действий, размещая число результатов на верху стека. Всегда известно число текущих результатов в стеке S, которые можно явно свернуть в единственный результат специальной редуцирующей операцией - мульти-функцией. Соответствующие команды компилятор размещает в стеке управления процессом для АПК.

Фактически исполняются команды по очереди, начиная с первой в регистре C управляющем процессом, хотя формально считается, что они могут исполняться в произвольном комбинаторном порядке. АПК прекращает работу при выполнении команды «останов» - STOP, которая формально характеризуется отсутствием изменений в состоянии машины. Можно методично дополнить известные описания перечисленных выше команд, обеспечивающих передачу данных из программы и контекста в стек, обработку списков и векторов, проверку данных на атомарность и равенство, организацию ветвлений, циклов, вызова функций или процедур.

Система команд АПК может быть расширена без особого роста трудоёмкости реализации простым дополнением правил для новых команд. Так, можно её механизм распространить на другие типы данных, например, на целые числа. Контроль типов данных на уровне системы команд отсутствует. Считается, что компилятор создал программу после проверки соответствия типов данных и при исполнении кода она уже не нужна. Тем не менее, имеются

операции динамического контроля типов данных, представление которых содержит тэг типа.

Для применения языка программирования (ЯП) на этапе отладки и эксперимента требуется расширение идеализированной семантики языка средствами обмена данными с общей памятью и периферийными устройствами, позволяющими вводить программы и видеть результаты её выполнения. В SECD и то, и другое происходит чудом. Машина SECD достаточна для обеспечения полноты вычислений по Тьюрингу, а для соответствия архитектуре Фон Неймана, учитывающей проблемы подготовки и отладки программ, нужна определённость дисциплины работы с общей или локальной памятью и ввода-вывода данных, что даёт машина M(SECD)+.

Подобные средства Н. Вирт включил в описание Пи-кода для языков Pascal и Oberon, команды которой включены как дополнение и расширение аналога SECD. Разница между SECD и M(SECD)+ в основном сводится к операциям над хранимыми данными. Кроме того, введены дополнительные команды по непосредственной работе с общей памятью для реализации присваиваний, передаче управления для реализации итераторов и обмена с внешними устройствами. Обе абстрактные машины, как SECD, так и M(SECD) +, содержат, кроме образа базовых средств (БС) ЯП, дополнительные команды, обеспечивающие пересылки данных между регистрами и реализационные средства, поддерживающие эффективность реализации ЯП (rplaca, метки и др.) и профилактику некорректного нарушения непрерывности действий при обмене данными.

Регистр M можно представить как список списков, каждый элемент которого начинается с имени глобальной переменной, вслед за которым расположено её текущее значение, а дальше следует протокол изменения значений, выглядящий как последовательность идентификаторов процесса с последующим установленным этим процессом значением. Пусть M содержит элемент X вида:

M = [. . . (**X xt** @At xt @A1 x1 @A2 x2 . . .)]

После изменения процессом @Ak значения переменной X на xk элемент X приобретёт вид:

M' = [. . . (**X xk @Ak xk @**At xt @A1 x1 @A2 x2 . . .)].

Произойдёт побочный эффект присваивания, допускающий при необходимости восстановление прежних значений. Такая реализация позволяет поддержать транзакционность обработки памяти. Эту работу для глобальных переменных в общей памяти выполняет команда SET, обеспечивающая транзакционность.

@At - номер текущего процесса, задавшего изменение значения глобальной переменной. Имена данных в общей памяти все различны.

Различается дисциплина работы с общей и локальной памятью Она поддержана разными командами и подчинена разным требованиям. В стеке S размещается список результатов однократных присваиваний. К верхнему элемен-

ту локального контекста прицепляется этот список в хвост, вслед за аргументами. Предполагается, что имена формальных параметров и локальных переменных различны. Вместо SET используется команда LET - сохранение локальных значений после однократных присваиваний в локальном контексте.

Пересылки и обмен данными нужны для профилактики возникновения временных интервалов между парами взаимосвязанных присваиваний. В результате АПК способен выполнять вычисления несколько более широкого класса, чем задано базовой семантикой данного ЯП. Это приводит к идее определения реализационного замыкания ЯП в соответствии с фактическими возможностями АПК. Такое замыкание выполняет роль прагматического ядра ЯП.

Реально при работе с устройствами через конечное время вырабатывается сигнал, говорящий или об успешном выполнении действия или о причине отказа в его завершении. Здесь для простоты команды определены как абстрактное понимание идеального выполнения действий, без аварийных ситуаций. Выработка сигнала об отказе внешнего устройства приводит к запоминанию в протоколе отказов информации об ущербно выработанных результатах, что можно учесть при отладке и выполнению остаточной программы в стиле смешанных вычислений.

Поддерживается вывод значения из стека S в процесс над внешним устройством OUT и ввод данного из процесса над устройством IN в стек S. В зависимости от специфики устройства введённое данное может исчезнуть из него или сохраниться, что можно рассматривать как разные дисциплины доступа к памяти. Считается, что на внешнем устройстве расположено именно то данное, которое требуется по программе из стека управления «C». При необходимости используется специальное данное «ESC».

OUT' = (x0 . OUT) – на устройстве OUT появляется выведенное данное. В стеке S оно остаётся, чтобы вывод данных можно было вставлять в любой позиции выражений программы.

Иногда, при неудачном срабатывании устройств, может быть необходимость в выравнивании состояния стека. Для этого введена команда **ANY.**

m ((**ESC** . s) e (**ANY** . c) d) → m ((**xany .** s) e c d)
xany - выравнивающее данное,
если был **ESC** — безуспешный ввод.

Для поддержки параллельных вычислений требуется ещё несколько команд по организации комплексов (неупорядоченных комплексов процессов **KIT**) и процессов (последовательности действий - **ROW**), передаче их результатов, локализации воздействий на транзакционную общую память и явную обработку ошибок. Общая память для порождённых процессов доступна через ссылку-адрес **@m** .

Число элементов процесса заранее известно, по их числу последовательно порождаются независимые автоматы для последовательного запуска элементов процессов: @AM1, @AM2 и т.д.

Команда **NEXT** даёт поддержку строго императивного запуска элементов процесса. Без этого второй элемент процесса работает, не дожидаясь завершения предшественника. Забрали в основной стек результаты выполненного процесса и можно запустить следующий. Реализация сообщений и ожиданий подобна рандеву в языке ADA – обмен происходит между активными процессами и каждый знает что и кому он передаёт или от кого сообщение получает.

COMSUB – компиляция «на лету» фрагмента программы, заданного в виде схемы-макроса и списка значений его фрагментных переменных. После символьной подстановки происходит компиляция, результат которой размещается в регистре управления выполнением процесса.

Реализация таких команд АПК существенно зависит от распределения памяти и независимости её частей, что приводит к необходимости контроля границ памяти при индексировании векторов и доступе по указателю. При компиляции может быть создана программа для АПК, пригодная для безаварийного выполнения при условии бесперебойного функционирования всех устройств. Все внешние данные соответствуют требованиям программы и команды устройств срабатывают неделимо.

Дальнейшее расширение ЯП до его полной оболочки может быть сведено к подключению дополнительных типов данных и присоединению расширенных семантик пошаговым методом:

- повышение продуктивности начального этапа программирования и экспериментальной отладки программ;
- увеличение потенциала системы программирования, т.е. числа типовых задач, решение которых обладает приемлемой для практики трудоёмкостью.

Семантический спуск от полного ЯП к его базовой семантике характеризуется повышением продуктивности реализации ядра ЯП и его АПК, одновременно с увеличением трудоёмкости применения частичной реализации ЯП.

Предложенное описание диалектов языка СИНХРО позволяет разработку много-поточной программы начинать с создания семейства независимых потоков в стиле чисто функционального программирования. Полученное семейство затем, в процессе ряда шагов экспериментальной отладки, перерабатывается в комплект взаимодействующих потоков над общей памятью, допускающий выполнение на разных многопроцессорных комплексах.

Синтаксическая сводка языка приведена в Приложении 2. Предварительная версия представлена как описание языка СИНХРОН в препринте [2]. Более поздние, дополнительные соображения по целям и задачам проекта и принятию решений опубликованы в ряде статей [3-6].

6. ПРИМЕРЫ ЗАДАЧ С РЕШЕНИЯМИ

«Вот гвоздь, вот подкова.
Раз, два и готово!»
С. Я. Маршак

Показать и пронаблюдать общие модели параллельных вычислений и связанные с их применением явления можно на примерах задач, описанных в книге Т. Хоара [22]. Используется материал олимпиадных задач и учебных примеров из школьных и факультативных курсов информатики. Кроме того, подобраны задачи для демонстрации конкретных проблем организации взаимодействия процессов, таких как «состояние гонки» и других «плавающих» ошибок, проявляющихся в случайные моменты времени и при изменении границ доступа к данным, «пропадающих» при попытке их локализовать. Решения учебных задач представлены на языке СИНХРОН, наследующем конструкции языков Grow и Робик с учетом современных технологий разработки информационных систем и разных парадигм программирования, позволяющих в разных формах представлять семантически эквивалентные программы, порождающие одинаковые семейства допустимых процессов. Отметим сразу, что использование разных форм для представления одних и тех же решений отнюдь не означает равносильности форм. Продуктивность форм различается в зависимости от их соответствия методам решения задач.

Примеры 6.1-6.6 и 6.25-6.29 ориентированы на диалект Асинхр, примеры 6.7-6.20 на диалект Синпар, примеры 6.21-6.24 на диалект Трансформ.

*6.*1. Вводные примеры из книги Т. Хоара (Могопоточность, Асинхр)

Пример 1. Простой торговый автомат — ПТА, способный принимать монетки и за них выдавать шоколадки. Описать в табличной форме сценарий работы автомата ПТА, принимающим монету, а потом выдающим шоколадку, что можно выразить формулой **(! мон ; ? шок),** где круглые скобки и «;» указывают, что это **очередь** последовательных действий. Команды:

! – **принимать** данное

? – **вывести** данное

универсальны, они доступны в любом автомате.

Переменные «мон» и «шок» описаны внутри ПТА и используются в его сценарии.

Таблица 6

Операторная, функциональная, предикатная и табличная формы семантически эквивалентных представлений очереди последовательных действий, допускающей выполнение одним автоматом.

Фрагмент ПТА	*Комментарий*
! мон ; **? шок**	Линейный участок двух последовательных действий

(! мон ; ? шок)	Выражение для потока из двух последовательных действий
! мон → ? шок	Предикатная форма двух последовательных действий
<table><tr><th>действия</th><th>примечание</th></tr><tr><td>**! мон**</td><td>*сначала прием монетки*</td></tr><tr><td>**? шок**</td><td>*потом выдача шоколадки*</td></tr></table>	Табличная форма представления потока из двух последовательных действий

Ради компактности то же самое можно представить более просто при условии согласования смысла клеток.

Таблица 7

Четыре представления одного и того же семейства действий

<table><tr><th>действия</th><th>примечание</th></tr><tr><td>**! мон**</td><td>*сначала прием монетки*</td></tr><tr><td>**? шок**</td><td>*потом выдача шоколадки*</td></tr></table>	**! мон ;** **? шок**
(! мон ; ? шок)	**! мон → ? шок**

Пример 2. Простой торговый автомат может работать с доверием. Описать сценарий управления автоматом ПТА_Д, допускающим выдачу шоколадки до получения монетки — доверяя покупателю, что можно выразить формулой **[! мон , ? шок]**, в которой квадратные скобки и запятая указывают, что это **набор** независимых действий. Можно взять шоколадку, а потом дать монетку или наоборот.

Таблица 8

Операторная, функциональная, предикатная и табличная формы семантически эквивалентных представлений набора независимых действий, выполняемых возможно разными автоматами.

Фрагмент ПТА_Д	*Комментарий*
[! мон , ? шок]	Участок из двух независимых действий
{ ! мон , ? шок }	Выражение для слоя из двух независимых действий
! мон ^ ? шок	Предикатная форма двух независимых действий
<table><tr><th colspan="2">действия</th><th>примечание</th></tr><tr><td>**! мон**</td><td>**? шок**</td><td>*прием монетки и выдача шоколадки*</td></tr></table>	Табличная форма представления слоя из двух независимых действий

Можно сразу обратить внимание, что табличная схема более чётко показывает разницу между набором и очередью действий, чем другие, более привычные формы.

Таблица 9

Четыре формы .представления одного и того же семейства действий

<table><tr><th colspan="2">действия</th><th>примечание</th></tr><tr><td>! мон</td><td>? шок</td><td>прием монетки и выдача шоколадки</td></tr></table>	**[! мон , ? шок]**
{ ! мон , ? шок }	**! мон ^ ? шок**

Пример 3. Простой торговый автомат, допускает переход от управления автоматом к программе с учетом числа шоколадок в автомате. Автомат может по разным причинам не выдать шоколадку, например, из-за их отсутствия. Пусть обстановка в N хранит число имеющихся в автомате шоколадок. Описать сценарий для автомата ПТА_У, сообщающий покупателю об отсутствии товара и возвращающий монету, что можно выразить формулой

(N → **(! мон ; ? шок) ; ? мон ; «шоколадок нет»)**.

Таблица 10

Операторная, функциональная, предикатная и табличная формы представления потока действий, выполнение которых обусловлено.

Фрагмент ПТА_У	*Комментарий*
N = 4; **! мон ;** [ЕСЛИ N ТО [**? шок ;** N = N - 1] ; **] ;** КРОМЕ N ТО [**? Мон ;** **? «шоколадок нет»]**	Операторная форма: *4 - число шоколадок в автомате* *прием монетки* *условие готовности автомата* *выдача шоколадки* *учет остатка в контексте* *возврат монетки, если выдачи не было* *диагностическое сообщение*
(N = 4; ! мон ; **(N → (? шок ; (N = N - 1)) ;** **(? мон ; ?«шоколадок нет»)))**	Выражение для комплекта из двух обусловленных действий
N = 4 **! мон** **N → (? шок ; N = N - 1)** **(~N → (? мон ; ?«шоколадок нет»)**	Предикатная форма двух независимо обусловленных действий
<table><tr><th>условие</th><th>действия</th><th>примечание</th></tr><tr><td></td><td>N = 4;</td><td>Число шоколадок</td></tr></table>	Табличная форма представления комплекта из двух обусловленных

	! мон	*Прием монетки*	действий
N	**? шок**	*Выдача шоколадки*	
	N = N - 1	*Учёт остатка*	
~N	**? мон**	*Возврат монетки*	
	?«шоколадок нет»	*Диагностика*	

Здесь задано, что в ПТА_У хранится известное число шоколадок, например 4. После приёма монетки включается комплект действий, допускающих обход:

при наличии шоколадок запускается выдача шоколадки и пересчёт числа оставшихся шоколадок;

при отсутствии шоколадок возвращается монетка. Операции «=» и «-» универсальны, т. е. общедоступны в любом автомате.

