Основы функционального программирования

Учебное пособие

Л.В.Городняя
Gorod@iis.nsk.su

Новосибирск, 2004

Содержание лекций стр

Учебное пособие разработано при поддержке Интернет-Университета Информационных технологий и опубликовано в серии «Основы информационных технологий» в 2004 году.

Курс разработан на базе Факультета информационных технологий Новосибирского госуниверситета.

Содержание курса соответствует PF1, PL7, PL3 классификатора CC2001CS.

Лекция 1. **Основные идеи**

В этой лекции дается общее представление о функциональном программировании и сфере его применения, включая анализ основных понятий и принципов и их иллюстрацию на материале истории языка Лисп, его диалектов, наследников и реализаций. Рассматривается роль функциональных программ в жизненном цикле программного обеспечения и эволюции информационных технологий, а также перспективы функционального подхода к совершенствованию информационных систем.

Идея функционального программирования опирается на интуитивное представление о **функциях** как о достаточно общем механизме представления и анализа решений сложных задач. Механизм функций основательно изучен математиками, и это позволяет программистам наследовать выверенные построения, обладающие предельно высокой моделирующей силой. Систематическое применение функционального программирования впервые достаточно ярко было продемонстрировано Джоном Мак-Карти и его учениками в методах реализации языка **Лисп** и программирования на этом языке [1]. Наиболее очевидные из этих методов были успешно ассимилированы другими **языками и системами программирования**. Обычно про функциональное программирование вспоминают при смене технологий, когда возрастает роль аналитики и исследовательских задач. В настоящее время часто употребляют термин <**функциональность**> при сравнительной характеристике информационных систем, что, видимо, свидетельствует о проявлении новой метрики, заслуживающей отдельного рассмотрения [2].

Функциональный стиль объединяет разные подходы к **определению процессов** вычисления на основе достаточно строгих абстрактных понятий и методов **символьной обработки** данных. Связь функционального программирования с математическими основами позволяет в тексте программы наследовать доказательность построения результата - если она достигнута, причем с использованием разных методов абстрагирования решаемой задачи [3,4].

Сложность решения задач с помощью функциональных определений преодолевается чисто алгебраически. Это позволяет представлять классы задач и их решений строгими

формулами, для наглядности упрощаемыми введением дополнительных функциональных символов. Активно используется **рекурсия** и символьные обозначения как данных, так и действий и любых формул, удобных при определении функций. Минимальный набор обозначений, к которым можно свести все правильные, т.е. вычислимые формулы системы, играет роль **базиса системы**, реализация которого является минимальной версией всей системы.

Основная трудность перехода к функциональному программированию - соблазн легкого пути, т.е. стремление быстро смоделировать привычные средства и методы программирования. Более надежный путь - исследовать функциональное программирование как незнакомый мир. Идеи функционального программирования легче воспринять как самостоятельную теорию или интеллектуальную игру, которая новыми путями непременно приведет к знакомым и интересным задачам, но обеспечит преимущество - изящные решения и глубину понимания.

Джон Мак-Карти предложил рассматривать функции как общее базовое понятие, к которому достаточно естественно могут быть сведены все другие понятия, возникающие при программировании [1]. Идеи языка Лисп вызвали не утихающие по сей день дискуссии о приоритетах в программировании и сущности программирования. Лисп послужил эффективным инструментом экспериментальной поддержки **теории программирования** и развития сферы его применения. Рост интереса к Лиспу коррелирует с улучшением элементной базы, повышением эксплуатационных характеристик оборудования и появлением новых сфер применения ИТ. Существует и активно применяется более трехсот диалектов Лиспа и родственных ему языков: Interlisp, muLisp, Clisp, Sheame, ML, Cmucl, Logo, Hope, Sisal, Haskell, Miranda и др.

Изучение функционального программирования начинается с овладения техникой работы с так называемыми **<чистыми>, строго математическими, идеальными функциями**. Для реализации функций характерен отказ от необоснованного использования присваиваний и низкоуровневого управления вычислениями в терминах передачи управления. Такие функции удобны при отладке и тестировании благодаря независимости от контекста описания и предпочтения явно выделенного **чистого результата**. Трудоемкость отладки композиций из хорошо определенных функций растет аддитивно, а не мультипликативно.

Кроме того, системы из таких функций могут развиваться в любом направлении: сверху вниз и снизу вверх (а также “вширь” и “вузь”, если понадобится)[С.С.Лавров. МПСС-84]. Можно быстро продвинуться по сложности решаемой задачи, не отвлекаясь на синтаксическое разнообразие и коллизии при обработке общих данных. Концептуально близкие идеи <структурированного программирования> были сформулированы лишь более чем через десять лет.

Особый интерес представляют рекурсивные функции и методы их реализации в системах программирования. Интуитивное понятие функции, в отличие от классического понятия множества, отчасти содержит концепцию времени: сначала аргументы вычисляются в порядке вхождения, затем в соответствии с заданным алгоритмом строится значение функции - ее результат, возможно, явно зависящий от результатов других функций или от этой же функции, но при других, ранее вычисленных, значениях аргументов.

Исследовательская и проектная работа обычно проходит фазу поиска оптимального решения. Функциональное программирование для поддержки этой фазы предлагает ряд обобщенных функций: в качестве результата функции допускаются **варианты значений,** равноправно выбираемые из конечного множества значений. Равноправие не распространяется лишь на тупиковую ситуацию, когда ни один предложенный вариант не может быть вычислен.

Существует много неимперативных моделей управления процессами, позволяющих прерывать и откладывать процессы, а потом восстанавливать их и запускать или отменять. Организация такого управления, достаточного для оптимизации и программирования параллельных процессов, реализуется с помощью так называемых **<замедленных>** или **<ленивых> вычислений (lazy evaluation)**. Основная идея таких вычислений заключается в сведении вызовов функций к представлению рецептов их вычисления в определенном контексте. Вычисляться каждый такой рецепт может не более чем один раз и то если его результат действительно нужен.

Здание функционального программирования получает логическое завершение на уровне определения **функций высших порядков**, удобных для синтаксически управляемого конструирования программ на основе спецификаций, типов данных, визуальных диаграмм, формул и т.п.

Известно, что лаконичность рекурсии может скрывать нелегкий путь. А.П.Ершов в предисловии к книге П.Хендерсона [3] привел поучительный пример не поддавшегося А.Чёрчу решения задачи о рекурсивной формуле, сводящей вычитание единицы из натурального числа к прибавлению единицы {1 -1 = 0 | (n +1) -1 = n }. Оно получено С.Клини лишь в 1932 году:

n –1 = F (n, 0, 0), где F (x, y, z) = если (x = 1) то 0
иначе если ((y +1) = x) то z
иначе F (x, y +1, z +1)

Решение получилось через введение формально усложненной функции F со вспомогательными аргументами, что противоречит интуитивному стремлению к монотонности и движению от простого к сложному.

История Лиспа насыщена жаркими спорами, притиворечивыми суждениями, яркими достижениями и смелыми изобретениями, которых более чем достаточно, чтобы утверждать: «Лисп - гениальное творение Джона МакКарти». Потенциал данного языка еще предстоит раскрыть. Выразительная сила Лиспа обретает новое дыхание на каждом эволюционном витке развития информационных технологий. При сравнительном анализе информационных систем моделирование их семантики на Лиспе позволяет классифицировать функционирование по уровню сложности, зрелости, полноты, точности и организованности. Универсальность Лиспа достаточна для изучения на его основе любой парадигмы информатики и программотехники. Лисп содержит в себе эталонную семантическую систему, пригодную для измерения функциональности других систем.

Информационный мир становится все более динамичным - Лисп приспособлен к программированию развивающихся построений и реорганизуемых конфигураций из разносортных компонентов. Многие реализационные находки Лиспа, такие как ссылочная организация памяти, “сборка мусора” для повторного использования памяти, частичная компиляция программ с интерпретацией промежуточного кода, полиморфизм, длительное хранение атрибутов объектов в период их использования и т.д. перекочевали из области исследований и экспериментов на базе Лиспа в практику реализации систем программирования.

Диалекты Лиспа (Logo, ML, MuLisp, Scheme, Hope, AutoLisp, CommonLisp, Reduce и др.) заняли обширную нишу в области учебно-экспериментального программирования, связанного с развитием теории программирования, системного программирования, разработки и прототипирования новых компьютерных комплексов и архитектур, конструирования и исследования систем построения оптимизирующих компиляторов и организации особо точных и высокопроизводительных вычислений.

На первый взгляд, идеи Лиспа противоречат традиционным подходам к программированию. Но это противоречие отступает перед строгой логикой языка, гармонично уравновешенной полнотой и ясностью реализационных решений. Определение Лиспа дает надежную основу для развития, варьирования и расширения Лисп-систем средствами как самого Лиспа, так и его окружения.

Базис Лиспа предельно лаконичен - атомы и структуры из простейших бинарных узлов плюс несколько базовых функций и функционалов. Базис содержит **встроенные (примитивные) функции**, которые анализируют, строят и разбирают любые структурные значения (atom, eq, cons, car, cdr,), и встроенные **специальные функции и функционалы**, которые управляют обработкой структур, представляющих вычисляемые выражения (quote, cond, lambda, eval). Над базисом строятся предельно простые формулы в виде круглоскобочных списков, где первый элемент - функция, остальные - ее аргументы, в том числе переменные, реализуемые с помощью разных вариантов стека или ассоциативного списка. Подробнее с идеями Лиспа и его математическими основами можно ознакомиться на страницах журнала <Компьютерные инструменты в бразовании>, N 2-5 за 2002 год.

Синтаксис Лиспа не требует особых ресурсов для запоминания разделителей и/или ограничителей и напряженного внимания на распознавание синтаксических позиций в разных рамочных конструкциях. Универсальный разделитель - пробел, ограничители - круглые скобки. В скобки заключается представление функции с ее аргументами. Все остальное - вариации в зависимости от **категории функций**, определенности атомов и вычислимости выражений, типов значений и структур данных. Функционалы - это одна из категорий функций, используемая при организации управления вычислениями.

По современным меркам реализации Лиспа компактны и не слишком требовательны к оборудованию. Существуют свободно распространяемые версии, занимающие менее мегабайта, пригодные к применению на любом процессоре.

В нашей стране программирование соприкоснулось с Лиспом из первых рук. Джон Мак-Карти в конце 1968 года познакомил Москву и Новосибирск с Лиспом, что побудило к реализации отечественных версий языка. Две реализации на БЭСМ-6 (ВЦ АН под рук. С. С. Лаврова и ВЦ СО АН под руководством А. П. Ершова) и одна на ЕС ЭВМ (ВЦ АН под рук. С. С. Лаврова) нашли применение в отечественных проектах по системному и теоретическому программированию, в исследованиях по математической лингвистике, искусственному интеллекту и обработке химических формул.

В настоящее время наблюдается устойчивый рост рейтинга интерпретируемых языков программирования и включения в компилируемые языки механизмов символьной обработки и средств динамического анализа, что повышает интерес к Лиспу.

История создания и развития языка программирования ЛИСП интересна как **социальный феномен** осуществления замысла, вызвавшего серьезные возражения и математиков, и программистов, но показавшего удивительную живучесть. Критика теоретиков была связана с так называемой "парадоксальностью" бестипового лямбда-исчисления. Практиков пугали накладные расходы на сборку мусора и интерпретацию в сравнении с компиляцией и статическим распределением памяти. (О наличии компилятора в традиционных Лисп-системах вспоминают редко.)

Наиболее общие принципы функционального программирования:

1) **Унификация понятий <функция> и <значение>.**

При символьном представлении информации нет принципиальной разницы в природе изображения значений и функций. Следовательно, нет и препятствий для обработки представлений функций теми же средствами, какими обрабатываются значения, т.е. представления функций можно строить из их частей и даже вычислять по мере поступления и обработки информации. Именно так конструируют программы компиляторы. В замкнутых системах не принято к такой технике информационных воздействий допускать обычных пользователей. Но исследователь вынужден вникать во все уровни своего экспериментального полигона.

2) Кроме функций-констант, вполне допустимы **функции-переменные.**

Отсутствие навыков работы с **функциональными переменными** говорит лишь о том, что надо осваивать такую возможность, потенциал которой может превзойти наши ожидания теперь, когда программирование становится все более аспектно и компонентно ориентированным. Объектно-ориентированная парадигма в этом плане не обладает достаточной функциональной полнотой.

3) **Самоприменимость.**

Первые реализации Лиспа были выполнены методом раскрутки, причем в составе системы сразу были предусмотрены и интерпретатор, и компилятор. Оба инструмента были весьма точно описаны на самом Лиспе, причем основной объем описаний не превосходил пару страниц, что позволяло активно использовать эти описания при изучении языка и программ, написанных на нем. Этот эксперимент послужил базой для определения систем программирования с помощью так называемой <операционной семантики>, получившей развитие в работах по Венской методике определения языков и систем программирования.

4) **Интегральность ограничений** на пространственно-временные характеристики.

Если не хватает памяти, то принципиально на всю задачу, а не на отдельные блоки данных, возможно, не слишком существенных для ее решения. При недостатке памяти специальная программа- “мусорщик” пытается найти свободную память. Новые реализации этого механизма рационально учитывают преимущества восходящих процессов на больших объемах памяти.

5) **Уточняемость решений.**

Современное применение информационных систем достаточно широко соприкасается с вариантами доступных решений, необходимостью уточнять отдельные особенности применения готовых решений и трудоемкостью анализа и поиска рационального выбора комплекта используемых средств. Реализация Лиспа обычно содержит списки свойств объектов, приспособленные к внешнему доопределению отдельных элементов поведения программируемой системы.

6) **Множественность определений.**

Наиболее концептуально полный Lisp 1.5 допускает множественные определения имен, что в рамках настраиваемой интерпретации обеспечивает кроме общеизвестного полиморфизма более общие схемы обработки ряда версий или вариантов функциональных построений.

Как авиация не соревнуется с автотранспортом в объеме грузоперевозок, так и функциональное программирование несравнимо со стандартными парадигмами в массовости применения, но это не умаляет его достоинств.

Многие современные языки и технологии программирования унаследовали опыт реализации и применения Лиспа и других языков символьной обработки. Так, например, Java берет на вооружение идеи неполной компиляции и освобождения памяти, объектно-ориентированное программирование реализует объекты, весьма похожие на списки свойств атомов Лиспа, хэш-таблицы языка Perl напоминают применение ассоциативных списков Лиспа. Python обрабатывает программы с нетипизированными переменными. Функциональное программирование, базирующееся на опыте применения Лиспа, можно описать в терминах любого языка, но Лисп дает этой идее достаточно полное звучание и формирует некую шкалу сравнения и определения стандартных конструкций и методов программирования, а также упорядочение явлений, характерных для экспериментальной разработки программ и поиска новых областей их применения.

Лекция 2. Элементарный Лисп

> Изучается ключевой метод функционального программирования – выбор семантического базиса для класса решаемых задач на примере организации информационной обработки символьными выражениями в языке Лисп. Вводятся базовые понятия, достаточные для символьного представления программ. Формализм рекурсивных функций и простые алгоритмы символьной обработки привлечены для обоснования и демонстрации функционального подхода к представлению программ. Анализируются требования к полноте и эффективности их обработки. В качестве исторической иллюстрации полномасштабного применения функционального программирования для решения достаточно сложной задачи используется символика языка Лисп, выбранная при организации символьной обработки для решения задач искусственного интеллекта [1].

Функциональный стиль программирования сложился в практике решения задач символьной обработки данных в предположении, что любая информация для компьютерной обработки может быть сведена к символьной.

Методы функционального программирования основаны на формальном математическом языке представления и преобразования формул. В дальнейшем будут проиллюстрированы практические способы применения функционального программирования, превращающие его в удобную технологию современного программирования.

Функциональное программирование отличается от большинства подходов к программированию тремя важными принципами:

1) Природа данных

Все данные представляются в форме символьных выражений. Дж. Мак-Карти назвал их S-выражениями. Состав **S-выражений** и типы их элементов не ограничиваются. Это позволяет с их помощью представлять любую информацию. Система программирования над такими структурами обычно использует для их хранения всю доступную память, поэтому программист освобожден от распределения памяти под отдельные блоки данных.

2) Самоописание обработки символьных выражений

Важная особенность функционального программирования состоит в том, что описание способов обработки S-выражений представляется программами, рассматриваемыми как символьные данные. Программы строятся из **рекурсивных функций** над S-выражениями. Определения и вызовы этих функций, как и любая информация, имеют вид S-выражений, то есть формально они могут обрабатываться как обычные данные, получаться в процессе вычислений и преобразовываться как **значения**.

3) Подобие машинным языкам

Система функционального программирования допускает, что программа может интерпретировать и/или компилировать программы, представленные в виде S-выражений. Это сближает методы функционального программирования с методами низкоуровнего программирования и отличает от традиционной методики применения языков высокого уровня. В принципе, такая возможность достижима на любом стандартном языке, но так делать не принято.

Наиболее очевидные следствия из выбранных принципов:

- Процесс разработки программ разбивается на две фазы: **построение базиса и его пошаговое расширение**.

-
- **Прозрачность ссылок** обеспечена совпадением значений одинаково выглядящих формул, вычисляемых в одинаковом контексте.
-
- **Стратегия универсальных функций и функционально полных систем**, трудоемкость первичной реализации которых компенсируется надежностью определений и простотой применения.
-
- Функциональное программирование активно применяется для генерации программ и систем, применяемых в областях с низкой кратностью повторения отлаженных решений (например, в учебе, проектировании, творчестве и научных исследованиях), ориентированных на оперативные изменения, уточнения, улучшения, адаптацию и т.п.

2.1. Данные

Любые **структуры данных** начинаются с элементарных значений. Следуя Лиспу, в функциональном программировании такие значения называют **атомами** или **символами**. Внешне атом обычно выглядит как последовательность из букв и цифр, начинающаяся с буквы, включая и другие литеры, не занятые в синтаксисе.

A
ATOM
ЛЕКЦИЯ2
ВотДлинныйАтомОченьДлинныйЕслиНадоАтомМожетБытьЕщеДлиннее
Ф4-длш139_к131б

Одинаково выглядящие атомы не различимы по своим свойствам, хотя проявления этих свойств могут быть обусловлены контекстом использования атомов. Термин “атом” выбран по аналогии с химическими атомами. Согласно этой аналогии атом может иметь достаточно сложное строение, но оно не рассматривается как обычное S-выражение. Устройство атома реализационно зависимо и лишь соответствует некоторой спецификации, содержание которой будет рассмотрено в девятой лекции. Атом не предназначен для разбора на части базовыми средствами языка.
Более сложные данные выстраиваются из унифицированных структур данных - одинаково устроенных **блоков памяти**. В Лиспе это **бинарные узлы**, содержащие **пары** объектов произвольного вида. Каждый бинарный узел соответствует минимальному блоку памяти, выделяемому системой программирования при организации и обработке структур данных. Выделение блока памяти и размещение в нем пары данных выполняет функция **CONS** (от слова consolidation), а извлечение левой и правой частей из блока выполняют функции **CAR** и **CDR,** соответственно.

**) Примечание.* В других языках функционального программирования используются векторы, кортежи, последовательности, потоки, множества, сети и другие структуры данных, обладающие достаточной гибкостью, т.е. способностью к организации единого доступа к любому числу элементов произвольного типа.

CONS => [CAR | CDR]

Бинарный узел, содержащий пару атомов ATOM и **Nil**, рассматривается как одноэлементный **список**:

(ATOM) = [ATOM | Nil]

Если вместо атома ATOM рекурсивно подставлять произвольные атомы, а вместо Nil - произвольные списки, затем вместо ATOM - построенные списки и так далее, то мы получим множество всех возможных списков. Атом Nil играет роль **пустого списка** и фактически эквивалентен ему. Можно сказать, что **список** - это заключенная в скобки последовательность из атомов, разделенных пробелами, или списков.

ATOM
(A B)
(A B C D E F G H I J K L M N O P R S T U V W X Y Z)
(C (A B))
((A B) C)
((A B) (D C))
((A B)(D(C E)))

Такая форма представления информации называется **списочной записью** (list-notation). Ее основные достоинства - лаконичность, удобство набора и отсутствие "синтаксического сахара". Она достаточно отражает взаимосвязи структур данных, размещаемых в памяти, и помогает задавать процедуры доступа к их компонентам.

Любой список может быть построен из пустого списка и атомов с помощью CONS, и любая его часть может быть выделена с помощью подходящей **композиции CAR-CDR**.

Функция CONS строит списки из бинарных узлов, заполняя их парами объектов, являющихся значениями пары ее аргументов. Первый аргумент произвольного вида размещается в левой части бинарного узла, а второй, являющийся списком, - в правой.

Функция CAR обеспечивает доступ к первому элементу списка - его "**голове**", а функция CDR - к укороченному на один элемент списку - его "**хвосту**", т.е. к тому, что остается после удаления головы.

Функция **ATOM** позволяет различать **составные** и атомарные **объекты**: на атомах ее значение "истина", а на структурированных объектах – "ложь".

Функция **EQ** выполняет проверку **атомарных** объектов на **равенство**.

Различие истинностных значений в Лиспе принято отождествлять с разницей между пустым списком и остальными объектами, которым программист может придать в программе некоторый другой смысл. Таким образом, значение "**ложь" – это всегда Nil**.

**) Примечание.* Во многих ЯП испльзуется 0 – 1 или идентификаторы True – False и др.

Если требуется явно изобразить **истинностное значение**, то для определенности в качестве значения "истина" используется константа – атом T (true) как стандартное представление, но роль такого значения может выполнить любой, отличный от пустого списка, объект.

Таблица 2.1. **Операции над списками.** Примеры соответствия между аргументами и результатами элементарных функций обработки списков .

Функция	Аргументы	Результат
	Конструирование структур данных	
CONS	A и Nil	(A)
CONS	(A B) и Nil	((A B))
CONS CONS	A и (B) (Результат предыдущего CONS) и ©	(A B) ((A B) C)
CONS	A и (B C)	(A B C)
	Доступ к компонентам структуры	

	данных:	
	Слева	
CAR	(A B C)	A
CAR	(A (B C))	A
CAR	((A B) C)	(A B)
CAR	A	Не определен
	Справа	
CDR	(A)	Nil
CDR	(A B C D)	(B C D)
CDR	(A (B C))	((B C))
CDR	((A B) C)	©
CDR	A	Не определен
	Смешаная обработка данных:	
CDR CAR	(A B C) Результат предыдущего CDR	(B C) B
CAR CAR	(A C) Результат предыдущего CAR	A Не определен
CONS CAR	A и (B) Результат предыдущего CONS	(A B) A
CONS CDR	A и (B) Результат предыдущего CONS	(A B) (B)
	Предикаты:	
	Атомарность – неделимость	
ATOM	VeryLongStringOfLetters	T
ATOM	(A B)	Nil - выполняет роль ложного значения
CDR ATOM	(A B) Результат предыдущего CDR	(B) Nil
ATOM	Nil	T
ATOM	( )	T
	Равенство	
EQ	A A	T
EQ	A B	Nil
EQ	A (A B)	Nil
EQ	(A B) (A B)	Не определен
EQ	Nil и ()	T

2.3. Точечная нотация

Исторически при реализации Лиспа в качестве единой базовой структуры для конструирования S-выражений использовалась так называемая "**точечная нотация**" (dot-notation), согласно которой левая и правая части бинарного узла равноправны и могут хранить данные любой природы.

Бинарный узел, содержащий пару атомов ATOM1 и ATOM2, можно представить в виде S-выражения вида:

(ATOM1 . ATOM2)

Если вместо атомов ATOM1, ATOM2 рекурсивно подставлять произвольные атомы, затем построенные из них пары и так далее, то получим множество всех возможных S-выражений.

Можно сказать, что S-выражение - это или атом или заключенная в скобки пара из двух С-

выражений, разделенных точкой. Все сложные данные выстраиваются из одинаково устроенных блоков - бинарных узлов, содержащих пары объектов произвольного вида. Каждый бинарный узел соответствует минимальному блоку памяти.

ATOM
(A . B)
(C . (A . B))
((A . B) . C)
((A . B) . (D . C))
((A . B) . (D . (C . E)))

Любое S-выражение может быть построено из атомов с помощью CONS, и любая его часть может быть выделена с помощью CAR-CDR.

Расширение типа данных, допускаемых в качестве второго аргумента “CONS”, ни в малейшей степени не усложняет реализацию этой функции, равно как и реализацию функций “CAR” и “CDR”, зато их описания становятся проще.

Функция CONS строит бинарный узел и заполняет его парой объектов, являющихся значениями пары ее аргументов. Первый аргумент размещается в левой части бинарного узла, а второй - в правой. Функция CAR обеспечивает доступ к объектам, расположенным слева от точки, а функция CDR - справа.

Таблица 2.2 Элементарные функции над произвольными S-выражениями

Функция	Аргументы	Результат
	Конструирование структур данных	
CONS	A и B	(A . B)
CONS	(A . B) и C	((A . B) . C)
CONS CONS	A B (Результат предыдущего CONS) и C	(A . B) ((A . B) . C)
	Доступ к компонентам структуры данных:	
	Слева	
CAR	(A . B)	A
CAR	((A . B) . C)	(A . B)
	Справа	
CDR	(A . B)	B
CDR	(A . (B . C))	(B . C)
	Смешанная обработка данных:	
CDR CAR	(A . (B . C)) Результат предыдущего CDR	(B . C) B
CDR CAR	(A . C) Результат предыдущего CDR	C Не определен
CONS CAR	A и B Результат предыдущего CONS	(A . B) A
CONS CDR	A и B Результат предыдущего CONS	(A . B) B
	Тождества: (на произвольных объектах)	
CONS CAR	Два произвольных объекта x и y Результат предыдущего CONS	 Исходный объект x

		(первый аргумент CONS)
CONS CDR	Два произвольных объекта x и y Результат предыдущего CONS	Исходный объект y (второй аргумент CONS)
CAR CDR CONS	Произвольный составной объект x - не атом. Тот же самый объект x. Результаты предыдущих CAR и CDR	Исходный объект x
	Предикаты:	
	Атомарность – неделимость	
ATOM	(A . B)	Nil - выполняет роль ложного значения
CDR ATOM	(A . B) Результат предыдущего CDR	B T
	Равенство	
EQ	(A . B) (A . B)	Не определен

Точечная нотация точнее, чем списки, представляет логику хранения данных в памяти и доступа к компонентам структур данных, но для непосредственного представления и обработки символьных выражений она оказалась менее удобной. Практически сразу была предложена более лаконичная запись наиболее употребимого подкласса S-выражений в виде списков произвольной длины вида (A B C D E F G H). В виде списков можно представить лишь те S-выражения, в которых при движении вправо в конце концов обнаруживается атом Nil, символизирующий завершение списка.

Атом Nil, рассматриваемый как представление пустого списка (), играет роль ограничителя в любом списке. Одноэлементный список (A) идентичен S-выражению (A . Nil). Список (A1 A2 ... Ak) может быть представлен как S-выражение вида:

(A1 . (A2 . (... . (Ak . Nil) ...))).

В памяти это фактически одна и та же структура данных.

**) Примечание.* Запятая в качестве разделителя элементов списка в первых реализациях Лиспа поддерживалась, но не прижилась. Пробел оказался удобнее.

Таблица 2.3. Соответствие списков и равнозначных им S-выражений

List-notation - списочная запись объекта	Dot-notation - точечная нотация того же объекта
(A B C)	(A . (B . (C . Nil)))
((A B) C)	((A . (B . Nil)) . (C . Nil))
(A B (C E))	(A . (B . ((C . (E . Nil)). Nil)))
(A)	(A . Nil)
((A))	((A . Nil) . Nil
(A (B . C))	(A . ((B . C) . Nil))
(())	(Nil . Nil)

Для многошагового доступа к отдельным элементам такой структуры удобно пользоваться мнемоничными обзначениями **композиций** из многократных CAR-CDR. Имена таких композиций устроены как цепочки из “a” или “d”, задающие **маршрут движения** из шагов CAR и CDR, соответственно, расположенный между “c” и “r”.