Можно обратить внимание, что некоторые очереди действий функционально эквивалентны наборам асинхронных действий, что более заметно по табличной форме, чем по более привычным операторным формам и выражениям.

Таблица 11

Четыре формы .представления потока действий с обходами

условие	действия	примечание
	N = 4;	*Число шоколадок*
	! мон	*Прием монетки*
N	**? шок**	*Выдача шоколадки*
	N = N - 1	*Учёт остатка*
~N	**? мон**	*Возврат монетки*
	?«шоколадок нет»	*Диагностика*

```
 N = 4;
! мон ;

 [ ЕСЛИ  N
      ТО  [  ? шок ;
             N = N - 1] ; ] ;
 ЕСЛИ_НЕ n ТО  [ ? Мон ;
       ?«шоколадок нет»
                ]
```

```
( N = 4 ;
  ! мон  ;
  (N →  ( ? ш о к  ;  ( N  =  N  - 1 )  )  ;
        (?мон  ;  ?«шоколадок нет» ) )
)
```

```
 N = 4
! мон
 N →  ( ? шок ;  N = N - 1)
~N →
  (? мон  ;  ?«шоколадок нет» )
```

Пример 4. Простой торговый автомат может допускать многократное пополнение запасов шоколадок с помощью специально подготовленной функции ПТА_Ф (n), сообщающей число загруженных в автомат шоколадок. Описать сценарий автомата, который после исчерпания запасов может принимать пополнение и снова работать. Определение функции можно представлять как слой в программе, например, это видно по определению

функции от числа, заданной фрагментом из Примера 3, учитывающим наличие шоколадок.

Таблица 12

Операторная, функциональная, предикатная и табличная формы представления определения функции и её вызова.

<table>
<tr><th>Фрагмент автомата ПТА_Ф,</th><th>Комментарий</th></tr>
<tr><td>пта_Ф = (n)
(! мон ;
[ЕСЛИ n
ТО [? шок ;
пта_Ф (n - 1)]
, КРОМЕ n ТО (? мон
; ? «шоколадок нет»)
]
)

пта_Ф (4; 10; 100) ;</td><td>число шоколадок в автомате
приём монетки
условие готовности автомата
выдача шоколадки
учет остатка
возврат монетки

запуск автомата ПТА,
последовательно выдающего
4, 20 и 100 шоколадок.</td></tr>
<tr><td>(ОПРЕД пта_Ф (n)
(! мон ;
(n →{ ? шок ; пта_Ф (n - 1) } ;
{ ? мон ,; ? «шоколадок нет» }))

(пта_Ф (4 20 100))</td><td>Функция задаёт число шоколадок.
выражение приёма монетки,
затем ветвление в зависимости
от числа шоколадок.
Просачивание функции
на список чисел шоколадок</td></tr>
<tr><td>пта_Ф (n)
пта_Ф (0) →(! мон
; [? мон] ; [? «шоколадок нет»]
пта_Ф (n) →{ ! мон
;[? шок] , [пта_ф (n - 1)] }</td><td>при аргументе 0 возврат
монетки
иначе выдача шоколадки
\и уменьшение счётчика</td></tr>
<tr><td>
<table>
<tr><th>Условие</th><th>Действия</th><th>Комментарий</th></tr>
<tr><td>пта_Ф (n)</td><td>Порядок действий</td><td>Объявление функции</td></tr>
<tr><td></td><td>! мон</td><td>приём монетки</td></tr>
<tr><td>n</td><td>? шок</td><td>выдача шоколадки</td></tr>
<tr><td></td><td>пта_ф (n - 1)</td><td>учет остатка</td></tr>
<tr><td>~n</td><td>? мон</td><td>в о з в р а т</td></tr>
<tr><td></td><td>? «шоколадок нет»</td><td>Диагноз</td></tr>
<tr><td colspan="3">Разделение слоёв: переход к вызову</td></tr>
<tr><td></td><td>пта_Ф (4; 20; 100)</td><td>Новые шоколадки по порядку</td></tr>
</table>
</td><td>Имя функции рассматривается как предикат и размещается в самой левой колонке.</td></tr>
</table>

Таблица 13

Четыре формы .представления функции для выполнения одинаковых потоков действий

Условие	Действия	*Комментарий*
пта_Ф (n)	**Порядок действий**	**Объявление функции**
	! мон	приём монетки
n	? **шок**	выдача шоколадки
	пта_ф (n - 1)	учет остатка
~n	**? мон**	**в о з в р а т**
	? «шоколадок нет»	Диагноз
Разделение слоёв: переход к вызову		
	пта_Ф(4;20;100)	*Новые шоколадки по порядку*

```
пта_Ф = (n)

 ( ! мон ;
  [ ЕСЛИ n
        ТО  [ ? шок ;
        пта_Ф (n - 1) ]
  , КРОМЕ n ТО (  ? мон
        ; ? «шоколадок нет»  )
   ]
)

пта_ Ф   ( 4 ; 1 0 ;   0 0 )    ;
```

```
(ОПРЕД пта_Ф (n)
    ( ! мон ;
( n →{ ? шок  ;  пта_Ф (n - 1) } ;
      { ? мон  ,; ? «шоколадок нет»  } )))

( пта_Ф  (4 20 100)  )
```

```
пта_Ф (n)
пта_Ф (0) →( ! мон
        ; [? мон ]
      ; [ ? «шоколадок нет» ]
пта_Ф (n) →{ ! мон
  ;[? шок ] , [ пта_ф (n - 1) ]  }

пта_Ф (4; 20 ; 100)
```

Можно обратить внимание, что механизм просачивания позволяет серийные действия по вызову функции рассматривать как одно действие над структурой данных. Если функция определена над числом, то при подаче ей структуры над числами в качестве аргумента функция применяется к каждому из чисел и результаты размещаются в подобную структуру данных.

Пример 5. Простой торговый автомат ПТА_N_M, выполняющий подсчет выручки. Считаем, что монетоприемник по объему превышает объем возможной выручки. Желательно решение сделать максимально распараллеленным.

Таблица 14

Операторная форма и табличная форма представления функции для распараллеленной программы.

Операторная форма	
пта_фв = (n m)	*m – число монет в автомате*
(! **мон ;**	*n — число шоколадок*
	прием монетки
[ЕСЛИ n	*условие готовности автомата*

TO [? **шок ,** пта_фв (n - 1, m + 1)] **,** КРОМЕ n TO (? **мон ,** **?«шок нет» ,** РЕЗ (m) **)** **]** **)** в = в + пта_фв ((4, 20, 100), в)	*выдача шоколадки* *учет остатка и подсчет выручки* *возврат монетки и диагноз* *выручка* *добавление шоколадок в произвольном порядке*

Табличная форма

пта_фв (n m)	**Действия**			*m – число монет в автомате* *n — число шоколадок*
	! мон			*прием монетки*
n	**? шок**	пта_фн (н-1, м+1)		*готовность и выдача*
~n	? **мон**	**?«шок нет»**	РЕЗ (m)	*возврат монетки и выручка*
Разделение слоёв: переход к вызову				
	в = в + пта_фв ((4, 20, 100), в)			*Добавки в любом порядке*

Определение функции от числа шоколадок и накопленной выручки внешне выглядит предельно распараллеленным, но интуитивно ясно, что такое определение скрывает параллельное выполнение подсчёта выручки и числа шоколадок, не считая собственно общей схемы управления процессом..

Пример 6. Счетчики (локализация функций). Разумно построить универсальные функции для подсчёта выручки и для учёта числа шоколадок, выделить их из общего процесса.

Таблица 15

Операторная и табличная формы представления потока действий,9 выполнение которых обусловлено.

<table>
<tr><th>Операторная форма</th><td>Комментарий</td></tr>
<tr><td>Учет = (n)
{ ЕСЛИ n ТО
Учет (n - 1) }</td><td>Выделение из примеров 4 и 5
функции учёта шоколадок</td></tr>
<tr><td>Выручка = (n m)
[ЕСЛИ n ТО
Выручка (n (m + 1))
, КРОМЕ n ТО РЕЗ (m - 1)]</td><td>Выделение из примера 5
функции подсчёта выручки</td></tr>
<tr><th>Табличная форма</th><th>Выражение</th></tr>
<tr><td><table>
<tr><td>Учёт (n)</td><td>Действия</td><td>Функция</td></tr>
<tr><td>n</td><td>Учёт (n - 1)</td><td>Учет остатка</td></tr>
</table></td><td>Учет (n) = (n → Учет (n - 1))</td></tr>
<tr><td></td><th>Предикатная форма</th></tr>
<tr><td><table>
<tr><td>Выручка(n m)</td><td>Действия</td><td>Функция</td></tr>
<tr><td>n</td><td>Выручка (n (m+1))</td><td>Рост выручки</td></tr>
<tr><td>~n</td><td>РЕЗ (m-1)</td><td>Коррекция</td></tr>
</table></td><td>Выручка (n m)
Выручка (0 m) → РЕЗ (m - 1)
Выручка (n m) → Выручка (n (m+1))</td></tr>
<tr><td>Разделение слоёв: переход к вызову</td><td></td></tr>
</table>

Можно обратить внимание на синтаксическое подобие различных форм в случае разных конкретных функций. Шаги таких рекурсивных универсальных функций следует синхронизовать с общей схемой управления работой автомата, содержащей вызовы этих функций и задающей взаимодействие с помощью барьеров.

Синпар — подъязык программирования взаимодействующих потоков.

Пример 7. Взаимодействие процессов. Расстановка барьеров для возможности взаимодействия со счётчиком. Имеются описания разных счетчиков, контролирующих работу автомата. Надо синхронизовать действия счетчиков.

Таблица 16

Бесконечный процесс обслуживания без контроля и с внешним контролем.

<table>
<tr><th>Операторная форма</th><td>Комментарий</td></tr>
<tr><td>пта_б = ( )
(! Мон ; \\Сначала прием монетки
? Шок ; \\Потом выдача шоколадки
пта_б) \\Следующий шаг цикла</td><td>Зацикливание без контроля</td></tr>
<tr><td>пта_б = ( )
(! Мон ; \\Сначала прием монетки
? Шок ; \\Потом выдача шоколадки
Контр: пта_б) \\Следующий шаг цикла</td><td>Зацикливание, допускающее внешний контроль с помощью внутреннего барьера</td></tr>
<tr><th>Табличная форма</th><td></td></tr>
<tr><td>
<table>
<tr><td>пта_б</td><td></td><td>Примечание</td></tr>
<tr><td></td><td>! мон</td><td>Сначала прием монетки</td></tr>
<tr><td></td><td>? шок</td><td>Потом выдача шоколадки</td></tr>
<tr><td></td><td>пта_б</td><td>Следующий шаг цикла</td></tr>
<tr><td colspan="3">Разделение слоёв: переход к вызову</td></tr>
</table>
</td><td>Зацикливание без контроля</td></tr>
<tr><td>
<table>
<tr><td>пта_б</td><td></td><td>Примечание</td></tr>
<tr><td></td><td>! мон</td><td>Сначала прием монетки</td></tr>
<tr><td></td><td>? шок</td><td>Потом выдача шоколадки</td></tr>
<tr><td>Контр</td><td>пта_б</td><td>Следующий шаг цикла</td></tr>
<tr><td colspan="3">Разделение слоёв: переход к вызову</td></tr>
</table>
</td><td>Зацикливание допускающее внешний контроль, ради которого введён внутренний барьер</td></tr>
</table>

При подготовке каждого потока можно наметить позиции, допускающие внешний контроль для взаимодействия процессов.

Пример 8. Взаимодействие процессов. Расстановка внутренних барьеров для возможности внешнего управления процессами.

Таблица 17

Разметка фрагментов из примеров 6 и 7 внутренними барьерами ради возможной синхронизации процессов учета числа шоколадок и шагов обслуживания.

<table>
<tr><th>Операторная форма</th><th>Комментарий</th></tr>
<tr><td>учет_б = (n)
[ЕСЛИ n ТО
Контр : учет_б (n - 1)
]</td><td rowspan="2">Шаги функции «Учёт» могут быть синхронизованы с шагами функции «пта-б», что выражено одноимённостью барьеров.</td></tr>
<tr><td>пта_б = ( )
(! Мон ;
? Шок ;
Контр : пта_б)</td></tr>
<tr><td colspan="2"></td></tr>
<tr><th>Табличная форма</th><th>Предикатная форма</th></tr>
<tr><td rowspan="6"><table>
<tr><td>Учёт (n)</td><td>Действия</td><td>Функция</td></tr>
<tr><td>n</td><td rowspan="2">Учёт (n - 1)</td><td rowspan="2">Учет остатка</td></tr>
<tr><td>Контр:</td></tr>
<tr><td colspan="3">Разделение слоёв: переход к вызову</td></tr>
<tr><td>пта_б</td><td></td><td>Примечание</td></tr>
<tr><td></td><td>! мон</td><td>Сначала прием монетки</td></tr>
<tr><td></td><td>? шок</td><td>Потом выдача шоколадки</td></tr>
<tr><td>Контр:</td><td>пта_б</td><td>Следующий шаг цикла</td></tr>
<tr><td colspan="3">Разделение слоёв: переход к вызову</td></tr>
</table></td><td>Учет (n)
Учет (n) → ((Барьер 'Контр);
(Учет (n - 1)))))</td></tr>
<tr><td>Выражение</td></tr>
<tr><td>(ОПРЕД пта_б ( )
((! мон ; ? шок ;
(Барьер 'Контр);
(пта_б))))</td></tr>
</table>

Обращаем внимание, что выражения и предикатная форма не имеют синтаксической позиции для простановки барьеров. Для разных функций могут быть удобны разные формы. При необходимости используется специальная функция «Барьер», изменяющая правило интерпретации выражений для учёта синхронизирующих воздействия. Это показывает условность границы между синтаксисом и семантикой языка программирования. Синтаксические средства операторной и табличной форм обеспечиваются операциями в выражениях и предикатной форме.

Пример 9. Определение взаимодействия процесса со счётчиком. Внешнее объявление барьеров для синхронизации функций, образующих процессы.

Таблица 18

Внешний барьер запускает механизм синхронизации потоков с одноимёнными внутренними барьерами.

<table>
<tr><th colspan="4">Операторная форма</th><th>Комментарий</th></tr>
<tr><td colspan="4">пта_и_учет = (n)
    [Контр] [пта_б , учет_б (n)]</td><td rowspan="4">n шагов рекурсии функции «учет_б» и общий внешний барьер «Контр» с «пта_б».

Обе функции сработают до своих барьеров «Контр», а после этого перейдут к следующему шагу рекурсии. Когда «n» сравняется с 0, то обе функции остановятся.</td></tr>
<tr><td colspan="4">Табличная форма</td></tr>
<tr><td>пта_и_учет (n)</td><td colspan="2">Комплект вызовов функций</td><td>Вызывает функции с внутренними барьерами</td></tr>
<tr><td>Контр</td><td>пта_б</td><td>учет_б (n)</td><td>Слой синхронизиваных потоков</td></tr>
</table>

Барьер в рекурсивной функции «УЧЕТ_Б» и в бесконечном процессе «ПТА_Б» ограничивает число выполнимых шагов числом шоколадок.