Указанные таким способом CAR-CDR исполняются с ближайшего к аргументу шага, т.е. в порядке, обратном записи.

Таблица 2.4. Примеры многошагового доступа к элементам структуры

	Многократные CAR-CDR	Вычисляются в порядке, обратном записи:
Caar	((A) B C)	A
Cadr	(A B C)	B - CDR затем CAR
Caddr	(A B C)	C - (дважды CDR) затем CAR
Cadadr	(A (B C) D)	C - два раза (CDR затем CAR)

Гипотезу об универсальности символьных данных, прежде всего, следует проверить при выборе представления форм, возникающих при написании программы и ее основного конструктива - переменных, констант, выражений, определений, ветвлений, вызовов функций:

1) Самая простая **форма** выражения - это **переменная**. Она может быть представлена как атом.

   X
   y
   n
   Variable1
   LongSong

2) Более сложные **формы** могут пониматься как применение функции к ее аргументам (**вызов функции**). **Аргументом функции** может быть любая форма. Вызов функции можно строить как список, первый элемент которого – представление функции, остальные элементы - аргументы функции.

   (функция аргумент1 аргумент2 ...)

Обычно S-выражение - это или атом, или список; значит, неатомарное S-выражение можно понимать как вызов функции, если все остальные понятия свести к применению некоторых функциональных объектов.

Теперь можно записывать формы для получения результатов операций из таблиц 2.1 и 2.2 в виде списков:

(ATOM Nil)
(CONS Nil Nil)

3) **Имена** функций, как и переменных, лучше всего изображать с помощью атомов, для наглядности можно выбирать заглавные буквы.

   Соответствие между именем функции и ее определением можно задать с помощью специального **конструктора функций LABEL**, первый аргумент которого - имя функции, второй – собственно именуемое определение функции. Формальным результатом LABEL является ее первый аргумент, который становится **объектом другой категории**. Он меняет свой статус – теперь это имя новой функции.

Пример 2.1:

Определение функции	Комментарии
(LABEL третий (LAMBDA (x) (CAR (CDR (CDR x)))))	имя новой функции параметры функции тело функции

Новая функция “третий” действует так же, как “Caddr” в таблице 2.4.

Именование функций работает подобно присваиванию значений переменным, но идентификатору присваивается объект другой категории – структура, символизирующая функциональный объект, содержащая список формальных параметров функции и тело ее определения. По отношению к процессам обработки данных разница между константами и переменными заключается лишь в том, что значение переменной может быть в любой момент изменено, а **константа изменяется существенно реже**. Обычно в рассуждениях о переменных и константах подразумевается, что речь идет лишь о данных. Поскольку функциональное программирование рассматривает представления функций как данные, постольку и **функции могут быть как константными, так и переменными**. Представления функции могут вычисляться и передаваться как параметры или результаты других функций. Соответствие между именем функции и ее определением может быть изменено, подобно тому, как меняется соответствие между именем переменной и ее значением.

4) **Композиции функций** можно строить с помощью вложенных скобок.

(функция1 (функция2 аргумент21 аргумент22 ...)
аргумент2 ...)

Приведенные правила ничего не говорят ни о природе данных и функций, ни о порядке вычисления аргументов и композиций функций. Речь идет исключительно о форме - внешнем виде списков, используемых для записи программы. Такая **синтаксическая оболочка**, в которой еще не конкретизированы операции над данными, является общей спецификацией реализационной основы для определения **аппликативных систем**, допускающих специализацию практически в любом направлении. Можно сказать, что Лисп является аппликативной системой, специализированной для обработки списков или S-выражений.

**) Примечание.* Язык функционального программирования Sisal специализирован для обработки многомерных векторов и организации параллельных процессов, выполняемых на суперкомпьютерах.

Этих правил достаточно, чтобы более ясно выписать **основные тождества** Лиспа, формально характеризующие элементарные функции CAR, CDR, CONS, ATOM, EQ над S-выражениями:

(CAR (CONS x y)) = x
(CDR (CONS x y)) = y
(ATOM (CONS x y)) = Nil
(CONS (CAR x) (CDR x)) = x *для неатомарных x.*
(EQ x x) = T *если x атом*
(EQ x y) = Nil *если x и y различимы*

Любые композиции заданного набора функций над конечным множеством произвольных объектов можно представить таким способом, но класс соответствующих им процессов весьма ограничен и мало интересен. Организация более сложного класса процессов требует более детального представления в программах соответствия между именами и их значениями или определениями, изображения ветвлений и объявления констант.

5) Традиция при изучении функционального программирования избегать знакомства с явными средствами объявления значений переменных ради формирования навыков задания таких значений как аргументов функций малосостоятельна, т.к. изменения переменных легко моделируется переопределением функций без аргументов. Поэтому признаем сразу, что **значение переменной можно объявить** специальной функцией **LET**.

   (Let PI 3.1415926)

6)Обычно в системы программирования встраивают наиболее часто используемые **константы**. Некоторые атомы изначально имеют определенный смысл, например, имена базовых функций, представление пустого списка Nil и тождественно истинная константа T.

В зависимости от контекста одни и те же объекты могут играть роль переменных или констант, причем значения и того, и другого могут быть произвольной сложности. Если объект играет роль константы, то для объявления константы достаточно заблокировать его вычисление, то есть как бы взять его в кавычки (quotation), отмечающие буквально используемые фразы, не требующие обработки. Для такой **блокировки** вводится специальная функция **QUOTE**, предохраняющая свой единственный аргумент от вычисления.

(QUOTE A) - константа A объявлена

(QUOTE (A B C)) - константа (A B C) объявлена

(ATOM (QUOTE A)) = T - аргументом предиката является атом A

(ATOM (QUOTE (A B C))) = Nil

- аргументом предиката является список (A B C)

(ATOM A) - значение не определено, т.к. оно зависит от вхождения переменной A, а ее значение зависит от контекста и должно быть определено или задано до попытки выяснить, атом ли это.

Можно сказать, что функция QUOTE выполняет в древовидной структуре программы роль помеченного контейнера. С ее помощью любое выражение может быть заключено в контейнер, а контейнер помечен указанием, что вычислять его содержимое не следует. Потом, при выполнении функции QUOTE, пометка и контейнер исчезают, и выражение может обрабатываться по общей схеме. Например:

(третий (QUOTE (A B C))) - применение функции “третий” к значению, не требующему вычисления.

7) Некоторые определения функций могут быть хорошо определены на одних аргументах, но зацикливаться на других, подобно традиционному определению факториала при попытке применить его к отрицательным числам. Результат может выглядеть как

исчезновение свободной памяти или слишком долгий счет без видимого прогресса. Такие функции называют **частичными**. Их определения должны включать в себя **ветвления** для проверки аргументов на принадлежность фактической области определения функции - **динамический контроль**. Точнее, вычисление ряда форм в определении может быть обусловлено заранее заданными предикатами.

Ветвление (**условное выражение**) характеризуется тем, что ход процесса зависит от некоторых предикатов (условий), причем условия следует сгруппировать в общий комплект и соотнести с подходящими ветвями. Такую организацию процесса вычисления обеспечивает специальная

функция **COND** (condition), аргументами которой являются двухэлементные списки, содержащие **предикаты и соответствующие им выражения**. Аргументов может быть сколько угодно, а обрабатываются они по особой схеме: сначала вычисляются первые элементы аргументов по порядку, **пока не найдется предикат со значением "истина"**. Затем выбирается второй элемент этого аргумента, и вычисляется его значение, которое и считается значением всего условного выражения.

(COND (p1 e1) (p2 e2) ... (pk ek))

| |_____|_____|_____ ветви условного выражения
|_____|_____|_____ предикаты для выбора ветви

Каждый предикат pi или ветвь ei может быть любой формы: переменная, константа, вызов функции, композиция функций, условное выражение.

Обычное условное выражение (if Predicate Then Else) или (если Predicate то Then иначе Else) может быть представлено с помощью функции COND следующим образом:

(COND (Predicate Then) (T Else))

или более наглядно:

(COND (Predicate Then)

(T Else)

)

Пример 2.2:

(COND ((EQ (CAR x) (QUOTE A))

(CONS (QUOTE B) (CDR x))) (T x))

Атом "T" представляет тождественную истину. Значение всего условного выражения получается путем замены первого элемента из значения переменной x на B в том случае, если (CAR x) совпадает с A.

Содержательно функции QUOTE, COND, LAMBDA и LABEL образуют базовый комплект средств управления программами и процессами, поддерживающий стиль программирования, идеологически близкий **структурному программированию**.[18]

Объявленные здесь **специальные функции** QUOTE, COND, LAMBDA и LABEL существенно отличаются от **элементарных функций** CAR, CDR, CONS, ATOM, EQ правилом обработки аргументов. Обычные функции получают значения аргументов, предварительно вычисленные системой программирования по формулам фактических параметров функции. Специальные функции не требуют такой предварительной

обработки параметров. Они сами могут выполнять все необходимое, используя представление фактических параметров в виде S-выражений.

8) Определения могут быть рекурсивными.

Как правило, рекурсивные вызовы функций должны быть заданы в комплекте с не рекурсивными ветвями процесса. Основное применение условных выражений - **рекурсивные определения** функций.

Пример 2.3:

(LABEL премьер (LAMBDA (x)(COND ((ATOM x) x)

(T (премьер (CAR x))))))

Новая функция “премьер” выбирает первый атом из любого данного. Если x является атомом, то он является результатом, иначе функцию “премьер” следует применить к первому элементу значения x, которое получается в результате вычисления формулы (CAR x). На составных x будет выполняться вторая ветвь, выбираемая по тождественно истинному значению встроенной константы T.

Определение функции “премьер” рекурсивно. Эта функция действительно работает в терминах самой себя. Важно, что для любого S-выражения существует некоторое число применений функции CAR, после которого из этого S-выражения выделится какой-нибудь атом, следовательно, процесс вычисления функции всегда определен, детерминирован и завершится за конечное число шагов. Можно сказать, что для определенности рекурсивной функции следует формулировать **условие ее завершения**.

Введенные обозначения достаточны, чтобы проследить за формированием значений и преобразованием форм в процессе исполнения функциональных программ.

Рассмотрим вычисление формы:

((LABEL премьер (LAMBDA (x)(COND ((ATOM x) x)

(T (премьер (CAR x))))))

(QUOTE ((A . B) . C)))

LABEL дает имена обычным функциям, поэтому фактический параметр функции “премьер” будет вычислен до того, как начнет работать ее определение, и переменная “x” получит значение “((A . B) . C))”.

x = ((A . B) . C))

Таблица 2.5. Схема вывода результата формы с рекурсивной функцией

Вычисляемая форма	Очередной шаг	Результат и комментарии
		Вход в рекурсию
(премьер (QUOTE ((A . B) . C)))	Выбор определения функции и	(COND ((ATOM x) x) (T (премьер (CAR x))))
		Первый шаг рекурсии
	выделение параметров функции	(QUOTE ((A . B) . C))
(QUOTE ((A . B) . C))	Вычисление аргумента	X = ((A . B) . C)

	функции	
(COND ((ATOM x) x) (T (премьер (CAR x))))	Перебор предикатов: выбор первого	(ATOM x)
(ATOM x)	Вычисление первого предиката	Nil = “ложь”, т.к. X – не атом. Переход ко второму предикату
T	Вычисление второго предиката	T = “истина” – константа. Переход к выделенной ветви
		Второй шаг рекурсии
(премьер (CAR x))	выделение параметров функции	(CAR x)
(CAR x)	Вычисление аргумента функции	X = (A . B) Рекурсивный переход к редуцированному аргументу
(COND ((ATOM x) x) (T (премьер (CAR x))))	Перебор предикатов: выбор первого	(ATOM x)
(ATOM x)	Вычисление первого предиката	Nil = “ложь”, т.к. X – не атом. Переход ко второму предикату
T	Вычисление второго предиката	T = “истина” – константа. Переход ко второй ветви
		Третий шаг рекурсии
(премьер (CAR x))	выделение параметров функции	(CAR x)
(CAR x)	Вычисление аргумента функции	X = A Рекурсивный переход к редуцированному аргументу
(COND ((ATOM x) x) (T (премьер (CAR x))))	Перебор предикатов: выбор первого	(ATOM x)
(ATOM x)	Вычисление первого предиката	T – т.к. X теперь атом Преход к первой ветви
X	Вычисление значений переменной	A Значение функции получено и вычисление завершено
		Выход из рекурсии

Условные выражения не менее удобны и для численных расчетов:

Пример 2.4. Абсолютное значение числа.

```
(LABEL Абс  (LAMBDA (x)(COND ((< x 0 ) (- x))
                             (T   x)     )))
```

Пример 2.5. Факториал неотрицательного числа.

```
(LABEL Факториал  (LAMBDA (N)(COND ((= N 0 ) 1 )
                                   (T (*  N (Факториал (- N 1 )))  )  )))
```

Это определение не завершается на отрицательных аргументах.

Работа с **числами, строками** и другим элементарными типами данных обычно строится как включение в язык и систему так называемых **самоопределимых данных**, выглядящих достаточно естественно, чтобы их смысл не требовал особых пояснений. Включается и набор наиболее употребимых базовых операций над такими данными. Если введены числа, то введены и традиционные **арифметические операции**, но форма их применения подчинена общим правилам:

(+ 1 2 3 4) = 10

Функция, определенная лишь для некоторых значений аргументов естественной области определения, называется **частичной функцией**.

Пример 2.6. Алгоритм Евклида для нахождения наибольшего общего делителя двух положительных целых чисел. (остаток [x, y] - функция, вычисляющая остаток от деления x на y.)

```
(LABEL НОД (LAMBDA (x y)(COND ((< x   y) (НОД y x))
                              ((= (остаток y x ) 0 ) x )
                              (T (НОД (остаток y x) x )) )))
```

Подробное изложение теории функций, определяемых рекурсивными выражениями, можно найти у многих математиков, например у А.И.Мальцева, “Алгоритмы и рекурсивные функции”, М. 1965.

Соответствие между именем функции и ее определением можно задать с помощью более удобной специальной псевдо-функции **DEFUN**, первый аргумент которой - имя функции, второй - список ее **формальных параметров**, третий - собственно **тело определения функции**. Формальным результатом DEFUN, как и LABEL, является ее первый аргумент, который также становится объектом другой категории, но теперь это имя новой функции, известной во всей дальнейшей программе.

Пример 2.7:

Определение функции	Комментарии
(DEFUN третий	имя новой функции
(LAMBDA (x)	параметры функции
(CAR (CDR (CDR x)))	тело функции
))	

Новая функция “третий” действует почти так же, как и ранее приведенное определение, но теперь можно функцию рассматривать как **глобальную** и обращаться к ней в форме.

(третий (QUOTE (A B C))) - применение новой функции

Вышеописанные лямбда-обозначения не вполне удобны для именования рекурсивных функций, хотя это возможно. Название функции используется внутри рекурсивных определений, символизируя целое определение. При определении функции “премьер”

удобнее использовать специальную функцию Defun, связывающую название функции и с определяющей формой, и со списком переменных.

(defun премьер (x) (cond ((atom x) x)

(T (премьер (car x)))))

Собственно механизм связывания пока определен не был. Функция defun использует лямбда-конструктор, чтобы представление функции получалось более естественно и использовалось как единая конструкция. Фактически LABEL и DEFUN реализуют аналог тождества:

премьер = (LAMBDA (x) (cond ((atom x) x)

(T (премьер (car x)))))

Роль локального **связывания** имени и определения функции в Лиспе выполняет специальная функция LABEL. Если EF - определение, а NF - его имя, то (LABEL NF EF) - функция, знающая свое имя NF. В форме (DEFUN премьер (x) (cond ((atom x) x) (T (премьер (car x))))) “x” - связанное имя переменной, а “премьер” - связанное имя функции, доступное программе.

Механизм **конструирования определений функций** на базе LABEL более логичен, чем DEFUN: наличие локального механизма формально пригодно и для реализации глобальных связей - они просто должны быть объявлены или расположены раньше всех локальных. Именно так и был формально определен реализационный базис **чистого Лиспа** для раскрутки полной Лисп-системы. Но на практике поначалу удобнее пользоваться функцией defun, отдавая себе отчет в том, что это нечто вроде строительных лесов, помогающих создать более стройное здание. Defun совмещает работу LABEL и LAMBDA - сразу, как и в большинстве языков программирования, позволяет ввести и название функции, и имена ее арументов. Если LABEL строит именованную функцию для ее рекурсивного применения внутри текущего выражения, то DEFUN запоминает определение имени для вызова функции в любом месте программы.

Как и любая форма, любое S-выражение, символьные представления функций могут быть значениями аргументов - **функциональные аргументы**.

Базис элементарного Лиспа образуют пять функций над S-выражениями, CAR, CDR, CONS, ATOM, EQ, и три специальных функции, обеспечивающих управление программами и процессами и конструирование функциональных объектов QUOTE, COND, LAMBDA.

Точные **границы применимости** построений над таким базисом будут определены в следующей лекции в форме определения универсальной функции **EVAL**, позволяющей вычислять значения выражений, представленных в виде списков, - **правило интерпретации выражений**.

Формально для перехода к самостоятельным упражнениям нужна несколько большая определенность по механизмам исполнения программ, представленных S-выражениями:

- аргументы функций, как правило, вычисляются в порядке их перечисления;
- композиции функций выполняются в порядке от самой внутренней функции наружу до самой внешней;
- представление функции анализируется до того, как начинают вычисляться аргументы, т.к. в случае специальных функций аргументы можно не вычислять;
- при вычислении лямбда-выражений **связи между именами переменных и их значениями**, а также между именами функций и их определениями, накапливаются в так называемом **ассоциативном списке**, пополняемом при вызове функции и освобождаемом при выходе из нее.

Лекция 3. **Универсальная функция**

> Изучается техника организации обычных вычислений использованием значений функций и их параметров как альтернативы стандартной программотехнике, основанной на изменениях состояний памяти. Рассматриваются методы управления эффективностью и порядком вычислений, организованных как применение функций к заданным аргументам.. Строится простейшее определение универсальной функции, задающей границы вычисления представленных списками определенных форм над S-выражениями.

Приведенные ранее описания и правила записи функций и выражений в этой лекции должны получить более строгое определение. Начнем с синтаксического обобщения. Подведем итог и представим результаты в виде простых БНФ (Формул Бекуса-Наура):

Синтаксис данных в Лиспе сводится к правилам представления атомов и S-выражений.

<атом> ::= <БУКВА> <конец_атома>

<конец_атома> ::= <пусто>
| <БУКВА> <конец_атома>
| <число> <конец_атома>

В Лиспе атомы - это мельчайшие частицы. Их разложение по литерам не имеет смысла.

<S-выражение> ::= <атом>
| (<S-выражение> . <S-выражение>)
| (<S-выражение> ...)

Данное правило констатирует, что **S-выражения** - это или атомы, или узлы из пары S-выражений, или списки из S-выражений.

/Три точки означают, что допустимо любое число вхождений предшествующего вида объектов, включая ни одного./

Согласно такому правилу () есть допустимое S-выражение. Оно в языке Лисп по соглашению эквивалентно атому Nil.

Базовая система представления данных - точечная нотация, хотя на практике запись в виде списков удобнее. Любой список можно представить точечной нотацией:

() = Nil
(a . Nil) = (a)
- - -
(a1 . (... (aK . Nil) ...)) = (a1 ... aK)

Такая **единая структура данных** оказалась вполне достаточной для представления сколь угодно сложных программ. Дальнейшее определение языка Лисп можно рассматривать как восходящий процесс генерации семантического каркаса, по ключевым позициям которого распределены **семантические действия** по обработке программ.

Другие правила представления данных нужны лишь при расширении и специализации **лексики** языка (числа, строки, имена особого вида и т.п.). Они не влияют ни на общий синтаксис языка, ни на строй его понятий, а лишь характеризуют разнообразие сферы его конкретных приложений.

Синтаксис программ является конкретизацией синтаксиса данных, а именно – выделением из класса S-выражений подкласса **вычислимых выражений** (форм), т.е. данных, имеющих смысл как выражения языка и приспособленных к вычислению. Внешне это выглядит как объявление объектов, заранее известных в языке, и представление разных форм, вычисление которых обладает определенной спецификой.

Выполнение программы устроено как **интерпретация данных**, представляющих выражения, имеющие значение. Ниже приведены синтаксические правила для обычных конструкций, к которым относятся идентификаторы, переменные, константы, аргументы, формы и функции. (Правила упорядочены по сложности взаимосвязи формул.)

<идентификатор> ::= <атом>

Идентификатор - это подкласс атомов, используемых при именовании неоднократно используемых объектов программы - функций и переменных. Предполагается, что идентифицируемые объекты размещаются в памяти так, что по идентификатору их можно найти.

> Понятие "идентификатор" выделено для того, чтобы по мере развития определения атома не требовалось на все виды атомов искусственно распространять семантику вычислений. (Например, у Бекуса в его знаменитой статье про “бутылочное горлышко” числа рассматриваются как операции доступа.)

<форма> ::= <константа>
| <переменная>
| (<функция> <аргумент> ... >)
| (COND (<форма> <форма>) (<форма> <форма>) ...)

<константа> ::= (QOUTE <S-выражение>)
| '<S-выражение>

<переменная> ::= <идентификатор>

Переменная - это подкласс идентификаторов, которым сопоставлено многократно используемое значение, ранее вычисленное в подходящем контексте. Подразумевается, что одна и та же переменная в разных контекстах может иметь разные значения.

Таким образом, **класс форм** - это объединение класса переменных и подкласса списков, начинающихся с QUOTE, COND или с представления некоторой функции.

<аргумент> ::= <форма>

Форма - это выражение, которое может быть вычислено.

Форма, представляющая собой **константу**, выдает эту константу как свое значение. В таком случае нет необходимости в вычислениях, независимо от вида константы. **Константные значения** могут быть любой сложности, включая вычислимые выражения. Чтобы избежать двусмысленности, предлагается константы изображать как результат специальной функции **QUOTE, блокирующей вычисление**. Представление констант с помощью QOUTE устанавливает границу, далее которой вычисление не идет. Использование апострофа (') - просто сокращенное обозначение для удобства набора внешних форм. Константные значения аргументов характерны при тестировании и демонстрации программ.

Если форма представляет собой переменную, то ее значением должно быть S-выражение, связанное с этой переменной до момента вычисления формы. (**Динамическое связывание**, в отличие от традиционного правила, требующего связывания к моменту описания формы, т.е. статическое связывание.)

Третья ветвь определения гласит, что можно написать **функцию, затем перечислить ее аргументы,** и все это как общий список заключить в скобки.

Аргументы представляются формами. Это означает, что допустимы **композиции функций.** Обычно **аргументы вычисляются в порядке вхождения** в список аргументов. **Позиция** "аргумент" выделена для того, чтобы было удобно в дальнейшем локализовывать разные схемы обработки аргументов в зависимости от категории функций. Аргументом может быть любая форма, но метод вычисления аргументов может варьироваться. Функция может не только учитывать тип обрабатываемого данного, но и управлять временем обработки данных, принимать решения по глубине и полноте анализа данных, обеспечивать продолжение счета при исключительных ситуациях и т.п.

Последняя ветвь определяет формат условного выражения. Согласно этому формату **условное выражение** строится из размещенных в двухэлементном списке синтаксически различимых позиций для **пропозициональных термов** и обычных форм. Двухэлементные списки из определения условного выражения рассматриваются как представление предиката и соответствующего ему S-выражения. Значение условного выражения определяется перебором **предикатов по порядку**, пока не найдется форма, значение которой отлично от Nil, что означает логическое значение "истина". Строго говоря, такая форма должна быть найдена непременно. Тогда вычисляется S-выражение, размещенное вторым элементом этого же двухэлементного списка. Остальные предикаты и формы условного выражения не вычисляют (**логика Мак-Карти**), их формальная корректность или определенность не влияют на существование результата.

Разница между пропозициональными и обычными формами заключается лишь в трактовке их результатов. Любая форма может играть роль предиката.

<функция> ::= <название>
| (LAMBDA <список_переменных> <форма>)
| (LABEL <название> <функция>)

<список_переменных> ::= (<переменная> ...)

<название> = <идентификатор>

Название - это подкласс идентификаторов, определение которых хранится в памяти, но оно может не подвергаться влиянию **контекста вычислений**.

Таким образом, **класс функций** - это объединение класса названий и подкласса трехэлементных списков, начинающихся с LAMBDA или LABEL.

Функция может быть представлена просто именем. В таком случае ее смысл должен быть заранее известен. Функция может быть введена с помощью **лямбда-выражения**, устанавливающего соответствие между аргументами функции и **связанными переменными**, упоминаемыми в теле ее определения (в определяющей ее форме). Форма из определения функции может включать переменные, не включенные в лямбда-список, - так называемые **свободные переменные**. Их значения должны устанавливаться на более внешнем уровне. Если функция рекурсивна, то следует объявить ее имя с помощью **специальной функции** LABEL. (Используемая в примерах DEFUN, по существу, совмещает эффекты LABEL и LAMBDA.)

Таблица 3.1. Синтаксическая сводка языка Лисп.

<форма> ::= <переменная>
| (QOUTE <S-выражение>)
| (COND (<форма> <форма>) ... (<форма> <форма>))
| (<функция> <аргумент> ... <аргумент>)

<аргумент> ::= <форма>

<переменная> ::= <идентификатор>

<функция> ::= <название>
| (LAMBDA <список_переменных> <форма>)
| (LABEL <название> <функция>)

<список_переменных> ::= (<переменная> ...)

<название> = <идентификатор>

<идентификатор> ::= <атом>

<S-выражение> ::= <атом>
| (<S-выражение> . <S-выражение>)
| (<S-выражение> ...)

<атом> ::= <БУКВА> <конец_атома>

<конец_атома> ::= <пусто>
| <БУКВА> <конец_атома>
| <число> <конец_атома>

Универсальная функция

Интерпретация или универсальная функция - это функция, которая может вычислять значение любой формы, включая формы, сводимые к вычислению произвольной заданной функции, применяемой к аргументам, представленным в этой же форме, по доступному описанию данной функции. (Конечно, если функция, которой предстоит

интерпретироваться, имеет бесконечную рекурсию, интерпретация будет повторяться бесконечно.)

Определим универсальную функцию eval от аргумента expr - выражения, являющегося произвольной вычислимой формой языка Лисп.

Универсальная функция должна предусматривать основные виды вычисляемых форм, задающих значения аргументов, а также представления функций, в соответствии со сводом приведенных выше правил языка. При интерпретации выражений учитывается следующее:

- **Атомарное выражение** обычно понимается как переменная. Для него следует найти связанное с ним значение. Например, могут быть переменные вида “x”, “elem”, смысл которых зависит от контекста, в котором они вычисляются.

- **Константы**, представленные как аргументы функции QUOTE, можно просто извлечь из списка ее аргументов. Например, значением константы (QUOTE T) является атом T, обычно символизирующий значение "истина".

- **Условное выражение требует специального алгоритма** для перебора предикатов и выбора нужной ветви. Например, интерпретация условного выражения

```
(COND
        ((ATOM x)   x)
        ((QUOTE T) (first (CAR x) ))
)
```

должна обеспечивать выбор ветви в зависимости от атомарности значения аргумента. Семантика чистого Лиспа не определяет значение условного выражения при отсутствии предиката со значением "истина". Но во многих реализациях и диалектах Лиспа такая ситуация не рассматривается как ошибка, а значением считается Nil. Иногда это придает условным выражениям лаконичность.