Пример 10. Автомат с заманиванием. Возможна организация процесса, выдающего иногда подарок.

Таблица 19

Синхронизация с вероятностным действием, выполняемым не каждый раз.

<table>
<tr><th>Операторная форма</th><th>Комментарий</th></tr>
<tr><td>пта_б = ( )
(! Мон ;
Контр : ? Шок , пта_б)</td><td>Бесконечный процесс,
выдающий шоколадки</td></tr>
<tr><td>Подарок = ( )
(Контр: \\ ВЕРОЯТНО ИНОГДА
% ? «возьми подарок» ; подарок)</td><td>Бесконечный процесс,
иногда выдающий подарок</td></tr>
<tr><td>пта_и_подарок = ( )
[Контр] [пта_б , подарок]</td><td>n шагов рекурсии
и внешний барьер для выдачи
подарка, лишь если была шоколадка</td></tr>
<tr><th>Табличная форма</th><td></td></tr>
<tr><td>
<table>
<tr><td>пта_б ( )</td><td>Действия</td><td></td><td>Функция</td></tr>
<tr><td></td><td colspan="2">! Мон</td><td></td></tr>
<tr><td>Контр</td><td>? Шок</td><td>пта_б</td><td>Приостановка</td></tr>
</table>
</td><td>Приостановки перед выдачей
шоколадки</td></tr>
<tr><td>Разделение слоёв: переход к вызову</td><td></td></tr>
<tr><td>
<table>
<tr><td>подарок ( )</td><td>Действия</td><td>Функция</td></tr>
<tr><td>Контр</td><td>% ? «возьми подарок»</td><td></td></tr>
</table>
</td><td>предложение забрать подарок</td></tr>
<tr><td>Разделение слоёв: переход к вызову</td><td></td></tr>
<tr><td>
<table>
<tr><td>пта_и_
подарок ( )</td><td>Действия</td><td></td><td>Функция</td></tr>
<tr><td>Контр</td><td>пта_б</td><td>подарок</td><td>Если была
шоколадка</td></tr>
</table>
</td><td>внешняя синхронизация подарка с
шоколадкой</td></tr>
</table>

Бесконечный процесс, совмещающий выдачу шоколадок с эпизодической выдачей подарка.

Пример 11. Настройка автомата с заманиванием и подсчёта числа выданных призов независимо от выдачи шоколадок.

Таблица 20

Очередь внешних барьеров влияет на порядок выполнения слоёв.

<table>
<tr><th>Операторная форма</th><th>Комментарий</th></tr>
<tr><td>пта_б = ( )
(! Мон ;
Контр : ? Шок ,
пта_б)</td><td>«учет_б» « подарок» из примера 7.</td></tr>
<tr><td>Выручка = (m)
[
Контр_В: Выручка (m + 1)
, КРОМЕ n ТО РЕЗ (m - 1)
]</td><td>Другой барьер, синхронизующий выдачу приза и подсчёт числа шоколадок</td></tr>
<tr><td>пта_учёт_и_подарок = (n)
[Контр_В , Контр] [пта_б ,
Выручка(0), учет_б (n) , подарок
]</td><td>n шагов рекурсии
Общие внешние барьеры для разных потоков..
Выручка считается раньше остального.</td></tr>
<tr><td colspan="2">Табличная форма</td></tr>
<tr><td colspan="2">
<table>
<tr><td>пта_б ( )</td><td colspan="2">Действия</td><td>Функция</td></tr>
<tr><td></td><td colspan="2">! Мон</td><td></td></tr>
<tr><td>Контр</td><td>пта_б</td><td>? Шок</td><td>Основной цикл</td></tr>
</table>
<table>
<tr><td>Выручка (m)</td><td>Действия</td><td>Функция</td></tr>
<tr><td>Контр_В</td><td>Выручка (m + 1)</td><td>Число монеток</td></tr>
<tr><td>~n</td><td>РЕЗ (m - 1)</td><td>Уточнение</td></tr>
</table>
<table>
<tr><td>пта_учёт_и_подарок(n)</td><td colspan="3">Действия</td><td></td></tr>
<tr><td>Контр_В</td><td colspan="2">Выручка(0)</td><td></td><td></td></tr>
<tr><td>Контр</td><td>пта_б</td><td>учет_б (n)</td><td>подарок</td><td></td></tr>
</table>
</td><td>Два барьера будут достигнуты по очереди, одновременно двух барьеров не бывает.</td></tr>
</table>

Пример 12. Варианты взаимодействий. Внешнее управление бесконечными процессами.

Таблица 21

Бесконечный процесс завершается благодаря синхронизации с конечным потоком

<table>
<tr><th>Операторная форма</th><th>Комментарий</th></tr>
<tr><td>пта_учет_выр = (n_ш n_м) [Контр]
[пта_б ( ) ,
учет_б (n_ш),
выручка (n_м)] \\выручка не синхронизована

пта_и_учет = (n_ш)
[Контр] [пта_б , учет (n_ш)]</td><td>барьер не во всех процессах
автомат с учетом числа шоколадок и монет

n_ш шагов рекурсии
общий барьер</td></tr>
<tr><td>Табличная форма</td><td></td></tr>
<tr><td>
<table>
<tr><td>пта_учет_выр (n_ш n_м)</td><td colspan="3">Действия</td><td></td></tr>
<tr><td>Контр</td><td>пта_б</td><td>учет_б (n_ш)</td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td>выручка (n_м)</td><td></td></tr>
</table>
<table>
<tr><td>пта_и_учет (n_ш)</td><td colspan="2">Действия</td><td>Функция</td></tr>
<tr><td>Контр</td><td>пта_б</td><td>учет (n_ш)</td><td>Основной цикл</td></tr>
</table>
</td><td>Бесконечные процессы пта_б завершатся через n_ш шагов благодаря синхронизации с функцией учет_б</td></tr>
</table>

Пример 13. Варианты взаимодействий. Соседство с асинхронностью.

Таблица 22

Внутренний барьер вне области влияния внешнего барьера.

<table>
<tr><th colspan="4">Операторная форма</th><th>Комментарий</th></tr>
<tr><td colspan="4">пта_учет_дар = (n_ш) [
[Контр] [пта_б () ,
учет_б (n_ш)] ,
подарок]</td><td rowspan="5">2 процесса синхронизованы,

«подарок» вне действия внешнего барьера

автомат заманивает и без шоколадок</td></tr>
<tr><td colspan="4">Табличная форма</td></tr>
<tr><td>пта_учет_дар (n_ш)</td><td colspan="2">Действия</td><td></td></tr>
<tr><td>Контр</td><td>пта_б</td><td>учет_б (n_ш)</td><td></td></tr>
<tr><td></td><td colspan="2"></td><td>подарок</td></tr>
</table>

6.2. Редактирование фрагментов (Синхронизация - Синпар)

Использование одноимённости в управлении синхронизацией создаёт довольно жёсткие ограничения. Предварительная разметка потоков барьерами может не соответствовать оперативным решениям по синхронизации в разных наборах потоков. Для согласования имён барьеров и переменных можно использовать технику редактирования, включая программируемую макротехнику. Редактирование фрагментов кроме простого сцепления фрагментов использует синтаксические макросы, вид параметров которых задаётся как синтаксическое подобие вхождению переменной в заданную строку согласно синтаксису используемого языка сценариев.

Пример 14. Функции для параметризация действий с заданием реквизита исполнителей.

Таблица 23

Макрос для программируемого редактирования имён переменных.

Фрагмент	*Комментарий*
ф_пта = МАКРО (м ~~ “!м” ш ~~ “ ?ш ”) **(** !м **;** ?ш **; Контроль:** ф_пта **)**	*проверка на пригодность быть операндами для ! и ?* *по умолчанию параметры сохраняются*

Параметры для вставки в синтаксически подобный фрагмент определения с зацикливанием. Задан вид параметров, первый должен быть пригоден для приёма данных, а второй — для выдачи.

Пример 15. Конкретизация параметров действий с заданием реквизита исполнителей.

Таблица 24

Вызов макроса для смены реквизита при сохранении сценария

Фрагмент	*Комментарий*
ф_пта (**@жетон, @игрушка**)	\\ *явная параметризация отложенных действий*

Получится другой автомат, сценарий которого содержит синтаксически подобные фрагменты, соответствующие виду параметров. Жетон пригоден для приёма, а игрушка для выдачи.

Пример 16. Вызов исполнителя ИТА — игровой торговый автомат со сменным реквизитом.

Таблица 25

Реорганизация торгового автомата с продажи шоколадок на выдачу игрушек за жетоны.

Фрагмент	*Комментарий*
[Контроль] **ИТА !! [**ф_пта (**@жетон, @игрушка**) **,** выручка (0) **]**	*явное задание исполнителя*

Предполагается, что исполнитель ИТА способен выполнять запрограммированные функции.

Пример 17. Управление «ИТА.выручка» не синхронизовано с рекурсией, в нем другой барьер.

Таблица 26

Неполная синхронизация из-за разноимённых внутренних барьеров.

Фрагмент	*Комментарий*
учет = **(** n **)** ЕСЛИ n ТО **Контр:** учет_б (n – 1)	
ф_пта **(@мон @шок) =** **(! мон ; ? шок ;** **Контроль:** ф_пта **)**	Вызов макроса на случай шоколадок
[Контр] ИТА!! **[** ф_пта (**жетон, игрушка**) **,** учет (4) , выручка **]**	*выдать 4 игрушки*

Синхронизация не происходит из-за разноимённости внутренних барьеров

Пример 18. Параметр-барьер позволяет имя барьера поменять.

Таблица 27

Макрос для замены имени барьера в определении потока с проверкой синтаксической эквивалентности параметра

Фрагмент	*Комментарий*
учет_б **= (** n **b ~~ “ b: ”)** учёт = (n) ЕСЛИ n ТО **b:** учет_б (n – 1) учет_б (10 **контр**)	*смена имени - параметр-барьер*

Пример 19. Переименование барьера и реквизита

Таблица 28

Макрос для синтаксически эквивалентной подстановки разных параметров

Фрагмент	*Комментарий*
ф_пта_б = (м ~~ “! м”, ш ~~ “ ? ш ”, **р ~~ “ b: ”**) **(!** м **; ?** ш **; р:** ф_пта_б **)**	*Смена имён :* *параметры — реквизит и барьер*

Пример 20. Синхронизация параметризованными барьерами

Таблица 29

Согласование имени внутреннего барьера.

Фрагмент	*Комментарий*
[Контр] [ф_пта_б (@жетон, @игрушка, **Контр**) **,** учет_б (4, **Контр**)**]**	\\ *подстановка барьера* \\ *в автомате 4 игрушки*

Трансформ — подъязык преобразования программ

Пример 21. Разложить простой автомат ПТА на два более простых: принимающий монету и выдающий шоколадку.

Таблица 30

Вспомогательный барьер для разбиения потока.

Фрагмент	*Комментарий*
[роб_мон !! (! **мон**; к:) ; роб_шок !! (к: ? **шок**)]	\\ *(! мон ; ? шок)* \\ *сначала прием монетки,* \\ *потом выдача шоколадки*

Сначала один робот принимает монетки и может быть приостановлен по барьеру «к», а другой робот в это момент по своему барьеру «к» начнёт выдавать шоколадки.

Пример 22. Простой торговый автомат с доверием разложить на два автомат: один автомат допускает выдачу шоколадки до получения монетки другим автоматом, доверяя покупателю.

Таблица 31

Вспомогательный барьер для разбиения набора.

Фрагмент	*Комментарий*
[роб_мон !! [! **мон; к:**] , роб_шок !! [? **шок; к:**]]	\\ **[** ! мон **,** ? шок **]** \\ *прием монетки и независимо* \\ *выдача шоколадки в любом порядке*

Один робот принимает монетку и другой выдаёт шоколадку в произвольном порядке, а после этого оба могут быть приостановлены.

Пример 23. Определение робота, работающего как простой автомат ПТА, принимающий монету, а затем выдающий шоколадку в последовательно формируемом контексте.

Таблица 32

Программа исполнителя состоит из контекста и сценария

Фрагмент	*Комментарий*
Робот_пта = { ИМЕНА **мон , шок** ; КНОПКА дай ! **мон** ; КНОПКА возьми ? ш**ок;** [ ] **(** ! **мон ;** ? **шок** **)** }	\\ *Объявление робота* - ***контекст:*** \\ *Описание имён и реквтизита* \\ *кнопка приёма монеток* \\ *кнопка выдачи шоколадок* ***Сценарий:*** \\ *без внешних барьеров* \\ *сначала прием монетки,* \\ *потом выдача шоколадки*

Пример 24. Расширение обстановки для игры, предоставляющей оперирование роботом, работающим как простой автомат ПТА, принимающий монету и выдающий шоколадку в произвольном порядке без упорядочения формирования контекста.

Таблица 33

Фрагмент	*Комментарий*
{ Робот_пта += КНОПКА дай , КНОПКА возьми , КНОПКА шаг , [ ] **[** дай **,** возьми **]**	\\ *Встраивание робота* \\ *кнопка приёма реквизита* \\ *кнопка выдачи реквизита* \\ *кнопка оперирования роботом* \\ *без синхронизации* \\ ***сценарий*** *допустимых нажатий:* \\ *прием ,* \\ *выдача*

6.3. Олимпиадная задача для младших школьников (Асинхр)

Ещё один пример — олимпиадная задача о жарке **трёх котлет на двух сковородах**. Надо 3 котлеты как можно скорее поджарить, имея только 2 сковороды, по размеру каждая пригодна для жарки одной котлеты. Для обжарки одной стороны котлеты требуется 5 минут. На одном из Intel-семинаров в Москве эта задача упоминалась как пример немасштабируемого алгоритма. Надо предложить и ясно записать масштабируемое решение.

Пример 25. 2 сковороды — 3 котлеты (производительность)

Написать программу жарки 3-х котлет на двух сковородах такую, что сковороды не простаивают.
3 котлеты на 2 сковороды.
Для обжарки одной стороны котлеты требуется 5 минут.
Имеется 2 сковороды, на них надо как можно скорее поджарить 3 котлеты.
Надо предложить и ясно записать масштабируемое решение.