- Остальные формы выражений рассматриваются по общей схеме как **список из функции и ее аргументов**. Обычно аргументы вычисляются, а затем вычисленные значения передаются функции для интерпретации ее определения. Так обеспечивается возможность писать композиции функций. Например, в выражении (first (CAR x)) внутренняя функция CAR сначала получит в качестве своего аргумента значение переменной x, а потом свой результат передаст как аргумент более внешней функции first.

- Если функция представлена своим названием, то среди названий различаются имена **встроенных функций**, такие как CAR, CDR, CONS и т.п., и имена функций, введенных в программе, например first. Для встроенных функций интерпретация сама «знает», как найти их значение по заданным аргументам, а для введенных в программе функций - использует их **определение**, которое находит по имени.

- Если функция использует **лямбда-конструктор**, то прежде чем его применять, **понадобится связывать переменные из лямбда-списка со значениями аргументов**. Функция, использующая
лямбда-выражение,

```
(LAMBDA (x)
      (COND
            ((ATOM x)   x)
            ((QUOTE T) (first (CAR x) ))
      )
)
```

зависит от одного аргумента, значение которого должно быть связано с переменной x. В определении используется **свободная функциональная переменная** first, которая должна быть определена в более **внешнем контексте**.

- Если представление функции начинается с LABEL, то **понадобится сохранить имя функции с соответствующим ее определением** так, чтобы корректно выполнялись рекурсивные вызовы функции. Например, предыдущее LAMBDA-определение безымянной функции **становится рекурсивным**, если его сделать вторым аргументом специальной функции LABEL, первый аргумент которой – fisrt, имя новой функции.

```
(LABEL first
      (LAMBDA (x)
            (COND
                  ((ATOM x)    x)
                  ((QUOTE T) (first (CAR x) ))
)    )      )
```

Таким образом, интерпретация функций осуществляется как **взаимодействие** четырех подсистем:

- обработка структур данных (cons, car, cdr, atom, eq);
- конструирование функциональных объектов (lambda, label);
- идентификация объектов (имена переменных и названия функций);
- управление логикой вычислений и **границей вычислимости** (композиции, quote, cond, **eval**).

В большинстве языков программирования аналоги первых двух подсистем нацелены на обработку элементарных данных и конструирование составных значений, кроме того, иначе установлены границы между подсистемами.

Прежде чем дать определение универсальной функции, опишем ряд **дополнительных функций**, полезных при обработке S-выражений. Некоторые из них пригодятся при определении интерпретатора. Воспользуемся при введении новых функций псевдо-функцией **DEFUN**, аргументы которой – название функции, список ее параметров и определяющая форма. **Символ «;» - начало строчного коментария.**

Начнем с общих методов обработки S-выражений.

AMONG – проверка, **входит ли заданный атом в данное** S-выражение.

```
(DEFUN among  (x y) (COND
                        ((ATOM y) (EQ x y))
                        ((among x (CAR y)) (QUOTE T))
                        ((QUOTE T) (among x (CDR y) ))
```

))

EQUAL - предикат, проверяющий **равенство двух S-выражений**. Его значение "истина" для идентичных аргументов и "ложь" для различных. (Элементарный предикат EQ определен только для атомов.) Определение EQUAL иллюстрирует условное выражение внутри условного выражения (двухуровневое условное выражение и двунаправленная рекурсия).

```
(DEFUN equal (x y) (COND
                    ((ATOM x) (COND
                            ((ATOM y) (EQ x y))
                            ((QUOTE T) (QUOTE NIL))
                           )
                    )
                    ((equal (CAR x)(CAR y)) (equal (CDR x)(CDR y)))
                    ((QUOTE T) (QUOTE NIL))
))
```

SUBST - функция трех аргументов x, y, z, строящая результат **замены** S-выражением x всех вхождений атома y в S-выражение z.

```
(DEFUN subst (x y z) (COND
                      ((equal y z) x)
                      ((ATOM z) z)
                      ((QUOTE T)(CONS
                              (subst x y (CAR z))
                              (subst x y (CDR z))

))))

(subst '(x . A) 'B '((A . B) . C))        ;= ((A . (x . A)) . C)
```

Использование equal в этом определении позволяет осуществлять подстановку и в более сложных случаях. Например, для редукции совпадающих хвостов подсписков:

```
(subst 'x '(B C D) '((A B C D)(E B C D)(F B C D)))          ;= ((A . x)(E . x)(F . x))
```

NULL - предикат, **отличающий пустой список** от остальных S-выражений. Позволяет выяснять, когда список исчерпан. Принимает значение "истина" тогда и только тогда, когда его аргумент - Nil.

```
(DEFUN null (x) (COND
                   ((EQ x (QUOTE Nil)) (QUOTE T))
                  ((QUOTE T)  (QUOTE Nil))
))
```

При необходимости можно компоненты точечной пары разместить в двухэлементном списке, и наоборот, из первых двух элементов списка построить в точечную пару.

```
(DEFUN pair_to_list  (x)  (CONS (CAR x) (CONS (CDR x) Nil)) )
```

```
(DEFUN list_to_pair  (x)  (CONS (CAR x) (CADR x)) )
```

По этим определениям видно, что списочная запись строится большим числом CONS, т.е. на нее расходуется больше памяти.

Основные методы обработки списков

Следующие функции используются, когда **рассматриваются лишь списки**.

APPEND - функция двух аргументов x и y, **сцепляющая два списка** в один.

```
(DEFUN append (x y) (COND
                    ((null x) y)
                    ((QUOTE T) (CONS
                            (CAR x)
                            (append (CDR x) y)
))))

(append '(A B) '(C D E))              ;= (A B C D E)
```

MEMBER - функция двух аргументов x и y, выясняющая, **встречается ли S-выражение** x среди элементов списка y.

```
(DEFUN member (x y) (COND
                    ((null x) (QUOTE Nil))
                    ((equal x (CAR y)) (QUOTE T))
                    ((QUOTE T) (member x (CDR y))
))
```

PAIRLIS - функция аргументов x, y, al **строит список пар** соответствующих элементов из списков x и y и присоединяет их к списку al. Полученный список пар, похожий на таблицу с
двумя столбцами, называется ассоциативным списком или ассоциативной таблицей.
Такой список может использоваться для связывания имен переменных и функций при организации
вычислений интерпретатором.

```
(DEFUN pairlis (x y al) (COND
                        ((null x) al)
                        ((QUOTE T) (CONS (CONS (CAR x)
                                               (CAR Y) )
                                         (pairlis (CDR x)
                                                  (CDR y)
                                                  al)


)))

(pairlis '(A B C) '(u t v) '((D . y)(E . y)))          ; = ((A . u)(B . t)(C . v)(D . y)(E . y))
```

ASSOC - функция двух аргументов, x и al. Если al - **ассоциативный список**, подобный тому, что формирует функция pairlis, то assoc выбирает из него первую пару,

начинающуюся с x. Таким образом, это функция **поиска определения или значения по таблице**, реализованной в форме ассоциативного списка.

```
(DEFUN assoc (x al) (COND
                    ((equal x (CAAR al)) (CAR al))
                    ((QUOTE T) (assoc x (CDR al))
))
```

Частичная функция - рассчитана на наличие ассоциации.

```
(assoc 'B '((A . (m n)) (B . (CAR x)) (C . w) (B . (QUOTE T))))          ; = (B . (CAR x))
```

SUBLIS - функция двух аргументов al и y, предполагается, что первый из аргументов AL устроен как ассоциативный список вида ((u1 . v1) ... (uK . vK)), где u есть атомы, а второй аргумент Y - любое S-выражение. Действие sublis заключается в обработке Y, такой, что **вхождения переменных** Ui, связанные в ассоциативном списке со значениями Vi, **заменяются на** эти **значения**. Другими словами в S-выражении Y вхождения переменных U заменяются на соответствующие им V из списка пар AL. **Вводим вспомогательную функцию** SUB2, обрабатывающую атомарные S-выражения, а затем - полное определение SUBLIS:

```
(DEFUN sub2 (al z) (COND
                   ((null al) z)
                   ((equal (CAAR al) z) (CDAR al))
                   ((QUOTE T) (sub2 (CDR al) z))
                  ) )


(DEFUN sublis (al y) (COND
                    ((ATOM y) (sub2 al y))
                    ((QUOTE T)(CONS
                            (sublis al (CAR y))
                            (sublis al (CDR y))
                           ) )))

(sublis '((x . Шекспир)(y . (Ромео и Джульетта))) '(x написал трагедию y))
;= (Шекспир написал трагедию (Ромео и Джульетта))
```

INSERT – **вставка z перед вхождением ключа** x в список al.

```
(DEFUN insert (al x z) (COND
                    ((null al) Nil)
                    ((equal (CAR al) x) (CONS z al))
                    ((QUOTE T) (CONS (CAR al) (insert (CDR al) x z)))
))

(insert ‘(a b c) ‘b ‘s)                    ; =  (a s b c)
```

ASSIGN – **модель присваивания переменным**, хранящим значения в ассоциативном списке. Замена связанного с данной переменной в первой паре значения на новое заданное значение. Если пары не было вообще, то новую пару из переменной и ее значения помещаем в конец a-списка, чтобы она могла работать как глобальная.

```
(DEFUN assign (x v al) (COND
                        ((Null al) (CONS (CONS x v) Nil ))
                       ((equal x (CAAR al))(CONS (CONS x v) (CDR al)))
                       ((QUOTE T) (CONS (CAR al) (assign x v (CDR al))))
))

(assign ‘a 111 ‘((a . 1)(b . 2)(a . 3)))          ;= ((a . 111)(b . 2)(a . 3))
(assign ‘a 111 ‘((c . 1)(b . 2)(a . 3)))          ;= ((c . 1)(b . 2)(a . 111))
(assign ‘a 111 ‘((c . 1)(d . 3)))                 ;= ((c . 1)(d . 3) (a . 111))
```

REVERSE – **обращение списка –** два варианта, второй с накапливающим параметром и вспомогательной функцией:

```
(defun reverse (m) (cond ((null m) NIL)
                         (T (append(reverse(cdr m))
                                   (list(car m)) )) ))

(defun reverse (m) (rev m Nil))
(defun rev (m n) (cond ((null m) N)
                       (T (rev(cdr m) (cons (car m) n)))))
```

Определение универсальной функции

Универсальная функция **eval**, которую предстоит определить, должна удовлетворять следующему условию: **если представленная аргументом форма сводится к функции, имеющей значение на списке аргументов этой же формы, то данное значение и является результатом функции eval.**

```
(eval '(fn arg1 ... argK))        ;= результат применения fn к аргументам arg1, ..., argK.
```

Явное определение такой функции позволяет достичь четкости механизмов обработки Лисп-программ.

```
(eval ‘((LAMBDA (x y) (CONS (CAR x) y))
       ‘(A B) ‘(C D) ))
; = (A C D)
```

Вводим **две основные функции eval и apply** для обработки форм и обращения к функциям, соответственно. Каждая из этих функций **использует ассоциативный список для хранения связанных имен - значений переменных и определений функций**. Сначала этот список пуст. Вернемся к синтаксической сводке вычислимых форм.

```
<форма> ::= <переменная>
        | (QOUTE <S-выражение>)
        | (COND (<форма> <форма>) ... (<форма> <форма>))
        | (<функция> <аргумент> ...)

<аргумент> ::= <форма>

<переменная> ::= <идентификатор>

<функция> ::= <название>
```

| (LAMBDA <список_переменных> <форма>)
| (LABEL <название> <функция>)

<список_переменных> ::= (<переменная> ...)

<название> = <идентификатор>

<идентификатор> ::= <атом>

Каждой ветви этой сводки соответствует ветвь универсальной функции:

```
(DEFUN eval0 (e)   (eval e '((Nil . Nil) (T . T))))
```

Вспомогательная функция eval0 понадобилась, чтобы в eval ввести **накапливающий параметр** – ассоциативный список, в котором будут храниться связи между переменными и их значениями и названиями функций и их определениями.

```
(DEFUN eval(e a) (COND
                        ( (atom e)  (cdr(assoc e a)) )
                        ( (eq (car e) 'QUOTE)  (cadr e))
                        ( (eq(car e) 'COND)    (evcon(cdr e) a))
                        ( T  (apply (car e) (evlis(cdr e) a) a) ))    )

(defun apply (fn x a) (COND
              ((atom fn)(cond
                          ((eq fn 'CAR)(caar x))
                          ((eq fn 'CDR)(cdar x))
                          ((eq fn 'CONS) (cons (car x)(cadr x)) )
                          ((eq fn 'ATOM)(atom (car x)) )
                          ((eq fn 'EQ) (eq (car x)(cadr x)) )
                          (T (apply (eval fn a) x a)) ) )
               )
((eq(car fn)'LAMBDA) (eval (caddr fn)
                           (pairlis (cadr fn) x a) ))
((eq (car fn) 'LABEL) (apply (caddr fn) x
                           (cons (cons (cadr fn)(caddr fn)) a)))))
```

assoc и pairlis уже определены ранее.

```
(DEFUN evcon (c a) (COND
                        ((eval (caar  c) a) (eval (cadar c) a) )
                        ( T  (evcon (cdr c) a)     ) ))
```

**) Примечание.* Не допускается отсутствие истинного предиката, т.е. пустого С.

```
 (DEFUN evlis (m a) (COND
                        ((null m) Nil )
                        ( T         (cons(eval (car m) a)
                                         (evlis(cdr m) a)    ))   )
```

При
```
(DEFUN eval0 (e)  (eval e ObList ))
```

определения функций могут накапливаться в системной переменной ObList, то есть работать как глобальные определения. ObList обязательно должна содержать глобальное определение встроенной константы Nil, можно и сразу разместить в ней T.

Поясним ряд пунктов этих определений.

Первый аргумент eval - форма. Если она - атом, то этот атом может быть только именем переменной, а **значение переменной должно уже находиться в ассоциативном списке**.

Если CAR от формы - QUOTE, то она представляет собой **константу**, значение которой **вычисляется как CADR от нее самой**.

Если CAR от формы - COND, то форма - условное выражение. **Вводим вспомогательную функцию EVCON** (определение ее будет дано ниже), которая обеспечивает вычисление предикатов (пропозициональных термов) по порядку и выбор формы, соответствующей первому предикату, принимающему значение "истина". Эта форма передается EVAL для дальнейших вычислений.