Таблица 34

Программа обжарки трёх котлет на двух сковородах

Фрагмент	*Комментарий*
Обж_3_2 = [ ] {	\\ *без барьеров*
сковороды = [1..2 :] ;	\\ *Имеется 2 пустых сковороды для жарки*
Обж = Функция	\\ *оба параметра - очереди*
(Жарить Готово)	\\ = *((2 2 2) ())*
[КРОМЕ Жарить	\\ *рекурсия работает*
ТО РЕЗУЛЬТАТ готово **,**	\\ *до опустошения очереди*
ЕСЛИ Жарить ТО **(**	\\ *в вектор из очереди переданы*
жарить → сковороды ;	\\ *два элемента*
	\\ *переданные элементы исчезают из очереди*
ЖДУ 5 ;	\\ *Время «жарки»*
сковороды = (сковороды **-** 1) ;	\\ *Одна сторона обжарена*
(сковороды ?≠0) → жарить **;**	\\ *Разделяем котлеты на готовые*
(сковороды ?=0) → готово **;**	\\ *и требующие жарки*
Обж (жарить **,** готово)	\\ *«Недожаренные» идут в очередь на жарку*
)	\\ *«Готовые» - в тарелку (со 2-го витка)*
]	\\ *Надо обжарить 2 стороны 3-х котлет.*
Обж ((2 2 2) ( )) **}**	\\ *Дана пустая тарелка для готовых котлет*

Скобки можно опускать, если это не мешает распознаванию выражения.

Таблица 35

Приведение решения к массштабируемой форме

На языке СИНХРО	*Комментарий*	*Пояснение*
ЦИКЛ ОЧЕРЕДЬ жарить = (2 2 2), готово = ()	% 3 котлеты на 2 сковороды *% обжарить 2 стороны каждой котлеты* *% тарелка для готовых котлет*	Можно менять размер очереди и вектора.
ВЕКТОР сковороды [1..2] ПОКА жарить ПОВТОР жарить → сковороды ЖДУ 5 сковороды = (сковороды - 1)	*% общая вертикаль выделяет блок* *% два первых из очереди исчезают* *% время «жарки» 5 минут* *% внутренний цикл:* ск [i] = ск [i] - 1, *% одна сторона обжарена*	Используются пересылки элементов и фильтры
сковороды (?≠ 0 , ?= 0) → (жарить, готово)	*% «недожаренные» (= 1) в очередь* *% накопление результата (со 2-го витка)* *% разделили готовые и недожаренные*	
РЕЗУЛЬТАТ готово	*% = (0 0 0)*	

Пример 26. Вариант программы «масштабируемой» жарки котлет.

То же самое в виде табличной формы выглядит более ясно, чётко показывает взаимосвязи переменных с понятным побочным эффектом, естественным в реальном мире.

Таблица 36

Таблично-двумерное представление программы Примера25

Представление программы				*Комментарии*
ЦИКЛ	готово = ()	жарить = (2 2 2)	сковороды [1..2]	*% обжарить по 2 стороны котлеты* *% имеется тарелка для готовых котлет* *% и две сковороды*
ПОКА		жарить		% условие завершения - пока есть что жарить
ПОВТОР		жарить → сковороды		*% два первых элемента из очереди на жарку* *% исчезают, перемещаются в сковороды*
	ЖДУ 5			*% время «жарки» одной стороны котлеты*
			сковороды = (сковороды - 1)	*% внутренний цикл:* ск [i] = ск [i] - 1, *% значит, что одна сторона обжарена*
	сковороды (?≠ 0 , ?= 0) → (жарить, готово)			*% «недожаренные» (= 1) идут в очередь* жарки *% накопление в готово (со 2-го витка)* *% разделили готовые и недожаренные (фильтр)*
РЕЗУЛЬТАТ	**готово**			*%* готово *= (0 0 0)* *% очередь исчерпана*

Табличная форма удобна для представления нагрузки отлаженной схемы управления процессом дополнительными действиями. Например, можно в таблице, подобной (гомоиконной) алгоритмической схеме, расположить дополнительные действия для прорисовки игры про жарку котлет из примера .

Таблица 37

Размещение речевых пояснений в гомоиконной таблице

<table>
<tr><td></td><td colspan="3">Действия по сценарию</td><td rowspan="10">U</td><td colspan="3">Прорисовка, сопровождающая действия.
Изображения подготовлены заранее [7]</td></tr>
<tr><td>ЦИКЛ</td><td>готово
= ()</td><td>жарить
= (2 2 2)[8]</td><td>сковороды
[1..2]</td><td>Пустая тарелка для готовых катлет</td><td>Лоток для сырых котлет.
На нём размещены сырые котлеты</td><td>Овальная сковорода ровно на две котлеты</td></tr>
<tr><td>ПОКА</td><td></td><td>жарить</td><td></td><td></td><td>пока лоток не пуст,
есть что жарить</td><td></td></tr>
<tr><td rowspan="4">ПОВТОР</td><td></td><td colspan="2">жарить → сковороды</td><td></td><td colspan="2">Из лотка котлеты перемещаются на сковороду, оставшиеся сдвигаются на край, чтобы потом раньше попасть на сковороду.</td></tr>
<tr><td colspan="3">ЖДУ 5</td><td colspan="3">время «жарки» одной стороны котлеты</td></tr>
<tr><td></td><td></td><td>сковороды
= (сковороды
-1)</td><td></td><td></td><td>На сковороде сырые котлеты замещаются полу-обжаренными.</td></tr>
<tr><td colspan="3">сковороды (?≠ 0 , ?= 0)
→ (жарить,
готово)</td><td colspan="3">Со сковороды котлеты сортируются на готовые и полу-обжаренные. Полу-обжаренные — на лоток вслед за уже находящимися там. Готовые размещаются в тарелке.</td></tr>
<tr><td>РЕЗУЛЬТАТ</td><td colspan="3">готово</td><td colspan="3">Все готовые котлеты на тарелке
готово = (0 0 0)</td></tr>
</table>

7 Имеются разные изображения для сырых, полу-обжаренных и готовых котлет.

8 2 — сырая котлета, 1 — полу-обжаренная, 0 — готовая.

Задачи из описаний разных ЯП

Пример 27. Однородные серии автоматов. Быстрая сортировка (часто приводится в описаниях языков параллельного программирования).

Таблица 38

Множественное присваивание и три фильтра.

Фрагмент	*Комментарий*
ФУНКЦИЯ qs (Data) [ЕСЛИ (size (Data) < 2 ТО Data) , КРОМЕ (size (Data) < 2 (Pivot = Data [1] ; **Low, Mid, High** = Data ((**?<, ?=, ?<**) Pivot) ; (qs (Low) || Mid || qs (High)))]	 \\ *просачивание* \\ *по списку фильтров* \\ *и структуре*

Пример 28. Ещё один пример даёт параллельная сортировка, в которой упорядочены процессоры, элементы сортируемого вектора и связи в конструкции ветвления. Это нечто вроде метода «поплавок» в параллельном варианте с вектором процессоров, упорядочивающих по два элемента. Предлагается сравнивать по 2 соседних элемента на процессор, меняя порядок элементов, где надо. Потом сдвинуть процессорам сортируемый вектор на один элемент, затем восстановить исходный порядок и т.д. пока не обнаружится, что результат сортировки достигнут, т.е. обмена больше не не происходит.

Таблица 39

На каждом процессоре по два соседних числа.

Представление программы	*Примечание*
вектор числа A [1 .. 2K], процессоры П [1 .. K]; повтор П [i из 1 .. K] !! { ? A [2*i - 1] =< A [2*i] % вариант | A [2*i - 1] ⇔ A [2*i]} % перебор вариантов слева направо ; % затем П [i из 1 .. K] :!! { ? A [2*i] >= A [2*i + 1] | A [2*i] ⇔ A [2*i +1] }) пока выполнялось (⇔)	Имеется вектор для сортировки значений и ряд процессоров. Цикл использует процессоры, способные сравнивать пары чисел. По результатам сравнения происходит обмен значениями между соседними элементами. Процессоры сдвигаются на один элемент и сравнивают новые пары. Если не возникло необходимости обмена, то цикл завершён.

Пример 29. Программа параллельной сортировки в табличной форме. Упорядочение действий из Примера 28 выражено размещением по вертикали (табл. 39). Заодно показано, что можно использовать и вложенные таблицы.

Таблица 40

Параллельная сортировка в двумерно-табличном виде.

<table>
<tr><th>Ключевые слова</th><th colspan="2">Смысл строки зависит от ключевого слова</th><th>Примечания</th></tr>
<tr><td>цикл</td><td></td><td>П [i из 1 .. K]</td><td>Пространство итерирования</td></tr>
<tr><td>повтор</td><td></td><td><table><tr><td>если</td><td>? A [2*i - 1] =< A [2*i]</td></tr><tr><td>то</td><td>A [2*i - 1] ⇔ A [2*i]</td></tr></table></td><td>вариант1 затем Действие может быть схемой</td></tr>
<tr><td>пока</td><td>выполнялось (⇔)</td><td></td><td>Выход из всего цикла</td></tr>
<tr><td>повтор</td><td></td><td><table><tr><td>если</td><td>? A [2*i] >= A [2*i + 1]</td></tr><tr><td>то</td><td>A [2*i] ⇔ A [2*i +1]</td></tr></table></td><td>вариант2</td></tr>
<tr><td>пока</td><td>выполнялось (⇔)</td><td></td><td>Пост-условие завершения цикла</td></tr>
<tr><td colspan="2">Результат</td><td>A</td><td></td></tr>
</table>

Пример 30. Однородные серии автоматов. Параллельный «пузырёк»

Таблица 41

Процессоры в роли исполнителей

Фрагмент	*Комментарий*
числа A [1 .. 2K]; процессоры П [1 .. K] ; ЦИКЛ П [1 .. K] **!!** (ФУНКЦИЯ (i) ЕСЛИ A [2*i - 1] < A [2*i] ТО A [2*i - 1] <=> A [2*i]}	*просачивание по процессорам* *Вызов исполнителей !* *Вектор процессоров вызывает:* *Функцию процессора на окрестности (2*i):* *ЕСЛИ левое число меньше (2*i)-го,* *ТО они меняются местами;*
; \\ затем ЕСЛИ A [2*i + 1] > A [2*i] ТО A [2*i] <=> A [2*i +1]) [1 .. K] \\ просачивание	*затем работа с правым от (2*i) числом:* *ЕСЛИ (2*i)-ое число меньше правого,* *ТО они меняются местами;* *просачиваемую по диапазону 1 .. K .*
ПОКА ВЫПОЛНЯЛОСЬ (<=>)	Выход при отсутствии перестановок.

8. Преобразование программ (Мета-программирование) - Трансформ

«Движенья нет, сказал мудрец брадатый»
А. С. Пушкин «Движенье»

Часть сложностей обучения практическому программированию вытекает из достаточно очевидного сдвига понятий, вызванного тем, что реально не существует возможности использовать ЯП в чистом виде. Фактически используется входной язык СП, являющийся расширенным подмножеством ЯП, дополненным тем или иным комплектом библиотечных процедур на конкретной аппаратуре. Кроме того, СП подвергают программы разным не вполне эквивалентным преобразованиям, увидеть и осознать эффект которых не всегда удаётся в учебной практике. Такие примеры для много-поточных программ могут быть связаны с распределением действий по специальным процессорам или с синхронизацией потоков действий. Всё это естественные задачи мета-программирования.

В случае много-поточных программ возможно преобразование потоков, нацеленное на сведение к однородной системе, однозначно отображаемой на заданный комплекс процессоров размещением потоков по процессам или назначением процессоров для выполнения отдельных потоков, что можно рассматривать как задачу метапрограммирования, что позволяет делать диалект Трансформ.

Для мета-программирования несколько изменяется точка зрения на программу, преобразуемую как данное. Внешне программа состоит из комментариев, имён-идентификаторов, ключевых слов, разделителей-ограничителей, парных скобок и других синтаксически различаемых категорий лексем или синтаксических конструкций, разнообразие которых учитывается при определении шаблонов для выполнения преобразований программ. Преобразования могут сохранять функциональную эквивалентность программы (оптимизация, компиляция, распараллеливание, тестирование), а могут и существенно изменять её свойства (специализация, конкретизация, создание профилей и отладочных версий) в зависимости от целей и этапа разработки программы. Подготовка преобразований может требовать доказательств, похожих на доказательства теорем.

Диалект Трансформ выглядит как надстройка над другими диалектами или мета-диалект. При его функционировании используются определения понятий конкретного диалекта, указываемого при вызове Трансформ. Такие понятия на уровне синтаксиса выглядят как «Понятие.Диалект», где «Понятие» обозначение сквозного понятия, имеющего определения в разных диалектах, а «Диалект» - обозначение используемого диалекта. Можно сказать, что

Трансформ настраивается на язык представления обрабатываемой программы.

В такой «причесываемой», ориентированной на преобразования, форме программы для языка СИНХРОН, все потоки начинаются с барьеров, и общая шкала событий упорядочена так, что последовательность событий потока ей не противоречит. Можно считать, что процессоры включаются сами. Шкала событий содержит списки ожидающих потоков. Действия, выполняемые процессорами, соотнесены с их исходными потоками.

Достаточно простые преобразования сети потоков позволяют варьировать схемы потоков и многие конструкции языка программирования сводить к взаимодействию простых потоков:

(A;B) ↔ (a:A; b:B) – Расстановка — стирание барьеров.

(a:A; b:B) ↔ [[a:A, b:B], (a: ; b:)] — вынесение последовательного управления в отдельный поток — восстановление последовательности действий.

(a: A; b: B) ↔ [(a: A; b:), (b: B)] — разрез последовательности — перенос «хвоста», если нет локальной информационной зависимости между A и B.

(a: A; b: B) ↔ [a: A, b: B] — разбиение последовательности на потоки — сплющивание очереди в слой, если A и B информационно не связаны.

[a: A, b: B] ↔ { (a: A; b: B) | (b: B; a: A)} — слияние потоков в одну последовательность — вытягивание слоя в очередь. Выбор варианта требует учета последовательности барьеров в других потоках и общей шкале событий.

Критерий оптимальности — объем выполнимых вычислений.

((a: A; b: B) ; c: C) ↔ (a: A; (b: B ; c: C)) ↔ (a: A; b: B ; c: C)

[[a: A, b: B] , c: C] ↔ [a: A, [b: B , c: C]] ↔ [a: A, b: B, c: C]

([A,B];C) ↔ [(A;C), (B;C)] — исключение слияний — слияние совпадающих продолжений (суффиксов или пост-фиксов).

[(A;B),C]↔ ([A,C]; [B,C]) — распределение параллельного потока — слияние совпадающих потоков

[[a: A], [a: B]]↔ [a: [A, B]] — варьирование числа одновременных потоков.

Обратимость преобразований и чувствительность их результата к информационным связям между фрагментами программы требуют формализации критериев применимости трансформаций и выбора подходящего

варианта.

Можно констатировать, что выяснение информационной связанности очередей B и A сводится к проверке существования контекста, в котором различны результаты содержащей их программы C при изменении порядка вычислений B и A.

$$[(A; B), C] \neq [(B; A), C]$$

В качестве примера рассмотрим варианты распараллеливающих преобразований, позволяющих из числа базовых средств вывести ветвления и циклы, реализуя их средствами декомпозиции программ и синхронизации потоков. Определение преобразования состоит из двух шаблонов: исходного и результирующего. Аргументом преобразования является вся программы, в тексте которой выделяются фрагменты, соответствующие первому, исходному шаблоны, заменяемому на результат подстановки во второй, результирующий шаблёб+он значений переменных, упомянутых в исходном шаблоне.

ПРЕОБРАЗОВАНИЕ ЕслиПар =
(**Если** УсловиеА **то** ВырВ **иначе** ВырС)
↔ [[**ЕСЛИ** УсловиеА **ТО** ВырВ], [**КРОМЕ** УсловиеА **ТО** ВырС]]
// распараллеливание ветвлений

ПРЕОБРАЗОВАНИЕ ЦиклРек =
(**ЦИКЛ** Действие1 **ПОВТОР** Действие2)
↔ [[Действие1 (**РОБОТ** Исп = [Действие2 Исп]
// сведение бесконечного цикла к рекурсии

ПРЕОБРАЗОВАНИЕ ЦиклИтер =
(**ЦИКЛ** Действие [**ПОКА** X **ИЗ** (A1 … AK)] **ПОВТОР** (Функция X)
// пространство итерирования
// при выборе X элементы исчезают из перечня
//как при переборе множеств
↔ (**ОЧЕРЕДЬ** Действие; *// инициирующее действие*
[ВМЕСТЕ (Функция A1**), (**Функция A2**), … , (**Функция AK**))])**
// независимые потоки для каждого элемента пространства

Другая категория задач связана с переводом программ из одной формы в другую.

Слой = **[** Дир ([**,**] Дир) ...**]** Row = **[** Com ([**,**] Com) ...**]** действие = Дир обход барьер цикл назначение дир фрагмент	→ (**Параллель** Дир Дир …) = (**Row** Com Com …)
Поток = **(** Дир ([**;**] Дир) ... **)** Line = **(** Дир ([**;**] Дир) ... **)**	→ (**Поток** Дир Дир …) = (**Line** Com Com …)

| Выр [Фильтр] → структура \\ *пересылка по фильтру* = Expr [Filter] → Structure	→ (**Перенос** Выр Структура Фильтр) = (**Move** Expr Structure Filter)

Действие = [РЕЗУЛЬТАТ] Выр \\ *вычисление результата* Com = [RESULT] Expr	→ Выр \\ *просто вычисление* → (Результат Выр) = (**Value** Expr)
| Имя **=** Выр \\ - *именование* = Name = Expr	→ (**Ссылка** Имя Выр) = (**Access** Name Expr)
| Имя <=> Имя \\ - *обмен значениями*	
| Выр [Фильтр] → структура \\ *пересылка по фильтру* = Expr [Filter] → Structure	→ (**Перенос** Выр Структура Фильтр) = (**Move** Expr Structure Filter)

| Имя = [ФУНКЦИЯ] **(** Параметр **…)** Фрагмент \\ *объявление* = Name = [FUNCTION] **(** Parameter **…)** Com	→ (**Опред** Имя (Параметр …) Дир) = (**Define** Name **(** Parameter **…)** Com)
| Барьер **:** Фрагмент \\ *абсолютная синхронизация потоков* = Barrier **:** Com | Барьер _**:** Фрагмент \\ *индексированная синхронизация* = Barrier _**:** Com	→ (**Барьер** Барьер) Дир = (Barrier **)** Com Синтаксически типизированная Макроподстановка Индексы при копировании фрагмента
| ЕСЛИ Выр ТО Фрагмент \\ *обход по условию – управление* = IF Expr THEN Com | ЕСЛИ_НЕ Выр ТО Дир \\ *по невыполнению* = IF_NO Expr THEN Com	→ (**Если** Выр Дир) = (**IF** Expr Com) → (**КРОМЕ** Выр Дир) = (**Exept** Expr Com)
| [ВЕРОЯТНО] Выр **%** Дир \\ *обход по вероятности* [9] = [PROBABLY] Expr **%** Com	→ (**Вероятно** Выр Дир) = (**Probably** Expr Com)

При отладке формируется набор контекстов, демонстрирующих конкретные свойства фрагментов, из которых собирается полная программа. Это

9 *Вычисление вероятности можно выделить как отдельный автомат*

контексты для отдельных потоков, для пар синхронизованных потоков, для интегрированной из потоков программы, а кроме того, контексты для удостоверения наличия-отсутствия информационных связей между фрагментами. Возможны **пользовательские** преобразования схем управления процессами, что позволит не только минимизировать ручную аранжировку распараллеливаемых программ, но и даст основу для формирования библиотек преобразования схем программ. Всё это задачи подъязыка Трансформ.

Потоки могут содержать действия, выполняемые на разных специальных процессорах. При необходимости обмена результатами процессоры могут использовать общую память. Пусть программа представляет собой неупорядоченный ряд потоков, использующих сторонние процессоры Пр1 и Пр2:

```
[ряд (поток Д1; Пр1 # Д2; Д3),
      (поток Д4; Пр2 # Д5; Пр1 # Д6) ]
```

Такой ряд пополняется отдельными потоками, выполняемыми процессорами Пр1 и Пр2, что можно выразить формулой:

```
→ [ряд (поток Д1; сигнал(Пр1) ; Д3) ,
        (поток Д4; сигнал(Пр2) ; сигнал(Пр1) ; Д6 ) ,
      Пр1 # ( ряд [ Д2 , Д6 ] ) ,
      Пр2 # Д5  ]
```

Переход от одной формы к другой можно представить таблицей 9.

Технически сложнее выглядят преобразования потоков, синхронизованных с помощью барьеров. Пусть программа представляет собой неупорядоченный набор потоков, использующих синхронизацию с помощью барьера «Бар»:

```
[набор (поток Д1; Бар: Д2; Д3),
        (поток Д4; Д5; Бар: Д6) ]
```

Такую программу можно свести к потоку двух неупорядоченных наборов, первый из них содержит действия, выполняемые до барьера, второй — после барьера:

```
→ (поток [набор (поток Д1; ) , (поток Д4; Д5 )];
      Бар: [набор (поток Д2; Д3 ) , (поток Д6 )] )
```

Необходимые преобразования можно выразить таблицами 42, 43.

Таблица 42

Преобразование программы, использующей сторонние процессоры.

Расширенное представление			→	*Приведение к базису*					*Примечания*
набор	поток	поток	→	набор	поток	поток	Пр1	Пр2	Добавляются 2 потока-процессора
	Д1	Д4	→		Д1	Д4	ряд		Д2 и Д4 независимы
	Пр1 # Д2	Пр2 # Д5	→		(сигнал Пр1)	(сигнал Пр2)	Д2 Д6	Д5	На их месте вставка сигналов про обращение к сторонним процессорам
	Д3	Пр1 # Д6	→		Д3	(сигнал Пр1)			Порядок оставшихся действий не нарушается

Таблица 43

Приведение динамической синхронизации потоков действий барьерами к статической серии с неупорядоченными рядами потоков.

Синхронизация потоков барьерами			→	*Приведение к каскаду наборов*			*Примечания*
				поток			Набор погружается в поток
набор	поток	поток	→	набор	поток	поток	
	Д1;	Д4;	→		Д1	Д4	Действия **до** барьера
	Бар: Д2;	Д5;	→			Д5	
	Д3	Бар: Д6			Бар:		Барьер
Результат			→	набор	поток	поток	Выделение хвостов потоков
			→		Д2	Д6	Действия **после** барьера
			→		Д3		
				Результат			

При необходимости можно таким образом выразить включение в программу базовых средств низкого уровня и видеть возможности оптимального распараллеливания программы на уровне подъязыка АПК.

ЗАКЛЮЧЕНИЕ

Рассматривая задачу использования учебных языков параллельного программирования как путь к решению непредсказуемых проблем [58] адаптации методов вычислений к различным особенностям используемых многопроцессорных комплексов и многоядерных процессоров, следует видеть, что решение этой проблемы, кроме ознакомления с проблемами и методами, требует разработки новых методов реализации программ с акцентом на тестирование, верификацию и отладку с возможностью принятия самостоятельных программистских решений [59, 60], а также развития средств и методов ясного описания семантики языков параллельного программирования, включая представление программируемых преобразований текста и переносимого кода программы с удостоверением их корректности [61].

Предлагаемый язык СИНХРОН представляет собой эксперимент по выбору базовых средств для достаточно полного решения проблем эффективной реализации параллельных алгоритмов, вынужденно требующих использовать весьма широкий спектр сложно совместимых средств: от управляющих действий более низкого уровня, чем в привычных языках программирования, до манипулирования пространствами решений по обработке данных в памяти, типичных для языков сверхвысокого уровня. Обзор исследований и решений в смежных областях представлен в [5, 6].

Особый круг образовательных проблем связан с навыками учёта особенностей многоуровневой памяти в многопроцессорных системах. Обычное программирование такие проблемы может не замечать, полагаясь на решения компилятора, располагающего статической информацией о типах используемых данных и способного при необходимости выполнить оптимизирующие преобразования программы. Новые и долгоживущие языки программирования, как правило, имеют мультипарадигмальный характер, что приводит к идее их расширения на многие модели параллельных вычислений.

Новые языки программирования содержат достаточно разнообразные конструкции для представления программ параллельных вычислений. Нужна система обучения, приспособленная не только к ознакомлению с отдельными эффектами и средствами параллельных вычислений, но и поддерживающая осознание особенностей взаимодействия процессов, удобная для экспериментов по формированию интуитивных навыков подготовки работоспособных программ, понимания новых возможностей аппаратуры и выполнение верификации программ [65-67]. Не исключено, что этому противостоит исторически сложившаяся традиция одномерного, линейного представления

текстов программ в виде бесконечной строки. В данной статье рассмотрен ряд форм, которые могут быть полезны при решении учебных задач по организации взаимодействующих параллельных процессов, включая процессы над общей памятью, на уровне решений, предложенных в языке учебного программирования СИНХРОН.

Благодарности. *Автор благодарен студентам ММФ и ФИТ НГУ, выполнивших в разные годы реализацию прототипов основных модулей для разработки языка СИНХРОН, что позволило сформулировать выбор решений и отладить прагматику языка. Сергей Коваль, Агамиров, Любовь Ефимова, Алена Боярских, Дмитрий Мажуга, Никита Шаченко, Сергей Демидов, Марк Бычков и другие.*

СПИСОК ЛИТЕРАТУРЫ

1. Магариу Н. А. Язык программирования АПЛ. — М.: «Радио и связь», 1983. — 96 с.
2. Городняя Л.В. Язык параллельного программирования Синхро, предназначенный для обучения. - Новосибирск, 2016.-30 с. (Препринт/ИСИ СО РАН; N 180). URL: https://www.iis.nsk.su/files/preprints/gorodnyaya_180.pdf
3. Воеводин В. В., Воеводин Вл. В. Параллельные вычисления. — СПб.: БХВ- Петербург, 2002. — 608 с.
4. Городняя Л.В. Парадигмы программирования. Часть 4. Параллельное программирование. — Новосибирск, 2015. — 74 с. — (Препр. / ИСИ СО РАН; № 175). — URL: http://www.iis.nsk.su/files/preprints/gorodnyaya_175.pdf.
5. Городняя Л.В. О неявной мультипарадигмальности параллельного программирования //Материал конференции: "Научный сервис в сети Интернет: труды XXIII Всероссийской научной конференции с. 104-116 https://library.keldysh.ru/prep_vw.asp?pid=9203
6. Городняя Л.В. Модели работы с памятью в учебном языке программирования СИНХРО //Научный сервис в сети Интернет: труды XXIV Всероссийской научной конференции (19-22 сентября 2022 г., онлайн). — М.: ИПМ им. М.В.Келдыша, 2022. — С. 137-154. https://doi.org/10.20948/abrau-2022-1 https://keldysh.ru/abrau/2022/theses/1.pdf
7. Городняя Л.В. Работа с данными в учебном языке программирования СИНХРО с. 87-97.
8. Городняя Л.В. Методика парадигмального анализа языков и систем программирования //Научный сервис в сети Интернет: труды XXI Всероссийской научной конференции (23-28 сентября 2019 г., г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2019. — С. 262-277. — URL: http://keldysh.ru/abrau/2019/theses/03.pdf doi:10.20948/abrau-2019-03
9. Андреева Т.А. и др. Компьютерные языки как форма и средство представления, порождения и анализа научных и профессиональных знаний. // Тр. XV Всерос. научно-методической конф. «Телематика-2008». — СПб., 2008. − С. 77–78.
10. Городняя Л.В. О курсе «Начала параллелизм» // Ершовская конференция по информатике. Секция «Информатика образования». — Новосибирск: ИСИ СО РАН, 2011. — С. 51–54.

11. Городняя Л.В. Парадигмы программирования. Ч. 5. Учебные языки программирования. — Новосибирск, 2015. — 60 с. — (Препр. / ИСИ СО РАН; № 176). — URL: http://www.iis.nsk.su/files/preprints/gorodnyaya_176.pdf
12. Городняя Л.В. О проблеме автоматизации параллельного программирования // Тр. Междунар. суперкомпьютерной конф. «Научный сервис в сети Интернет: многообразие суперкомпьютерных миров» — URL: http://agora.guru.ru/abrau2014 .
13. А.П. Ершов Программирование - вторая грамотность http://ershov.iis.nsk.su/ru/second_literacy/article
14. Кушниренко А.Г., Лебедев Г.В., Сворень Р.А. Основы информатики и вычислительной техники: Пробный учебник для средних учебных заведений. Учебное издание. Москва: Издательство «Просвещение», 1990.
15. Кушниренко Анатолий Георгиевич, Грибанова Ирина Николаевна, Райко Миля Вячеславовна, Зайдельман Яков Наумович Как мы проводим вводное занятие по алгоритмике в разновозрастной группе дошкольников и младших школьников *https://fgoskomplekt.ru/upload/iblock/a2c/4o0ce5xyjyrha1b8qwsgxlhnjqp1iqme.pdf*
16. О logo // https://web.archive.org/web/20041013130519/http://www.int-edu.ru/logo/logo.html
17. Г*оликов Д. В., Голиков А. Д.* Книга юных программистов на Scratch. — SmashWords, 2013. — ISBN 978-1310227554. (Вертун)
18. Питер ван Рой https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng108.pdf
19. Seif Haridi and Nils Franzén Tutorial of Oz http://mozart2.org/mozart-v1/doc-1.4.0/tutorial/
20. Ф. Брукс. Мифический человеко-месяц, или Как создаются программные системы. СПб.: Символ-Плюс, 1999
21. Хоар Ч. Взаимодействующие последовательные процессы. — М.: Мир, 1989,. — 264 с.
22. Вальковский В.А., Малышкин В.Э. Синтез параллельных программ и систем на вычислительных моделях. Издательство "Наука", Сибирское отд-е, Новосибирск, 1988г., 129 с.
23. https://parallel.ru/tech/tech_dev/par_lang.html
24. Городняя Л.В. Парадигмы программирования. Ч.4 Параллельное программирование 73 с. (Препр. / ИСИ СО РАН; № 175). — URL: https://www.iis.nsk.su/files/preprints/gorodnyaya_175.pdf)