Все остальные случаи рассматриваются как **список из функции с последующими аргументами**.

Вспомогательная функция EVLIS обеспечивает вычисление аргументов, затем представление функции и список вычисленных значений аргументов передаются функции APPLY.

Первый аргумент apply - функция. Если она - атом, то существует две возможности. Атом может представлять **одну из элементарных функций (car cdr cons atom eq)**. В таком случае соответствующая ветвь вычисляет значение этой функции на заданных аргументах. В противном случае, этот атом - **имя ранее заданного определения**, которое можно найти в ассоциативном списке, подобно вычислению переменной.

Если функция начинается с LAMBDA, то ее **аргументы попарно соединяются со связанными переменными**, а тело определения (форма из лямбда-выражения) передается как аргумент функции eval для дальнейшей обработки.

Если функция начинается с LABEL, то ее **название и определение соединяются в пару, и полученная пара размещается в ассоциативном списке**, чтобы имя функции стало определенным при дальнейших вычислениях. Они произойдут как рекурсивный вызов apply, которая вместо имени функции теперь работает с ее определением при более полном ассоциативном списке - в нем уже размещено определение имени функции. Поскольку определение размещается "наверху" стека, оно становится доступным для всех последующих переопределений, то есть работает как **локальный объект**. Глобальные объекты, такие как обеспечиваются псевдо-функцией DEFUN, устроены немного иначе, что будет рассмотрено в следующей лекции.

Определение универсальной функции является важным шагом, показывающим **одновременно и механизмы реализации функциональных языков, и технику функционального программирования на любом языке**. Пока еще не описаны многие другие особенности языка Лисп и функционального программирования, которые будут рассмотрены позднее. Но все они будут увязаны в единую картину, основа которой согласуется с этим определением.

1) В **строгой теории аппликативного программирования** все функции следует определять всякий раз, когда они используются. На практике это неудобно. Реальные системы имеют большой запас встроенных функций (более тысячи в Clisp-е), известных языку, и возможность присоединения такого количества новых функций, какое понадобится.

2) В чистом языке Лисп базисные функции CAR и CDR не определены для атомарных аргументов. Такие функции, имеющие осмысленный результат не на всех значениях естественной области определения, называют частичными. Отладка и применение частичных функций требует большего контроля, чем работа с тотальными, всюду определенными функциями. Во многих реализациях функциональных языков программирования все функции всегда вырабатывают значение. При необходимости каждый существенный класс объектов пополняется значением класса ERROR, символизирующим исключительные ситуации.

Во многих реализациях Лиспа все элементарные функции вырабатывают результат и на списках, и на атомах, но его смысл зависит от системных решений, что может создавать трудности при переносе программ на другие системы. Базисный предикат EQ всегда имеет значение, но смысл его на неатомарных аргументах будет более ясен после знакомства со структурами данных, используемыми для представления списков в машине.

3) Для функциональных языков характерно большое разнообразие условных форм, конструкций выбора, ветвлений и циклов, практически без ограничений на их комбинирование. Форма COND выбрана для начального знакомства как наиболее общая. За редким исключением в Лиспе нет необходимости писать в условных выражениях (QUOTE T) или (QUOTE NIL). Вместо них используются **встроенные константы** T и Nil, соответственно.

4) В реальных системах функционального программирования **обычно поддерживается работа с целыми, дробными и вещественными числами в предельно широком диапазоне**, а также работа с кодами и строками. Такие данные, как и атомы, являются минимальными объектами при обработке информации, но отличаются от атомов тем, что их смысл задан непосредственно их собственным представлением. Их понимание не требует ассоциаций или связывания. Поэтому и константы такого вида не нуждаются в префиксе в виде апострофа.

Приведенное выше **самоопределение Лисп-интерпретации является концептуальным минимумом**, обеспечивающим постепенность восприятия более сложных особенностей специфики функционального программирования, а также методов реализации языков и систем программирования, которые будут иллюстрироваться в дальнейшем. Они будут введены как расширения или уточнения чистого определения универсальной функции.

3.3. Предикаты и истинность в Лиспе

Хотя формальное правило записи программ вычислений в виде S-выражения предписывает, что константа T - это (QUOTE T), было оговорено, что в системе всегда пишется T. Кроме того, Nil оказался удобнее, чем атом F, встречавшийся в начальных предложениях по Лиспу аналог значения FALSE.

В Лисп есть два атомных символа, которые представляют **истину и ложь,** соответственно. Эти два атома - **T и NIL**. Данные символы - действительные значения

всех предикатов в системе. Главная причина в удобстве кодирования. Во многих случаях **достаточно отличать произвольное значение от пустого списка**.

(eval T NIL) = T

Формы (QUOTE T) и (QUOTE NIL) будут также работать, потому что:

(eval (QUOTE T) NIL) = T
(eval (QUOTE NIL) NIL) = NIL

Заметим, что

(eval (QUOTE T) alist) = T

будет работать при любом **alist** в силу причин, которые объясняются в лекции 6.

Формального различия между функцией и предикатом в Лиспе не существует. Предикат может быть определен как функция со значениями либо T либо NIL. Это верно для всех предикатов системы. Можно использовать форму, не являющуюся предикатом там, где требуется предикат: предикатная позиция условного выражения или аргумент логического предиката. Семантически любое S-выражение, только не NIL, будет рассматриваться в таком случае как истинное. Первое следствие из этого - предикат Null и логическое отрицание идентичны. Второе - то, что (QUOTE T) или (QUOTE X) практически эквивалентны T как константные предикаты.

Предикат EQ ведет себя следующим образом:

1) Если его аргументы различны, значением EQ является NIL.

2) Если оба его аргументы являются одним и тем же атомом, то значение - T.

3) Если значения одинаковы, но не атомы, то его значение T или NIL в зависимости от того, **идентично ли представление аргументов в памяти**.

4) Значение EQ всегда T или NIL. Оно никогда не бывает не определено, даже если аргументы плохие.

Универсальная функция - это джин, выпущенный из бутылки, т.к. потенциал такой функции ограничен лишь способностями нашего воображения. Пока рассмотрены лишь простейшие, самые очевидные следствия из возможности явно применять и уточнять механизмы символьного представления и определения функций, такие как использование **накапливающих параметров**, **связывание** обозначений по схеме знак-смысл в ассоциативном списке как имен переменных со значениями, так и названий функций с определениями, а также применение **вспомогательных функций** для достижения прозрачности определений. **Эти возможности имеют место в любом языке высокого уровня**. Но попутно выполнено достаточно строгое построение совершенно **формальной математической системы**, называемой “**Элементарный ЛИСП**”. Составляющие этой формальной системы следующие:

1) Множество символов, называемых S-выражениями.

2) Система функциональных обозначений для основных понятий, необходимых при программировании обработки S-выражений.

3) Формальное представление функциональных обозначений в виде S-выражений.

4) Универсальная функция (записанная в виде S-выражения), интерпретирующая обращение произвольной функции, записанной как S-выражение, к ее аргументам.

5) Система базовых функций, обеспечивающих техническую поддержку обработки S-выражений, и специальных функций, обеспечивающих управление вычислениями.

Более интересные и не столь очевидные следствия возникают при расширении этой формальной системы, что и будет продемонстрировано в следующих лекциях.

Лекция 4. **Отображения и функционалы**

> Программирование отображений и использование функционалов демонстрируется как метод резкого повышения производительности программирования и эффективности отладки программ. Изучается механизм безымянных определений. Рассматриваются разные схемы отображений аргументов и формирования результатов на основе отображающих функций над компонентами структур данных и определения различных функциональных схем переработки данных.

Отображения структур данных и функционалы

Отображения обычно используются при анализе и обработке данных, представляющих информацию разной природы. Вычисление, кодирование, трансляция, распознавание - каждый их таких процессов использует исходное множество цифр, шаблонов, текстов, идентификаторов, по которым конкретная отображающая функция находит пронумерованный объект, строит закодированный текст, выделяет идентифицированный фрагмент, получает зашифрованное сообщение. Таким образом работает любое введение обозначений - от знака происходит переход к его смыслу.

Отображения - ключевой механизм информатики. Построение любой информационной системы сопровождается определением и реализацией большого числа отображений. Сначала выбираются данные, с помощью которых представляется информация. В результате по данным можно восстановить представленную ими информацию - извлечь информацию из данных (по записи числа восстановить его величину). Потом конструируется набор структур, достаточный для размещения и обработки данных и программ в памяти компьютера (по коду команды можно выбрать хранимую в памяти подпрограмму, которая построит новые коды чисел или структур данных).

Говорят, что **отображение** существует, если задана пара множеств и **отображающая функция**, для которой первое множество - область определения, а второе - область значения. При определении отображений, прежде всего, должны быть ясны следующие вопросы:

- что представляет собой отображающая функция;
- как организовано данное, представляющее отображаемое множество;
- каким способом выделяются элементы отображаемого множества, передаваемые в качестве аргументов отображающей функции.

Это позволяет задать порядок перебора множества и метод передачи аргументов для вычисления отображающей функции. При обходе структуры, представляющей множество, отображающая функция будет применена к каждому элементу множества. Проще всего выработать структуру множества результатов, подобную исходной структуре. Но возможно не все полученные результаты нужны или требуется собрать их в иную структуру, поэтому целесообразно прояснить заранее еще ряд вопросов:

- где размещается множество полученных результатов;
- чем отличаются нужные результаты от полученных попутно;
- как строится итоговое данное из отобранных результатов.

При функциональном стиле программирования ответ на каждый из таких вопросов может быть дан в виде отдельной функции, причем роль каждой функции в схеме реализации отображения четко фиксирована. Схема реализации отображения может

быть представлена в виде определения, формальными параметрами которого являются обозначения функций, выполняющих эти роли. Такое определение называется "функционал". Более точно, **функционал может оперировать функциями** в качестве аргументов или результатов.

Функции, выполняющие конкретные роли, могут быть достаточно общими, полезными при определении разных отображений, - они получают имена для многократного использования в разных системах определений. Но могут быть и разовыми, нужными лишь в данном конкретном случае - тогда можно обойтись без их имен, использовать определение непосредственно в точке вызова функции.

Таким образом, определение отображения может быть разбито на части (функции и функционалы) разного назначения, типичного для многих схем информационной обработки. Это позволяет упрощать отладку систем определений, повышать коэффициент повторного использования отлаженных функций. Можно сказать, что отображения - эффективный механизм абстрагирования, моделирования, проектирования и формализации крупномасштабной обработки информации. Возможности отображений в информатике значительно шире, чем освоено практическим программированием, но их применение требует дополнительных пояснений, которые и являются предметом этой лекции.

Числа и мультиоперации

Любую информацию можно представить в виде символьных выражений.
В качестве основных видов символьных выражений выбраны списки и атомы.
Атом - неделимое данное, представляющее информацию произвольной природы.
Во многих случаях знание природы информации дает более четкое понимание особенностей изучаемых механизмов. Программирование работы с числами и строками – привычная, хорошо освоенная область информационной обработки, удобная для оценки преимуществ использования функционалов. Опуская технические подробности, просто отметим, что числа и строки рассматриваются как **самоопределимые атомы**, смысл которых не требует никакого ассоциирования, он понятен просто по виду записи.

Например, **натуральные числа** записываются без особенностей и могут быть почти произвольной длины:

1
-123
98765432100000000000000123456789

Можно работать с **дробными и вещественными** числами:

2/3
3.1415926

Строки заключаются в обычные двойные кавычки:

"строка любой длины и из любых символов, включая что угодно".

Список - составное данное, первый элемент которого может рассматриваться как функция, применяемая к остальным элементам, также представленным как символьные выражения. Это относится и к операциям над числами и строками:

```
(+ 1 2 3 4 5 6)      ;= 21
(- 12  6   3)        ;= 3
(/  3 5)             ;= 3/5
(1+  3)              ;= 4
```

Большинство операций над числами при префиксной записи естественно рассматривать как **мультиоперации** от произвольного числа аргументов.

```
(string-equal  "строка  1"  " строка1")   ;= Nil
(ATOM "a+b-c")                            ;= T
(char  "стр1" 4 )                         ;= "1"
```

Со строками можно при необходимости работать **посимвольно**, хотя они рассматриваются как атомы.

Любой список можно превратить в константу, поставив перед ним "'" апостроф. Это эквивалентно записи со специальной функцией "QUOTE". Для чисел и строк в этом нет необходимости, но это не запрещено

```
‘1          ;= 1
‘"abc"     ;=  "abc"
```

Можно строить композиции функций. Ветвления представлены как результат специальной функции COND, использующей отличие от Nil в качестве значения “истина”. Числа и строки таким образом оказываются допустимыми представлениями значения “истина”.

Отказ от барьера между представлениями функций и значений дает возможность символьные выражения использовать как для изображения заданных значений, включая любые структуры над числами и строками, так и для определения функций, обрабатывающих любые данные. (Напоминаем, что определение функции - данное.) **Функционалы** - это функции, которые используют в качестве аргументов или результатов другие функции. При построении функционалов **переменные могут играть роль имен функций**, определения которых находятся во внешних формулах, использующих функционалы.

Функционалы - общее понятие

Рассмотрим технику использования функционалов на упражнениях с числами и наметим, как от простых задач перейти к более сложным.

Пример 4.1. Для каждого числа из заданного списка получить следующее за ним число и все результаты собрать в список.

```
(defun next (xl)                  ; Следующие числа:
    (cond                         ; пока список не пуст
      (xl (cons (1+ (car xl))     ; прибавляем 1 к его голове
            (next (cdr xl))       ; и переходим к остальным,
)   ) ) )                         ; собирая результаты в список

(next '(1 2 5 ))    ; = (2 3 6 )
```

Примечание *) Символ “;” отделяет **комментарий**, расположенный до конца строки
Пример 4.2. Построить список из "голов" элементов списка

```
(defun 1st (xl)                 ; "головы" элементов = CAR
   (cond                        ; пока список не пуст
     (xl (cons (caar xl)        ; выбираем CAR от его головы
               (1st (cdr xl))   ; и переходим к остальным,
)   ) ) )                      ; собирая результаты в список

(1st '((один два )(one two )(1 2 )) )          ; = (один one 1)
```

Пример 4.3. Выяснить **длины** элементов списка

```
(defun lens (xl)                ; Длины элементов
   (cond                        ; Пока список не пуст
     (xl (cons (length (car xl)) ; вычисляем длину его головы
               (lens (cdr xl))   ; и переходим к остальным,
)   ) ) )                      ; собирая результаты в список

(lens '((1 2 ) () (a b c d ) (1 (a b c d ) 3 )) )  ; = (2 0 4 3 )
```

Внешние отличия в записи этих трех функций малосущественны, что позволяет ввести более общую функцию map-el, в определении которой имена "car", "1+" и "lenth" могут быть заданы как значения параметра fn:

```
(defun map-el (fn xl) ; Поэлементное преобразование XL
                      ; с помощью функции FN
   (cond                        ; Пока XL не пуст
     (xl (cons (funcall fn (car xl) )
               ; применяем FN как функцию голове XL[1]

               (map-el fn (cdr xl))   ; и переходим к остальным,
)   ) ) )                      ; собирая результаты в список
```

Эффект функций next, 1st и lens можно получить выражениями:

```
(map-el #'1+ xl)          ; Следующие числа:
(map-el #'car xl)         ; "головы" элементов = CAR
(map-el #'length xl)      ; Длины элементов
```

Примечание: #’ – префикс функции-значения.

---

[1] Примечание: На Lisp 1.5 это определение выглядит изящнее, не требует встроенной функции funcall, необходимой в случае Clisp-а:

```
(defun map-el (fn xl)
   (cond
     (xl (cons (fn (car xl) )          ; применяем первый аргумент как функцию
                                       ; к первому элементу второго аргумента
               (map-el fn (cdr xl) )
)   ) ) )
```

```
(map-el #'1+   '(1 2 5 ))   ; = (2 3 6 )
(map-el #'car  '((один два )(one two )(1 2 )) )  ; = (один one 1)
(map-el #'length '((1 2 ) () (a b c d ) (1 (a b c d ) 3 )) )
                        ; = (2 0 4 3 )
```
соответственно.

Все три примера можно решить с помощью таких определяющих выражений:

```
(defun next (xl) (map-el #'1+ xl ))      ; Очередные числа:
(defun 1st  (xl) (map-el #'car xl ))     ; "головы" элементов = CAR
(defun lens (xl) (map-el #'length xl )) ; Длины элементов
```

Эти **определения функций формально эквивалентны** ранее приведенным – они сохраняют отношение между аргументами и результатами.
Параметром функционала может быть любая вспомогательная функция.

Пример 4.4. Пусть дана вспомогательная функция sqw, возводящая числа в квадрат

```
(defun sqw (x) (* x x))        ; Возведение числа в квадрат
(sqw 3)                        ; = 9
```

Построить список квадратов чисел, используя функцию sqw:

```
(defun sqware (xl) ;          ; Возведение списка чисел в квадрат
    (cond                      ; Пока аргумент не пуст,
      (xl (cons (sqw (car xl))    ; применяем sqw к его голове
                (sqware (cdr xl)) ; и переходим к остальным,
)   ) ) )                    ; собирая результаты в список

(sqware '(1 2 5 7 ))              ; = (1 4 25 49 )
```

Можно использовать map-el:

```
(defun sqware (xl) (map-el #'sqw xl))
```

Ниже приведено определение функции sqware- без вспомогательной функции, выполняющее умножение непосредственно. Оно влечет за собой двойное вычисление (CAR xl), т.е. такая техника не вполне эффективна:

```
(defun sqware- (xl)
    (cond
      (xl (cons (* (car xl) (car xl) )        ; квадрат головы
                              ; вычислять приходится дважды
                (sqware- (cdr xl))
)   ) ) )
```

Пример 4.5. Пусть дана вспомогательная функция tuple, превращающая любое данное в пару:

```
(defun tuple (x) (cons x x))
(tuple 3)                          ; = (3 . 3)
```

```
(tuple 'a)                          ; = (a . a)
(tuple '(Xa))    ; = ((Xa) . (Xa)) = ((Xa) Xa) - это одно и то же!
```

Чтобы преобразовать элементы списка с помощью такой функции, пишем сразу:

```
(defun duble (xl) (map-el #'tuple xl))   ; дублирование элементов
(duble '(1 (a) ()))                ; = ((1 . 1) ((a) a) (()))
```

Немногим сложнее организовать покомпонентную обработку двух списков.

Пример 4.6. Построить ассоциативный список, т.е. список пар из имен и соответствующих им значений, по заданным спискам имен и их значений:

```
(defun pairl (al vl)                ; Ассоциативный список
   (cond                            ; Пока AL не пуст,
     (al (cons (cons (car al) (car vl))      ; пары из "голов".
           (pairl (cdr al) (cdr vl)) ; Если VL исчерпается,
                                     ; то CDR будет давать NIL
)   ) ) )
```

```
(pair '(один два two three) '(1 2 два три))
  ; = ((один . 1)(два . 2)(two два )(three три ))
```

Пример 4.7. Определить функцию покомпонентной обработки двух списков с помощью заданной функции fn:

```
(defun map-comp (fn al vl)          ; fn покомпонентно применить
                                    ; к соотвественным элементам AL и VL
   (cond
     (xl (cons (funcall fn (car al) (car vl))
       ; Вызов данного FN как функции
           (map-comp (cdr al) (cdr vl))
)   ) ) )
```

Теперь **покомпонентные действия** над векторами, представленными с помощью списков, полностью в наших руках. Вот списки и сумм, и произведений, и пар, и результатов проверки на совпадение:

```
(map-comp #'+ '(1 2 3) '(4 6 9))
                                    ; = (5 8 12 ) Суммы
(map-comp #'* '(1 2 3) '(4 6 9))
                                    ; = (4 12 27 ) Произведения
(map-comp #'cons '(1 2 3) '(4 6 9))
                                    ; = ((1 . 4)(2 . 6)(3 . 9)) Пары
(map-comp #'eq '(4 2 3 ) '(4 6 9))
                                   ; = (T NIL NIL ) Сравнения
```

Достаточно уяснить, что надо делать с элементами списка, остальное довершит функционал map-comp, отображающий этот список в список результатов заданной функции.

Пример 4.8. Для заданного списка вычислим ряд его атрибутов, а именно - длина, первый элемент, остальные элементы списка без первого.

```
(defun mapf (fl el)
   (cond                  ; Пока первый аргумент не пуст,
     (fl (cons (funcall (car fl) el )
                            ; применяем очередную функцию
                            ; ко второму аргументу
           (mapf (cdr fl) el)
             ; и переходим к остальным функциям,
)  ) ) )      ; собирая их результаты в общий список

(mapf '(length car cdr) '(a b c d ))          ; = (4 a (b c d ))
```

Композициями таких функционалов можно применять **серии функций** к списку общих аргументов или к параллельно заданной последовательности списков их аргументов. Естественно, и серии, и последовательности представляются списками.

4.3. Безымянные функции

Определения в примерах 4 и 5 не вполне удобны по следующим причинам:

- В определениях целевых функций duble и sqwure встречаются имена специально определенных вспомогательных функций.

- Формально эти функции независимы, значит, программист должен отвечать за их наличие при использовании целевых функций на протяжении всего жизненного цикла программы, что трудно гарантировать.

- Вероятно, имя вспомогательной функции будет использоваться только один раз - в определении целевой функции.

С одной стороны, последнее утверждение противоречит пониманию смысла именования как техники, обеспечивающей неоднократность применения поименованного объекта. С другой стороны, придумывать подходящие, долго сохраняющие понятность и соответствие цели, имена - задача нетривиальная.

Учитывая это, было бы удобнее вспомогательные определения вкладывать непосредственно в определения целевых функций и обходиться при этом вообще без имен. Конструктор функций lambda обеспечивает такой стиль построения определений. Этот конструктор любое выражение expr превращает в функцию с заданным списком аргументов (x1 .
.. xK) в форме так называемых lambda-выражений:

```
( lambda (x1 ... xK) expr )
```

Имени такая функций не имеет, поэтому может быть применена лишь непосредственно. Defun использует данный конструктор, но требует дать функциям имена.

Пример 4.9. Определение функций duble и sqwure из примеров 4 и 5 без использования имен и вспомогательных функций:

```
(defun sqware (xl) (map-el  (lambda (x) (* x x)) xl))

(defun duble (xl) (map-el  (lambda (x) (cons x x)) xl))
```

Любую **систему взаимосвязанных функций** можно преобразовать к одной функции, используя вызовы безымянных функций.

4.4. Композиции функционалов, фильтры, редукции

Вызовы функционалов можно объединять в более сложные структуры таким же образом, как и вызовы обычных функций, а их композиции можно использовать в определениях новых функций.

Композиции функционалов позволяют создавать и более мощные построения, достаточно ясные, но требующие некоторого внимания.

Пример 4.10. Декартово произведение хочется получить определением вида:

```
(defun decart (x y)
  (map-el #'(lambda (i)
                (map-el #'(lambda (j) (list i j))
                y)
            )  x)  )
```

Но результат вызова

```
 (decart '(a s d) '( e r t))
```

дает

```
 (((A E) (A R) (A T)) ((S E) (S R) (S T)) ((D E) (D R) (D T)))
```

вместо ожидаемого

```
 ((A E) (A R) (A T) (S E) (S R) (S T) (D E) (D R) (D T))
```

Дело в том, что функционал map-el, как и map-comp (пример 7), собирает результаты отображающей функции в общий список с помощью операции cons так, что каждый результат функции образует отдельный элемент.
А по смыслу задачи требуется, чтобы список был одноуровневым.
Посмотрим, что получится, если вместо cons при сборе результатов воспользоваться функцией append.

Пример 4.11. Пусть дан список списков. Нужно их все сцепить в один общий список.

```
(defun list-ap (ll)
    (cond
      (ll (append (car ll)
            (list-ap (cdr ll) )
```

```
)   ) )   )

(list-ap '((1 2)(3 (4))))              ; = (1 2 3 (4))
```

Тогда по аналогии можно построить определение функционала map-ap:

```
(defun map-ap (fn ll)
    (cond
      (ll (append (funcall fn (car ll) )
            (map-ap fn (cdr ll) )
)   ) )   )

(map-ap 'cdr  '((1 2 3 4) (2 4 6 8) ( 3 6 9 12)))
          ; = (2 3 4     4 6 8      6 9 12)
```

Следовательно, интересующая нас форма результата может быть получена:

```
(defun decart (x y)
  (map-ap #'(lambda (i)
                    (map-el #'(lambda (j) (list i j))
                  y)    )   x)    )
 (decart '(a s d) '( e r t))
     ; = ((A E) (A R) (A T) (S E) (S R) (S T) (D E) (D R) (D T))
```

Сцепление результатов отображения с помощью append обладает еще одним полезным свойством: при таком сцеплении **исчезают вхождения пустых списков** в результат. А в Лиспе пустой список используется как ложное значение, следовательно, такая схема отображения пригодна для организации фильтров. **Фильтр** отличается от обычного отображения тем, что окончательно **собирает не все результаты**, а лишь удовлетворяющие заданному предикату.

Пример 4.12. **Построить список голов непустых списков можно следующим образом:**

```
(defun heads (xl) (map-ap
     #'(lambda (x)(cond (x (cons (car x) NIL))))
                  ; временно голова размещается в список,
                  ; чтобы потом списки сцепить
                xl
)                  )
(heads '((1 2) () (3 4) () (5 6)) )    ; = (1 3 5)
```

Рассмотрим еще один типичный вариант применения функционалов. Представим, что нас интересуют некие интегральные характеристики результатов, полученных при отображении, например, сумма полученных чисел, наименьшее или наибольшее из них и т.п. В таком случае говорят о **свертке** результата или его редукции. **Редукция** заключается в сведении множества элементов к одному элементу, **в вычислении которого задействованы все элементы множества**.

Пример 4.13. Подсчитать сумму элементов заданного списка.

```
(defun sum-el ( xl)
```

```
    (cond ((null xl) 0)
      (xl (+ (car xl)
             (sum-el (cdr xl) )
)   ) )   )

(sum-el '(1 2 3 4 ) )              ; = 10
```

Перестроим такое определение, чтобы вместо "+" можно было использовать произвольную бинарную функцию:

```
(defun red-el (fn xl)
    (cond ((null xl) 0)
      (xl (funcall fn (car xl)
             (red-el fn (cdr xl) )    ))))
 (red-el '+ '(1 2 3 4 ) )                ; = 10
```

В какой-то мере map-ap ведет себя как свертка - она сцепляет результаты без сохранения границ между ними.

Такие формулы удобны при моделировании множеств, графов и металингвистических формул, а к их обработке сводится широкий класс задач не только в информатике.

Встроенные функционалы (Clisp)

Отображающий функционал можно написать самим, а можно и воспользоваться одним из встроенных. Согласно стандарту, в реализацию языка Clisp обычно включены функционалы: **map, mapcar, maplist, mapcan, mapcon, mapc, mapl [6,7].** Каждый из них **покомпонентно обработает любой набор списков**. Отличаются они схемами выбора аргументов для отображающей функции, характером воздействия на исходные данные и оформлением результатов, передаваемых объемлющим формулам.

Map (map result-type function sequences ...)

Функция function вызывается на всех первых элементах последовательностей, затем на всех вторых и т.д. Из полученных результатов function формируется **результирующая последовательность**, строение которой задается параметром result-type с допустимыми значениями cons, list, array, string, NIL.

Mapcar (mapcar function list ...)

Функция function применяется к первым элементам списков, затем ко вторым и т.д.
Другими словами, function применяется к “головам” методично сокращающихся списков, и результаты применения собираются в **результирующий список.**

Примеры 4.14:

```
(mapcar #'+ '(1 2 3) '(4 5 6))          ; = (5 7 9)
(mapcar #'list '(1 2 3)'(4 5 6))        ; = ((1 4) (2 5) (3 6))
(defun evlis (args)(mapcar #’eval args))
          ; вычисление аргументов
```

**) Примечание. Без учета ассоциативного списка*

(defun evlis (args AL)(mapcar #'(lambda (x)(eval x AL)) args))

Maplist (maplist function list ...)