25. Г. Майерс Надежность программного обеспечения М.: «Мир», 1980. 360 с.
26. Майерс Г Искусство тестирования программ М.:Финансы и статистика, 1982, 176 с.
27. В. В. Кулямин Методы верификации программного обеспечения https://www.ispras.ru/publications/methods_of_software_verification.pdf
28. Резникова Ж. И. Интеллект и язык животных и человека. Основы когнитивной этологии. — М.: Академкнига, 2005. — 518 с.
29. Городняя Л.В. От трудно решаемых проблем к парадигмам программирования //Информационные и математические технологии в науке и управлении Выпуск №1(21) / 2021 с. 94-109 https://www.imt-journal.ru/archive/public/article?id=183
30. Городняя Л.В. Перспективно-стратегические парадигмы программирования академика Андрея Петровича Ершова // V международная конференция «Развитие вычислительной техники в России, странах бывшего СССР и СЭВ» Россия, Москва, НИУ ВШЭ, 6–8 октября 2020 года https://www.sorucom.org/articles/materialy-mezhdunarodnoy-konferentsii-sorucom-2020/4540/
31. Вальковский В.А., Котов В.Е., Марчук А.Г., Миренков Н.Н. Элементы параллельного программирования
32. Котов, В.Е. Параллельное программирование с типами управления [Текст] / В.Е. Котов // Кибернетика. – 1979. – No 3. – С. 1–13
33. Т.И.Лельчук, А.Г.Марчук. Поляр язык параллельного асинхронного программирования.- Программирование, $ 4, 1983, с. 59-68.
34. Бульонков, М.А. Базовый язык параллельного программирования Барс [Текст] / М.А. Бульонков, А.В. Быстров, Н.Н. Дудоров, В.Е. Котов // Программирование. – 1986. – No 6. – С. 32–40.
35. А.В.Быстров, Н.Н.Дудоров, В.Е.Котов. О базовом языке.- Сб.: Языки и системы программирования, Новосибирск, 1979, с. 85-106.
36. Евстигнеев В.А., Городняя Л.В., Густокашина Ю.В. "Язык функционального программирования SISAL", // Интеллектуализация и качество программного обеспечения, Новосибирск, 1994, с. 21-42
37. Cann D. C. SISAL 1.2: A Brief Introduction and tutorial. Preprint UCRL-MA-110620. Lawrence Livermore National Lab., Livermore – California, May, 1992. – 128 p.
38. Звенигородский Г.А. Первые уроки программирования. — Библиотечка «Квант». — М.: Наука, 1985. — Т. 41. http://cip.iis.nsk.su/files/course/zven-ves.pdf. (Робик)

39. В.Э. Малышкин Технология фрагментированного программирования http://omega.sp.susu.ru/books/conference/PaVT2012/short/212.pdf
40. Перепелкин В.А. Система LuNA автоматического конструирования параллельных программ численного моделирования на мультикомпьютерах // Проблемы информатики. No 1(46). ИВМиМГ СО РАН (Новосибирск), 2020. с. 66-74. DOI: 10.24411/2073-0667-2020-10004
41. Малышкин В.Э., Перепёлкин В.А., Щукин Г.А. Распределённый алгоритм управления данными в системе фрагментированного программирования LuNA // Проблемы информатики. No 1(34). 2017. с. 78-88
42. Малышкин, В.Э. Технология фрагментированного программирования [Текст] / В.Э. Малышкин // Вестник ЮУрГУ. Серия «Вычислительная математика и информатика». – 2012. – No 46. –С.45-55
43. Лавров С.С. Расширяемость языков. Подходы и практика. В сб.: Прикладная информатика, вып. 2. М.: Финансы и статистика, 1984, с. 17 - 22.
44. Хендерсон П. Функциональное программирование. Применение и реализация: Пер. с англ.—М.: Мир, 1983.—349 с.
45. А. Филд, П. Харрисон. **Функциональное программирование.** (Functional Programming) Авторы: А. Филд, П. Харрисон. Перевод с английского М.В. Горбатовой, А.А. Рябинина, В.Л. Торхова, М.В. Федорова под редакцией В.А. Горбатова. Научное издание. (Москва: Издательство «Мир». Редакция литературы по информатике, 1993)
46. Адамович А.И., Климов Анд.В. Об опыте использования среды метапрограммирования Eclipse/TMF для конструирования специализированных языков // Научный сервис в сети Интернет: труды XVIII Всероссийской научной конференции (19-24 сентября 2016 г. г. Новороссийск). — М.: ИПМ им. М.В.Келдыша, 2016. — С. 3-8. — doi:10.20948/abrau-2016-45
47. Schwartz, Jacob T., "Set Theory as a Language for Program Specification and Programming". – Courant Institute of Mathematical Sciences, New York University, 1970. SETL (1972 J.T.Schwartz)
48. About SETL and GNU SETL. https://setl.org/doc/setl.html
49. Л.В. Городняя Подходы к представлению синтаксиса языков программирования Препринт 185 https://www.iis.nsk.su/files/preprints/Preprint_185.pdf
50. Б. Я. Штейнберг, О. Б. Штейнберг, Преобразования программ

фундаментальная основа создания оптимизирующих распараллеливающих компиляторов, Программные системы: теория и приложения, 2021, том 12, выпуск 1, 21–113
51. Л.В.Городняя Абстрактная машина языка программирования учебного назначения СИНХРО
52. https://cyberleninka.ru/article/n/abstraktnaya-mashina-yazyka-programmirovaniya-uchebnogo-naznacheniya-sinhro?ysclid=m6vpjzj0f9627267882
53. https://earthz.ru/solves~3~t350 — Задачи по математике
54. Быстров А.В. Сетевые средства синхронизации процессов // Тр. Всесоюзного научно-технического семинара "Программное обеспечение многопроцессорных систем". Калинин, 1985. - С. 44-46.
55. Котов Сети Петри М.: Наука, Главная редакция физико-математической литературы, 1984 — 160 с.
56. Котов, В.Е. Алгебра регулярных сетей Петри [Текст] / В.Е. Котов // Кибернетика. – 1980. – No 5. – С. 10–18.
57. Канеман Д. Думай медленно... решай быстро. (Thinking, Fast and Slow) — М.: АСТ, 2013. — 625 с.
58. алеб, Нассим Николас (2015), Черный лебедь. Под знаком непредсказуемости, КоЛибри, ISBN 978-5-389-09894-7
59. Бентли Дж. "Жемчужины программирования" (2-е издание) //Питер, 2002, 272 с.
60. Алан Купер Психбольница в руках пациентов. Алан Купер об интерфейсах // Питер, 2023 , 384 с.
61. Демидов С.Е. Создание интерфейса оболочки обучающих параллелизму роботов. 56-я Международная научная студенческая конференция «Студент и научно-технический прогресс», 2018, с. 72-72
62. Шаченко Н.В. Проектирование и разработка ядра обучающей параллелизму системы роботов. 56-я Международная научная студенческая конференция «Студент и научно-технический прогресс», 2018, с. 85-85
63. Л. В. Городняя , А. С. Боярских О проблеме начального обучения параллельному программированию. Томск 2009 . с. 20-24. http://conference.tsu.ru//pvv/&conference_page=text&pageid=115
64. Городняя Л.В. Об одном подходе к синтезу транслятора на примере языка Литтл. // Теория и практика системного программирования. − Новосибирск, 1977. − С. 60-71.
65. Кондратьев Д.А., Промский А.В. Разработка самоприменимой системы верификации. Теория и практика // Моделирование и анализ

информационных систем. —2014. — Том 21. — № 6. — С. 71–82. — DOI: https://doi.org/10.18255/1818-1015-2014-6-71-82
66. Касьянов В.Н., Касьянова Е.В. Язык программирования Cloud Sisal. – Новосибирск, 2018. – 55 с. – (Препр. / ИСИ СО РАН; № 181).
67. Касьянов В.Н., Гордеев Д.С., Золотухин Т.А., Касьянова Е.В., Кондратьев Д.А. Система облачного параллельного программирования CPPS: визуализация и верификация Cloud Sisal программ : моногр. / Под ред. В.Н. Касьянова ; Ин-т систем информатики им. А.П. Ершова СО РАН. – Новосибирск: ИПЦ НГУ, 2020. – 256 с. – (Сер.: Конструирование и оптимизация программ; Вып. 22). DOI: https://doi.org/10.31144/978-5-4437-1123-2.pp.256

Приложение 1

АПК: Система команд ядра языка СИНХРО (прагматика)

Суммарно комплект команд включает в себя действия, часть из которых определены в книге П. Хендерсона (LD, LDC, LDF, AP, RTN, RAP, DUM, SEL, JOIN, CONS, **CAR, CDR, CONS,** ATOM, EQ, SUB, ADD, MUL, DIV, STOP) [7],

LD – ввод данного из локального контекста в стек;
LDC – ввод константы из программы в стек;
LDF – ввод определения функции в стек;
LDM – ввод данного из общей памяти в стек;
COMPSUB – компиляция «на лету» фрагмента программы, заданного в виде текста;
AP – применение функции, определение которой уже в стеке;
RTN – возврат из определения функции к вызвавшей ее программе;
RAP – применение рекурсивной функции;
DUM – резервирование памяти для хранения поколений аргументов рекурсивной функ-
ции или цикла;
SEL – обход участка или ветвление в зависимости от активного (верхнего) значения стека;
JOIN – переход к общей точке после обхода;
CRCL – цикл – повтор;
BR – принудительное завершение выполнения цикла, комплекта, потока или функции;
PAIR – CONS – формирование узла по двум верхним значениям стека для списков;
HEAD – CAR – первый элемент узла из активного значения стека;
TAIL – CDR – остаток узла без первого элемента активного значения стека;
ARR – формирование вектора из заданного числа элементов, расположенных в стеке;
ELM – выбор элемента из вектора по заданному индексу;

ATOM – неделимость (атомарность) верхнего элемента стека;
NUMBER – проверка на число;
LIST – проверка на список;
ARRAY – проверка на вектор;
TEXT – проверка на текст или строку;
EQ – равенство двух верхних значений стека с учетом произвольного значения;
STOP – останов;
SET – запись в общую память комплекса из стека;
LET – сохранение локальных значений в контексте;
MLL – пересылка головного элемента из списка в список головного элемента;
MLV – пересылка из списка головного элемента в указанный элемент вектора;
MVL – пересылка из указанного элемента вектора в головной элемент списка;
MVV – пересылка из указанного элемента вектора в указанный элемент другого вектора;
CHNG – обмен данными в общей памяти комплекса;
DELETE – удаление верхнего элемента стека;
WRT – вывод данных из стека в процесс внешнего периферийного устройства с сигналом
успеха-провала;
RD – ввод данных от процесса внешнего периферийного устройства в стек с сигналом
успеха-провала;
ANY – размещение в стеке выравнивающего данного, если данное с устройства не введено;
KIT – комплекс из процессоров или набор процессов выполнения действий-директив, мощность комплекса или набора известна;
ROW – процесс для выполнения очереди действий-директив на одном процессоре, длина очереди известна в каждый момент;
REZ – размещение заданного числа результатов процесса в свой стек;
WAIT – ожидание приостановки указанного процесса (завершения или сообщения);
SEND – передача сообщения указанному процессу;
NEXT – ожидание завершения предыдущего действия;

NUMBER – выяснение, является ли данное числом;
INC – увеличение числа на 1;
DEC – уменьшение числа на 1;
SUB – вычитание из верхнего элемента стека;
ADD – прибавление к верхнему элементу стека;
MUL – произведение двух чисел, первое в стеке;
DIV – частное от деления верхнего элемента стека на второй элемент;
LT – выяснение, верхнее значение в стеке меньше ли, чем второе;
ERR – реакция на ошибку, аппаратную или программируемую, если в коде программы
она встроена.
Технические детали определения этих и остальных команд приведены в статье [21].

Команды-запросы обычных процессоров, на которых выполняются активные процессы:
GIVE – запрос регистра для переменной с номером H;
SAFE – запрос на хранение данного в переменной по регистру H;
READ – запрос на чтение данного из переменной по регистру H;
UNDO – запрос на получение предыдущего данного из переменной по адресу H;
FREE – объявление, что не нужна больше переменная по регистру H.

Команды-двойники процессора общей памяти, пассивно ждущего запросов от активных
процессов или отладчика:
TAKE – выбор регистра для переменной с указанным номером H, счетчик кратности доступа к нему его надо увеличить на 1;
WRITE – размещение данного в регистре H. Если данное – указатель на вектор или список, то они копируются полностью в общую память. В протокол вносится копия указателя на
данного и номер активного процесса;
VAR – чтение данного из регистра H для передачи обычному процессору –

состояние регистра H не меняется;
UNDO – чтение предыдущего данного из регистра H – состояние регистра H не меняется;
FREE – освобождение памяти, доступной через регистр H, – счетчик кратности доступа уменьшается на 1.

Таблица 44

Дублеты - парные команды, срабатывают только неразрывно вместе

Активные команды локальных процессоров	**Пассивные команды-двойники процессора ждущие запросов от локальных про**
GIVE – запрос регистра для переменной в общей памяти	**TAKE –** выбор регистра для переменной, счётч доступа к нему его надо увеличить на 1.
SAFE – запрос на хранение данного в переменной в общей памяти	**WRITE** – размещение данного. В протокол вно на данное и номер процесса.
READ – запрос на чтение данного	**VAR –** доступ процесса к данному из общей па
UNDO – запрос на получение предыдущего значения переменной	**UNDO –** доступ к предыдущему значению пере состояние регистров переменной не меняется.
FREE – объявление, что больше переменная не нужна.	**FREE** – освобождение памяти, счётчик кратнос уменьшается на 1.

Для локальных процессоров возможны запросы на регистр для переменной, на хранение данного в переменной по её регистру, на чтение данного из переменной по регистру, на получение предыдущего данного из переменной по адресу и на объявление, что больше переменная не нужна.

Таблица 45

Дублеты – парные команды – срабатывают только неразрывно вместе.

Запросы из активных процессоров	*Двойники в процессоре общей памяти*	*Неразделимая пара*	*Примечание*
GIVE	TAKE	GIVE-TAKE	Дай-Бери: Запрос на пере
SAFE	WRITE	SAFE-WRITE	Храни-Пишу: Запись знач
READ	VAR	READ-VAR	Читаю-Доступ: Запрос зн
UNDO	UNDO	UNDO-UNDO	Верни-Возврат: Восстано
FREE	FREE	FREE-FREE	Освободи-Свободен: Осв

Сохранение значения из стека в глобальной памяти:

T – номер текущего процесса, задавшего запрос на использование или

Вид программы для АПК

Программа ::= Контекст Набор ?S *// вывод содержимого стека*

Контекст ::= Регистры

Процессоры *// один по умолчанию, остальные порождаются*

Набор ::= Процессы

Процессы ::= (Элемент / « ») *// перечень элементов через пробел*

Элемент ::= Команда *// из списка команд АПК*

Константа *// имя переменной или число или строка в кавычках*

Регистры ::= *S* | E | C | D | R | M | V | |

// S – стек текущих результатов вычислений

// E – локальный контекст

// C – текущий процесс/поток

// D – dump - резервная память

// R – регистр недетерминированных вариантов

// M – общая память, пассивный процессор

// V – паспорт доступа к общей памяти

Приложение 2

Синтаксис учебного языка программирования СИНХРО

Самоописание БНФ

Мета-знак ::= «**|**» | «**::=**» | «**"**» | **<Пусто>**

// Разделитель вариантов, частей правила, ограничителя символов
*// и **Пустая строка***

*// **Мета-знаки выделены шрифтом** **arial black***

Правило **::=** Понятие **"**::=**"** Цепочка
// обозначение и его замещающее определение

Понятие **::= Имя** *// Обозначение нетерминала, имеющего определение*

Цепочка **::=** Элемент Цепочка *// конечное число элементов*
| <Пусто> *// цепочка может быть пустой, учитывается последней*

Элемент **::= "**Символ**" |** Понятие **|** Лексема // составляющие определений

Символ **::=** Знак **|** ЗнакСимвол *// несколько знаков без пробелов*

Знак ::= *// любая литера*

Лексема ::= *число* | *Имя* | Ключ |/7 строка *// определяются вне языка*

Описание РБНФ на БНФ (расширенные)

Мета-знак ::= **«|» | «::=» | «"» | <Пусто> | «\»**
| «**(**» | «**)**» | «**[**» | «**]**» | «**{**» | «**}**» | «» | «» |

// Разделитель вариантов, частей правила, ограничителя символов,
// разделитель в перечне, скобки

Правило **::=** Понятие **"**::=**"** Определение
// обозначение и его замещающее определение

Понятие **::= Имя** *// Обозначение нетерминала, имеющего определение*

Определение **::=** Вариант **["|" (** Вариант **\ "|")]** *// конечное число Вариантов*
| <Пусто> *// пустой вариант выбирается последним*
// Любое число вариантов, один должен быть

Вариант **::= "**Символ**" |** Лексема **|** Понятие **|** Правило // локально однократное
// простые составляющие определений
| Группа **|** Факультатив **|** Перечень **|** Любой
// составные составляющие определений

Символ **::=** Знак **|** ЗнакСимвол *// несколько знаков без пробелов*

Знак ::= *// любая литера*

Лексема **::=** Символ **|** Ключевое_слово **| число | строка| Имя**

// имя не входит в число понятий

// определяются вне языка

// Символ без кавычек, если это не мешает разбору

Ключевое_слово **::= ТЕКСТ** *// заглавные буквы*

Группа **::= (** Элемент **\ " ")**

// конечная цепочка элементов через пробелы в скобках

Элемент **::= Лексема | Понятие** *// составляющие элементов*

Факультатив **::= [** Элемент **]** *// необязательный элемент*

Перечень **::= (_** Элемент **** Лексема **)**

// конечное число перечисленных элементов, разделённых лексемой

Любой **::= {** Лексема **\ " " }** *// одна из перечисленных через пробел лексем*

Определение ИРБНФ на РБНФ (индексируемые)

Мета-знак ::= **«|» | «::=» | «"» | <Пусто>** | **«\»** | **«...»** | **«(»** | **«)»**

| **«[»** | **«]»** | **«{»** | **«}»** | «→» | «↔» | «.» | «_»

Понятие **::= Имя** | ИндексПеременная | ПеременнаяЯП

// Обозначение сквозного понятия, имеющего определения в разных ЯП

ИндексПеременная **::=** Имя_Номер | ПеременнаяЯП_Номер

// Имя понятия дополнено номером синтаксической позиции

// для хранения значения, полученного при разборе текста

ПеременнаяЯП **::=** Имя**.**Диалект *// имя понятия из другого ЯП*

Вариант **::=** | Трансформация *// определены в диалекте Трансформ*

Многократно **::=** Группа **...**

// примерная схема трансформаций:

Трансформация **::=** Макрос *// выглядит как функция*

| Шаблон *// выглядит как структура данных*

| Изменение *// работает как замена на другую форму*

| Преобразование *// правая часть эквивалентна левой*

| Перевод *// правая часть на другом диалекте*

Изменение **::=** Шаблон_1 → Шаблон_2 *// переход к новому состоянию*

Перевод **::=** Образец_1 ↔ Образец_2 *// в другой диалект*

Образец **::=** **"**Символ**"** **|** Понятие **|** Лексема // составляющие определений

| Группа **|** Перечень *// нет факультативов и любых*

Фрагмент **::=** Символ **|** Лексема **|** Слог *// используется в макросах*

Слог **::=** Лексема Понятие *// для простой автоматизации разбора*

/ *Лексема Понятие Лексема*

Синтаксис диалекта Мульти

Мета-знак ::= «**|**» | «**::=**» | «****» | «**(**» | «**)**» | «**[**» | «**]**» | «**{**» | «**}**» | «_»

Программа ::= [Контекст **]** **[** Набор **]** **? Данное**

// Всегда выводится во внешний мир сообщение о результате

Контекст ::= ((Имя = Данное **) \ ;) ;**

| (Доска ::= Число_1*Число_2 **)** Контекст

// Обстановка и протоколы

*// **Доска** ::= Число*Число // Размеры клетчатой доски*

Данное ::= Число **|** Текст **|** Клетка **|** Перечень

// записи или координаты на доске или в специальном регистре

| Изображение **|** Исполнитель

//Рисунки доступны в общей памяти — регистр M

Клетка *// одноклеточная картинка или спрайт*

Изображение *//Комплект жёстко связанных клеток*

// для прорисовки реквизита или исполнителя

Набор ::= [Имя = **]** [**(** Поток **\ [** , **])**] *// В любом порядке*

Поток ::= [Имя = **]** (**(** Действие **\ [** ; **]** **)**) *// По очереди*

Действие ::= [Имя =**]** Команда **| @!!** Имя **| (** Действие **)**

//вызов исполнителя пополняет список команд

// подобно библиотеке, модулю или пакету

| [ВЕРОЯТНО **]** **%** Число_1/Число_2 Действие

| ЕСЛИ Условие ТО Действие

// возможен вариант КОГДА — обязательно будет

| КРОМЕ Условие ТО Действие

| ЖДУ Имя *// исполнитель рядом, вызван*

|| ПАУЗА Число *// синхронизующий интервал времени*

| ЦИКЛ Действие_1 **[** ПОКА Условие **]** ПОВТОР **[**Число **]** Действие_2

// цикл может быть бесконечным, в этом нет ничего плохого,

//это полезно при организации сервисов

Условие ::= Имя **|** Данное **|** Имя ИЗ Перечень

// выбранное данное ИЗ перечня исчезает

Команда ::= Имя **|** ШАГ **[** **@!!** Имя **]**

// очередной шаг сценария, возможно другого исполнителя

|{ ПУСК СТОП ВВЕРХ ВНИЗ ВПРАВО ВЛЕВО ПАС **}**

| [ВМЕСТЕ **(** Действие **\[** **,]**] **)]** *// в любом порядке*

| **(** ОЧЕРЕДЬ **(** Действие **\[** **;** **]** **)** **)** *// последовательно*

| ВОЗВРАТ **[** **@!!** Имя **]** *// вернуть шаг исполнителя*

Имя *— лексема, зависит от реализации*

Перечень ::= (Данное **...** **)**

// перечень произвольного числа данных через пробел

Исполнитель ::= Имя **|** Кнопка **|** АПК **|** Диалект

// абстрактная машина в качестве исполнителя

| Дежурик **|** Муравей **|** Художник **|** Математик **|** Писатель

// Звенигородский - Робик

| Торговый_Автомат *// по мотивам книги Хоара*

| **[** Изображение **]** **(** **(**Инструкция **\[** **,** **]** **)** Сценарий **)**

//безымянный исполнитель

// Изображение, *система команд и сценарий*

| Имя = Исполнитель *//именованный исполнитель*

// Исполнители автономны, общаются только через наполнение доски

Инструкция ::= Команда

// доступна после вызова исполнителя до остановки его сценария

Сценарий ::= Программа ***// запускается при вызове исполнителя***

// линейный участок с обходами, допуская самоприменение

// может быть подготолен на другом диалекте

Диалект ::= Тансформ **|** Асинхр **|** Синпар

Тансформ *// создание и преобразование исполнителей*

Асинхр *// асинхронные потоки*

Синпар *// синхронизация потоков*

Синтаксис диалекта Трансформ
(дополнение Мульти)

// Работает с программами

Мета-знак ::= «**|**» | «**::=**» | «****» | «**(**» | «**)**»
| «**[**» | «**]**» | «**{**» | «**}**» | «**'**» | «**.**» | «_»
| «⟶» | «⟷» | «**...**»

Диалект ::= Трансформ **|** Мульти **|** Асинхр **|** Синпар **|** АПК
Тансформ *// создание и преобразование исполнителей и программ*
Мульти // ознакомление с параллелизмом
Асинхр *// асинхронные потоки*
Синпар *// синхронизация потоков*
АПК *// базовая семантика — абстрактная машина*

Программа ::= [Контекст **]** Метапрограмма **? Данное**
// в контексте заданы макросы и трансформации
// Всегда выводится во внешний мир сообщение о результате
Метапрограмма ::= [Контекст**.**Диалект **] [** Набор**.**Диалект **]**
((Трансформация { Участок } **) ** Набор**.**Диалект **)**
[Набор**.**Диалект **]**

Трансформация ::= Макрос *// выглядит как функция*
| Шаблон *// выглядит как структура данных*
| Изменение *// работает как замена на другую форму*
| Преобразование *// правая часть эквивалентна левой*
| Перевод *// правая часть на другом диалекте*
| Трансформация / Действие
// можно что-то доделать после трансформации
Участок ::= Контекст**.**Диалог *// входит один раз*
| Набор**.**Диалог *// преобразуемые части программы*

Изменение ::= Шаблон → Шаблон'

// один шаг перехода к новому состоянию

// неизменяемая структура данных с изменяемыми значениями

Шаблон ::= **(** Имя **** Порядок_1 **)** *// в левой части*

// Регистр = адрес или структура в памяти

| **(** Имя' **** Порядок_2 **)** *// в правой части*

// Тот же самый Регистр с изменённым значением

// Порядки не обязаны совпадать

Преобразование ::= Образец_1 ↔ Образец_2

// обратимая трансформация текста

// при формировании правой части левая копируется в комментарии

Образец ::= (Функция АргТ **...**) **|** (Фильтр АргТ)

| (Структура АргТ **...**)

// в Образцах могут быть общие имена слева и справа

// слева — именование, справа — подстановка

| Фрагмент

| Определение

| Образец / Условие *// дополнительная защита, страж*

Перевод ::= Определение**.**Диалект_1 ↔ Определение**.**Диалект_2

Фрагмент ::= Символ **|** Лексема **|** Слог

Слог ::= Лексема Понятие

| Лексема Понятие Лексема

Знак ::= **{** пробел + * / - = /= < > **%** ~ , ; := <=> → ← _ ~ @**!!** ? ! ОК **:** **}**

// разновидность лексем

Ключ ::= **{** МАКРО ОШИБКА Мин Макс Длина Первый Последний ВЕРОЯТНО ЕСЛИ ТО КРОМЕ ЖДУ ПАУЗА ЦИКЛ ПОКА ПОВТОР СООБЩЕНИЕ РЕЗУЛЬТАТ СИГНАЛ ФУНКЦИЯ ПРИОСТАНОВКА ГОТОВЫ ПУСК СТОП ШАГ ВВЕРХ ВНИЗ ВПРАВО ВЛЕВО ВМЕСТЕ ОЧЕРЕДЬ БЛОКИРОВКА ПУСК ВОЗВРАТ УДАЛИТЬ ВСТАВИТЬ **}**

// разновидность лексем

Понятие ::={ КонтекстД НаборД // **у всех**
ВырД ИмяД ОпределениеД ФункцияД Мемо-функцияД СхемаД ДанноеД ФрагментД МакроПарД ТекстД РисункиД СтруктураД БарьерД ПотокД СвёрткаД ДействиеД КомандаД ЧислоД УсловиеД ФормулаД АргД ОперацияД АдресД ИсполнительД ИндексД **}** // **в образцах трансформаций**

АргТ ::= ВырТ **|** «_» // *хоть что*
// текст, выражение или что-то до разделителя
// аргументы вычисляются только явными командами

ВырТ ::= текст // *невычисляемый параметр*

Данное ::= Рисунки **|** Структура **|** Фрагмент **|** Программа **|** Исполнитель
// кроме изображения реквизита и исполнителей, структуры и
// редактируемые данные из протоколов и сценариев исполнителей

Контекст ::= ((Имя = Определение **)** \ «;» **)**
// В обстановке определения, включая протоколы

Определение ::= Функция **|** Мемо-функция **|** Схема *//Макрос*
| Исполнитель *// Создание нового*
| Трансформация *// Альтернативы и части*

Схема ::= Макрос *// Шаблон для подстановки или схема*

Макрос **::=** ИмяМ = МАКРО (ИмяП …)
Фрагмент (МакроПар \ «,») → Фрагмент′ (МакроПар′ \ «,»)
? «//» ?«Фрагмент (МакроПар \ «_»)»
// вывод исходной формулы в виде комментария

МакроПар ::= Текст1 **[** ~~ Текст2 **]** *// проверка корректности*
| Имя *// символьная константа без кавычек*
//текст, обладающий синтаксическим подобием виду параметра

Действие ::= Команда | [ОШИБКА**]** Данное
//допуск ошибочных ситуаций

Условие ::= Имя **|** Данное **|** Имя ~~ Текст *// Проверка на синтаксис*

Команда ::= РЕДАКТИРОВАТЬ **@!!** Имя Данное Адрес

//включая АПК и диалекты

| УДАЛИТЬ **@!!** Имя Данное Адрес

//из протокола исполнителя, включая АПК

| ВСТАВИТЬ **@!!** Имя Данное Адрес *// в протокола исполнителя*

Адрес ::= Число **|** Имя *// координаты для редактирующих воздействий*

Фрагмент ::= Данное *// не обязательно текст*

//данное, подготовленное в схему или сценарий исполнителя

Исполнитель ::= Редактор **|** Отладчик **|** Тестировщик **|** Оптимизатор

| НовыйИсполнитель

Выр ::= Имя **|** Данное **|** (ГенСим Имя) **|** (Имя ~~ Текст)

| (Функция Арг …) **|** (Фильтр Арг) **|** (Структура Арг …)

| (Схема Фрагмент …)

Арг ::= Выр

// выражение, вырабатывающее для функции входное значение

Структура ::= (Выр **** Порядок) **|** [Выр **** Порядок]

|{ Выр **** Порядок }

| [**((** Индекс **:** Выр **)** **** Порядок **)**]

| {**[**Имя**:** **]** Выр **[*** Целое **]** **** Порядок}

Порядок ::= { «;» «,» }

Индекс ::= Целое

НовыйИсполнитель ::= РОБОТ Имя СистемаКоманд Сценарий

СистемаКоманд ::= [Команда **** **«,»**]

Редактор

Отладчик

Тестировщик

Оптимизатор

(диалект Асинхр — функциональное программирование многопоточных программ как дополнение к Трансформ+Мульти)

Программа ::= [Контекст] [Набор] ? Выр /
/ Всегда есть сообщение о вычисленном результате
Определение ::= Функция | Мемо-функция *// Иерархия определений*
Данное ::= @ Программа | @ Данное | @ Функция | @ Мемо-функция
// адресуемые данные
Набор ::= [Имя =] ([Поток \ [,]) Свёртка]]
// свёртка результатов независимых потоков
Поток ::= [Имя =] (Действие) \ ([**;**]) Свёртка]]
// свёртка результатов последовательности действий потока
Свёртка ::= { + * Мин Макс Длина Первый Последний }
Действие ::= СООБЩЕНИЕ (Выр) | Имя = Выр | *Имя1* **<=>** *Имя2*
| [РЕЗУЛЬТАТ] Выр | СИГНАЛ (Имя) | **!** | **?** Выр
| *(Выр Фильтр)* **>>** *структура в СинПар*
| (Имя1 … ИмяК) → (Имя1′ … ИмяК′) | Команда [(Арг …)]
| **@!!** Имя *//вызов исполнителя или диалекта*
// пополняет систему команд и/или набор процессоров
Функция ::= ИмяФ = [ФУНКЦИЯ] (ИмяП …) Поток
| Мемо-Функ
// Определение функции, иерархия
Схема ::= ИмяМ = МАКРО (ИмяП …) Фрагмент
// Шаблон для подстановки или схема Трансформ
Условие ::= Имя | Данное | Выр *// любой предикат*
Выр ::= Формула | Фильтр
Команда ::= Формула
Формула ::= (Арг1 ОперацияИ Арг2) | (Арг ОперацияН)
| (ОперацияК Арг) *// списки операций*
ОперацияИ ::= { **= + * / - = /= < > % ~~ , ; := <=>** →} // 1 из
// все инфиксные
ОперацияН ::= { **-, ?, !**, ОК {Операция, Команда}} *// все префиксные*
ОперацияК ::= { **:** _//барьер, **@!!** //исполнитель}
Фильтр ::= (Арг1 ОперацияФ Арг2) *// все постфиксные*
ОперацияФ ::= {**= /= < > ~~ >> →**} *//в СИНПАР*
// перенос элементов структуры в другую

(диалект Синпар — параллельное программирование как расширение Аспар+Трансформ+Мульти)

Данное ::= Барьер *// Синхронизаторы*
| @ Программа | @ Данное | @ Функция | @ Мемо-функция
// адресуемые данные
Набор ::= [Имя =] [Барьер] ([Поток \ [,]) Свёртка]]
// Глобальный барьер для синхронизации всех потоков
| [Имя =] ([Поток \ [,]) Свёртка]]
// свёртка результатов независимых потоков
Поток ::= [Имя =] [Барьер] (Действие) \ ([**;**]) Свёртка]]
// свёртка результатов последовательности действий потока
Действие ::= [Барьер :] Действие *// локальный барьер*
| СООБЩЕНИЕ (Выр) | Имя = Выр | Имя1 **<=>** Имя2
| [РЕЗУЛЬТАТ] Выр | СИГНАЛ (Имя) | **!** | **?** Выр
| (Выр Фильтр) **>>** структура
| (Имя1 … ИмяК) → (Имя1′ … ИмяК′) | Команда [(Арг …)]
| **@!!** Имя *//вызов исполнителя или диалекта*
// пополняет систему команд и/или набор процессоров
Условие ::= (ПРИОСТАНОВКА? Барьер) | (ГОТОВЫ Барьер)
// достигнут локальный барьер
// достигнуты все одноимённые локальные барьеры
| Имя | Данное | Выр *// любой предикат*
Команда ::= (БЛОКИРОВКА Барьер) | (ПУСК Барьер)
// если барьер совпадает с глобальным, то приостановка потока
// Глобальный барьер достигнут во всех потоках,
// можно все их активировать
| Формула
Барьер ::= Имя

Интегральная сводка синтаксиса языка СИНХРО

Программа ::= [Контекст] [Набор] ? Выр /
Контекст::= ((Имя = Определение) \ **;**)
Определение ::= Функция | Мемо-функция | Схема | Данное
Схема ::= ИмяМ = МАКРО (ИмяП …) Фрагмент *(*МакроПар \ «_»*)*
→Фрагмент′ *(*МакроПар′ \ «_»*) ? «//» ?*«Фрагмент *(*МакроПар \ **«_»**)»*
МакроПар ::= Текст1 ~~ Текст2
Данное ::= Рисунки | Структура | Барьер | Фрагмент | @ Данное
| @ Программа | @ Определение | ОШИБКА Данное
Набор ::= [Имя =] [Барьер] ([Поток \ [,]) Свёртка]]
Барьер ::= Имя
Поток *::= [Имя =] (Действие) \ ([;]) Свёртка]]*
Свёртка ::= { + * Мин Макс Длина Первый Последний }
Действие ::= [Имя =] Команда | нена**@!!** Имя *//вызов исполнителя*
| [Барьер :] Действие *// локальный барьер*
| [ВЕРОЯТНО] Число/число Действие
| ЕСЛИ Условие ТО Действие
| КРОМЕ Условие ТО Действие
| ЖДУ (Имя) *// исполнитель рядом*
| ПАУЗА Число *// синхронизующий интервал времени*
| ЦИКЛ Действие1 ПОКА Условие ПОВТОР Действие2
| [ОШИБКА] Данное *//допуск ошибочных ситуаций*
| СООБЩЕНИЕ (Выр) | Имя = Выр | Имя1 **<=>** Имя2
| [РЕЗУЛЬТАТ] Выр | СИГНАЛ (Имя) | **!** | **?** Выр
| (Выр Фильтр) **>>** структура
| (Имя1 … ИмяК) → (Имя1′ … ИмяК′) | Команда [(Арг …)]
Функция ::= ИмяФ = [ФУНКЦИЯ] (ИмяП …) Поток
Схема ::= ИмяМ = МАКРО (ИмяП …) Фрагмент
Условие ::= Имя | Данное | Выр *// любой предикат*
| (ПРИОСТАНОВКА? Барьер) | (ГОТОВЫ Барьер)
Исполнитель ::= Имя | Кнопка | Дежурик | Муравей | Художник
| Математик | Писатель | ((Команда \ [,]) Программа)
| Редактор | Отладчик | Тестировщик

Выр ::= Имя | Данное | **(**Имя ~~ Текст **)** | Формула
| (Функция [Арг …]) | (Фильтр Арг) | (Структура Арг …)
| (Схема [Фрагмент …])
Арг – *вырабатывает для функции подходящее значение*
Формула ::= (Арг1 Операция Арг2) | (Арг Операция)
| (Операция Арг) *// списки операций*
(МИНФ Операция) = (Арг Операция Арг)
(**Операция** : { = + * / - = /= < > **%** ~~ **, ; :=** <=> →})
(МПРЕФ Операция) = (Операция Арг)
(**Операция**: { **-, ?, !**, ОК{Операция, Команда}})
(МПОСТ Операция) = (Арг Операция)
(**Операция** : { **:** _*//барьер*, **@!!** *//исполнитель*})
Формулы = (МИНФ { + * / - = /= < > **%** ~~ **, ; :=** <=> →})
| (МПРЕФ { -, ?, !, ОК}) *// все префиксные*
| (МПОСТ { **:** _ //барьер, @!! *//исполнитель*})
Фильтр ::= (МИНФ {= /= < > ~~ >> →})
Команда ::= Имя | ПУСК | СТОП | ШАГ **@!!** Имя
| ВВЕРХ | ВНИЗ | ВПРАВО | ВЛЕВО
| ВМЕСТЕ (Действие) \ ([,]) *// одновременно*
| ОЧЕРЕДЬ (Действие) \ ([**;**]) *// последовательно*
| (БЛОКИРОВКА Барьер) | (ПУСК Барьер)
| ВОЗВРАТ **@!!** Имя *// вернуть шаг исполнителя*
| РЕДАКТИРОВАТЬ **@!!** Имя Данное Адрес
| УДАЛИТЬ **@!!** Имя Данное Адрес
| ВСТАВИТЬ **@!!** Имя Данное Адрес
Имя — *лексема, зависит от реализации*
Адрес ::= Число | Имя *// координаты для редактора*
Фрагмент ::= Данное *// не обязательно текст*
Структура ::= (Выр \ **;**) | (Выр \ **,**) | [Выр \ **,**] | [Выр \ **;**] |{ Выр \ , }
| [(Индекс) **:** (Выр [,])] | [(Индекс) **:** (Выр ;)]
| { [Имя**:**] Выр [***** Целое } *// Кратность*
Индекс ::= Целое | Имя

Приложение 3

Проект реализации учебного языка программирования СИНХРО, предназначенного для ознакомления с параллельными процессами

Этапы обучения:

1) Процесс ознакомления начинается с опыта в оперативном управлении персонажами игры, участвующими во взаимодействующих процессах. Персонажей немного (2 — 6), каждый имеет свою панель управления. Сюжет и правила игры заранее известны. Для начала это игры на клетчатой доске. Игры специально подобраны так, чтобы возникали разные проблемы, которые можно избежать, если во время про них подумать. Этот этап нужен для формирования интуиции по программированию взаимодействующих процессов. Часть сюжетов взяты из книги Т.Хоара и других источников по простейшим моделям параллельных вычислений (Мульти-оперирование, Трансформ).

2) Далее идёт этап организации и отладки процессов взаимодействия параллельных программ на игрушечных сюжетах, возможно выбранных методистами или придуманных учащимися, включая конструирования своих вариантов исполнителей (Мульти-конструирование, Трансформ).

3) Затем приходит время представления предельно распараллеленных многопоточных программ по задачам из реального мира, хорошо знакомого учащимся (Аспар. Трансформ).

4, Приходит время ознакомления с опытом организации взаимодействующих процессов в многопоточном и многопроцессорном программировании, используя переход от прототипов на языке СИНХРО к обычным языкам программирования, не менее, чем к двум, один из которых — язык функционального программирования (Синпар, Трансформ).

5) Завершается работа с языком СИНХРО практикой организации многопроцессорных программ на диалектах АПК и Трансформ.

Язык СИНХРО содержит следующие подъязыки, связанные с этапами обучения:

Этапы 1 и 2. **Мульти** — подъязык низкого уровня конструирования учебных тренажёров и оперирования ими при ознакомлении с феноменами параллелизма и обучении системному программированию *(наследует Logo, Робик, HFP, Lsl)* — конструктор тренажёров в виде игр на клетчатой доске. Моделирование одновременности для визуализации параллелизма отличается от

последовательности более коротким интервалом прорисовки элементов. Реализация Мульти представляет собой расширяемую **визуальную интерфейсную оболочку** языка СИНХРО.
Этап 3. **Асинхр** — подъязык сверх высокого уровня для представления естественного параллелизма с различными схемами управления фрагментами при определении **асинхронных** программируемых процессов *(наследует Setl, БАРС, OpenMP, Haskell, F#). Реализация устроена как синтаксически управляемый интерпретатор, допускающий настройку на разные схемы параллелизма. Поддерживаются* теоретико-множественные спецификации и ООП. Асинхр представляет собой более **абстрактную оболочку** языка СИНХРО.
Этап 4. **СинПар** — подъязык высокого уровня для **синхронизации** много-поточных программ с использования локальной памяти без побочных эффектов *(наследует APL, Sisal, MPI). Реализация устроена как компилятор с ЯВУ на виртуальную машину,* позволяющий быстро отлаживать программы в рамках парадигмы функционального программирования. СинПар представляет собой **основную оболочку** языка СИНХРО.
Этап 5. **АПК (абстрактный многопроцессорный комплекс)** — подъязык низкого уровня или виртуальная машина для представления многопроцессорных программ с использованием общей памяти *(наследует bash, SECD, pi-код, JVM, Сигма). Реализация устроена как интерпретатор, позволяющий варьировать ход продвижения процессов.* АПК представляет собой ядро языка СИНХРО, предполагающую машинно-ориентированную разработку для разных архитектур, система команд ядра дана в Приложении 2.
Этапы 1-5. **Трансформ** — подъязык мета-программирования для представления, отладки, оптимизации, декомпозиции и **преобразования** программ на всех этапах обучения, а также измерения характеристик производительности программ *(наследует БНФ, Рефал, YACC, LEX, mpC, Clang-LLVM) — символьный макропроцессор с синтаксическим контролем типов параметров. Реализация устроена как макрогенератор, нагруженный средствами контроля видов параметров макроса.* Трансформ допускает реконфигурацию, комплексацию и вообще любые программируемые **трансформации** как программ, так и открытой системы программирования (СП) для языка СИНХРО, представляет собой сборщик программ языка СИНХРО. Открытость СП понимается как доступ к исходным текстам, право их изменять и создавать свои версии СП.

Таким образом, реализация языка СИНХРО устроена как взаимодействие виртуальной машины АПК, сборщика программ Трансформ и оболочек языка

Мульти, СинПар. Асинхр , выбираемых в зависимости от этапа обучения, допускающих совместное применение.

Системные модули:
А. Абстрактная, виртуальная машина, приспособленная к расширению системы команд и уточнению отдельных механизмов интерпретации низкоуровневых программ над многопроцессорным комплексом (**АПК**).
Б. Синтаксически управляемый обработчик-конструктор программ, поддерживающий программируемые расширения **БНФ** и оперирование произвольными маршрутами в синтаксическом графе определения языка программирования, включая выделение подъязыков, соответствующих представленным наборам тестов. (**Трансформ**)
В. Универсальный компилятор-интерпретатор, допускающий сменную кодогенерацию для переносимости программ на разные многопроцессорные комплексы (граф-процессоры. суперкомпьютеры и сети, мобильные устройства — расширенные **вариации** АПК).
Г, Универсальный аппликативный интерпретатор, настраиваемый на разные семантические системы для **генерации** разного уровня оболочек (Мульти, СинПар. Асинхр).

Общая схема реализации представлена таблицей ___

Уровень	*Этапы 1 и 2*	*Этап 3*	*Этап 4*	*Этап 5*
Мета	**БНФ-**управляемый отладчик-конструктор-сборщик программ **Трансформ**			
Оболочки	**Мульти**	**Асинхр**	**СинПар.**	Другие, целевые ЯП
Ядро	Абстрактная, виртуальная машина **АПК**)			

Л.В. Городняя

СИНХРО - ЯЗЫК ОЗНАКОМЛЕНИЯ С МИРОМ ПАРАЛЛЕЛИЗМА

Препринт

___

Рукопись поступила в редакцию
Редактор
Рецензент

Подписано в печать
Формат бумаги 60 × 84 1/16 Объем 0.7 уч.-изд.л.
Тираж __ экз.