Функционал аналогичен mapcar, но function **применяется к "хвостам" списков** list, начиная с полного списка.

Пример:

(maplist #'list '(1 2 3)'(4 5 6))
; = *(((1 2 3) (4 5 6)) ((2 3) (5 6)) ((3) (6)))*

Mapc
Mapl

Оба функционала работают как mapcar и maplist, соответственно, за исключением того, что они **в качестве формального результата выдают первый список** (своеобразная аналогия с формальными аргументами).

Примеры 4.16:

(mapc #'list '(1 2 3)'(4 5 6)) *; = (1 2 3)*
(mapl #'list '(1 2 3)'(4 5 6)) *; = (1 2 3)*

Mapcan
Mapcon

И эти два функционала аналогичны mapcar и maplist, **но формирование результатов** происходит не с помощью операции cons, которая строит данные в новых блоках памяти, а **с помощью деструктивной функции** nconc, которая при построении новых данных использует память исходных данных, из-за чего исходные данные могут быть искажены.

В общем случае, отображающие функционалы представляют собой различные виды **структурной итерации** или итерации по структуре данных. При решении сложных задач полезно использовать отображения и их композиции, а также иметь в виду возможность создания своих функционалов.

Map-into отображает результат в конкретную последовательность.

Подведение итогов

Показанные построения достаточно разнообразны, чтобы было полезно формулировать, что ожидается от применения **техники функционального программирования**:

- Отображающие функционалы позволяют **строить программы из крупных действий.**

- Функционалы обеспечивают **гибкость отображений.**

- Определение функции может совсем **не зависеть от конкретных имен.**

- С помощью функционалов можно **управлять выбором формы результатов**.

- **Параметром функционала может быть любая функция**, преобразующая элементы структуры.

- Функционалы позволяют формировать **серии функций от общих данных**.

- Встроенные в Clisp функционалы приспособлены к **покомпонентной обработке произвольного числа параметров**.

- Любую **систему взаимосвязанных функций можно преобразовать к одной функции**, используя вызовы безымянных функций.

Лекция 5. **Имена, определения и контексты**

Рассматриваются основные методы расширения функциональных систем с помощью иерархии разнородных контекстов определений. Изучаются приемы достижения удобочитаемости функциональных программ при определении сложных функций и анализируются особенности типовых схем связывания имен переменных с их значениями, принятых в системах программирования. Знакомство с методом неподвижных точек в системах рекурсивных определений логически завершает схему выбора решений по взаимодействию имен с определениями.

Интерпретирующая система. Реализационное уточнение интерпретации

Эта глава предназначена для реализационного уточнения уже известных теоретических рассуждений. Ряд уточнений показан на примере, представляющем программу, которая определяет три функции UNION, INTERSECTION и MEMBER, а затем применяет эти функции к нескольким тестам [1]. На этом примере будут рассмотрены средства и методы, обеспечивающие **удобочитаемость функциональных программ** и удобство их развития при отладке. Как правило, удобство достигается включением в систему дополнительных функций для решения проблем, принципиальное решение которых уже обеспечено на уровне теории.

Пример:

Функции UNION и INTERSECTION применяют к множествам, каждое множество представлено в виде списка атомов. Заметим, что все функции рекурсивны, а UNION и INTERSECTION используют MEMBER. Схематично работу этих функций можно выразить следующим образом:

```
member = lambda [a;x] [  null[x]       ==> Nil ]
                      [ eq[a;car[x]] ==> T ]
                      [  T              ==> member [a;cdr[x]]        ]

union = lambda [x;y] [ null[x]              ==> y  ]
                     [member[car[x];y] ==> union [cdr[x];y] ]
                     [T                       ==>
                                cons[car[x];union[cdr[x];y]]      ]

intersection = lambda [x;y] [  null[x]        ==> NIL  ]
                            [ member[car[x];y] ==>          cons[car[x];intersection[cdr[x];y]]  ]
                                  [T  ==> intersection[cdr[x];y]     ]
```

Определяя эти функции на Лиспе, мы используем дополнительную псевдо-функцию DEFUN, объединяющую эффекты lambda и label. **Псевдо-функция** - это функция, которая выполняется ради ее воздействия на систему, тогда как обычная функция - ради ее значения. DEFUN заставляет функции стать определенными и допустимыми в системе равноправно со встроенными функциями.
Программа выглядит так:

```
(DEFUN
      MEMBER (A X)
       (COND
```

```
        ((NULL X)      Nil)
        ((EQ A (CAR X)) T)
        (T             (MEMBER A (CDR X)) )
)       )

(DEFUN
    UNION (X Y)
      (COND
        ((NULL X) Y)
        ((MEMBER (CAR X) Y) (UNION (CDR X) Y) )
        (T (CONS (CAR X) (UNION (CDR X) Y))) )) )
))

(DEFUN
     INTERSECTION (X Y)
            (COND
    ((NULL X) NIL)
    ((MEMBER (CAR X) Y) (CONS (CAR X) (INTERSECTION (CDR X) Y)) )
    (T (INTERSECTION (CDR X) Y))
))

(INTERSECTION '(A1 A2 A3) '(Al A3 A5))
(UNION '(X Y Z) '(U V W X))
```

Эта программа предлагает вычислить пять различных форм.
Первые три формы сводятся к применению псевдо-функции DEFUN.
Ее значение - имя определяемой функции, в данном случае - MEMBER, UNION, INTERSECTION. Более точно можно сказать, что полная область значения псевдо-функции DEFUN включает в себя некоторые
доступные ей части **системы**, обеспечивающие хранение информации о функциональных объектах, а формальное ее значение – атом, символизирующий определение функции.

Значение четвертой формы - (A1 A3). Значение пятой формы - (Y Z C B D X). Анализ пути, по которому выполняется рекурсия, показывает, почему элементы множества появляются именно в таком порядке.

В этом примере продемонстрировано несколько элементарных правил написания функциональных программ, сложившихся при реализации интерпретатора Лисп 1.5 в **дополнение к идеализированным правилам**, сформулированным в строгой теории Лиспа.

**) Примечание.* С точностью до разницы между EVALQUOTE – EVAL.

1) **Программа состоит из последовательности вычисляемых форм**. Если форма список, то ее первый элемент интерпретируется как функция. Остальные элементы списка – аргументы для этой функции. Они вычисляются с помощью EVAL, а функция применяется к ним с помощью APPLY, и полученное значение выводится как результат программы.

2) **Особого формата** для записи программ не существует. Границы строк игнорируются. Формат программы, включая идентификацию, выбран просто для удобства чтения.

3) **Любое число пробелов и концов строк** можно разместить в любой точке программы, но не внутри атома.

4) Не используются (QUOTE T), (QUOTE NIL). Вместо них применяется T, NIL, что влечет за собой соответствующее изменение определения EVAL.

5) **Атомы должны** начинаться с букв, чтобы их было **легко отличать от чисел**.

6) Точечная нотация может быть использована. Любое число пробелов перед или после точки, кроме одного, будет игнорироваться (один пробел нужен обязательно).

7) Точечные пары могут появляться как элементы списка, и списки могут быть элементами точечных пар.

Например:

((A . B) X (C . (E F D))) - есть допустимое S-выражение.

Оно может быть записано как

((A . B) . (X . ((C . (E . (F . (D . Nil)))) . Nil))) или

((A . B) X (C E F D))

8) Форма типа (A B C . D) есть сокращение для (A . (B . (C . D))). Любая другая расстановка точек на одном уровне есть ошибка, например (A. B C).

9) Набор основных функций обеспечен системой. Другие функции могут быть введены программистом. **Порядок введения функций не имеет значения**. Любая функция может использоваться в определении другой функции.

Вывод S-выражений на печать и в файлы выполняет псевдо-функция **PRINT**, чтение данных обеспечивает псевдо-функция **READ**. Программа из файла может быть загружена псевдо-функцией **LOAD**. Например:

```
(LOAD 'TEST.LSP)
```

В таком случае надо позаботиться о выводе результатов программы с помощью псевдо-функции PRINT. Например:

```
(PRINT (INTERSECTION '(A1 A2 A3) '(Al A3 A5))  )
(PRINT (UNION '(X Y Z) '(U V W X)) )
(PRINT (UNION (READ) '(1 2 3 4)) )
; объединение вводимого списка со списком '(1 2 3 4)
```

Именование значений и подвыражений

Переменная - это символ, который используется для представления аргумента функции. Предположим, что интерпретатор получает следующее S-выражение:

```
((LAMBDA (X Y) (CONS X Y)) 'A 'B)
```

Функция: (LAMBDA (X Y) (CONS X Y))
Аргументы: (A B)

EVAL0 через EVAL передает эти аргументы функции APPLY. (См. лекцию 3).

```
(apply #'(LAMBDA (X Y) (CONS X Y)) '(A B) Nil )
```

APPLY свяжет переменные и передаст функцию и удлиннившийся a-список EVAL для вычисления.

```
(eval '(CONS X Y) ' ((X . A) (Y B) Nil))
```

EVAL вычисляет переменные и сразу передает их консолидации, строящей из них бинарный узел.

```
(Cons 'A 'B) = (A . B)
```

Реальный интерпретатор пропускает один шаг, требуемый формальным определением универсальных функций".

На практике сложилась традиция в систему функционального программирования включать специальные функции, обеспечивающие иерархию контекстов, в которых переменные обладают определенным значением. Для Clisp это LET и LET*.

Let сопоставляет **локальным переменным** независимые выражения. С ее помощью можно вынести из сложного определения любые совпадающие подвыражения.
Пример:

```
(defun UNION- (x y)   (let   ( (a-x (CAR x))
                               (d-x (CDR x))   )

        (COND  ((NULL x)              y)
               ((MEMBER a-x y) (UNION d-x y)  )
               (T                      (CONS a-x (UNION d-x y)) )   )     ))
```

Let* - сопоставляет локальным переменным взаимосвязанные выражения. Она позволяет дозировать сложность именуемых подвыражений.
Пример:

```
(defun ( (member (a x)  (let* (   (n-x (null x))
                                  (a-x (car x))
                                  (d-x (cdr x))
                                  (e-car (eq a a-x))   )

        (COND (N-X  Nil) (E-CAR T) (T (MEMBER A D-X)))  )        ))
```

**) Примечание.* Эквивалентность с точностью до побочного эффекта.

Глобальные переменные можно объявить с помощью специальной функции **DEFPARAMETER.**

```
(DefParameter glob '(a b c))
```

Значение такой переменной доступно в **любом контексте**, оно может быть переопределено. Возможно оттеснение одноименными локальными переменными с восстановлением при выходе из соответствующих контекстов.

```
(let ((glob 12))(print glob))
(print glob)
```

Напечатано будет:

```
12
(A B C)
```

Константы, как принято говорить**,** представляют сами себя, в противоположность переменным, представляющим что-то другое. Это не вполне точно. Корректнеее говорить, что одна переменная более константна, чем другая, если она связана на более высоком уровне и ее значение изменяется не столь часто.

В реальных системах константное значение атома может быть организовано с помощью так называемого списка свойств атома, являющегося представлением переменной. Каждый атом имеет свой p-список (property list), доступный через хэш-таблицу идентификаторов, что действует эффективнее, чем a-список. С каждым атомом связана специальная структура данных, в которой размещается имя атома, его значение, определение функции, представляемой этим же атомом, и список произвольных свойств, помеченных индикаторами. При вычислении переменных EVAL исследует эту структуру до поиска в a-списке. Такое устройство констант не позволяет им служить переменными в a-списке.

Константы могут быть заданы программистом. Чтобы переменная X стала обозначением для (A B C D), надо воспользоваться псевдо-функцией **DEFCONSTANT**.

```
(DefConstant  X  '(A B C D))
```

Особый интерес представляет тип констант, которые всегда обозначают себя, Nil - пример такой константы. Такие константы как T, Nil, а также самоопределимые константы (числа, строки), не могут использоваться в качестве переменных. Смысл чисел и строк не может быть изменен с помощью DEFCONSTANT.

Функции, представленная атомом, **реализационно подобна** механизму, которым атом обозначает переменную или аргумент. Если функция рекурсивна, то ей надо дать имя. Здесь это делается с помощью формы LABEL, которая связывает название с определением функции в ассоциативном списке (**a-списке).** Название связано с определением функции точно так же, как переменная связана со своим значением, поэтому теоретически можно обойтись LAMBDA, а LABEL не входит в базис Лиспа.

На практике LABEL используется редко – она введена в учебных целях как вспомогательное построение. Удобнее связывать название с определением другим способом – подобно глобальному значению. Это делается путем размещения определения функции в **списке свойств атома (p-список)**, символизирующего ее название. Выполняет данную операцию псевдо-функция DEFUN, описанная в начале этой лекции.

Когда APPLY интерпретирует функцию, представленную атомом, она исследует p-список до текущего состояния a-списка. Таким образом, DEFUN будет опережать LABEL.

Тот факт, что большинство функций - константы, определенные программистом, а не переменные, изменяемые программой, вызван отнюдь не каким-либо недостатком функционального программирования. Напротив, он указывает на потенциал подхода, который мы не научились использовать.

**) Примечание.* Работы по теории компиляции, оптимизации и верификации программ, смешанным вычислениям, суперпрограммированию и т.п. активно используют средства функционального программирования.

Системы функционального программирования обеспечивают манипулирование функциональными переменными так же, как и обычными. Но организуется это с помощью ряда специальных функций, осуществляющих переход от символов, изображающих функции, к функциональным объектам, представленным этими символами. Такой переход может быть реализован в рамках любого языка программирования, но на Лиспе он выглядит естественно и поддерживается в любой Лисп-системе. Рассмотрим средства Clisp, обеспечивающие структурирование функциональных объектов.

Labels - позволяет из списка определений функций формировать контекст, в котором вычисляются выражения.

Flet - специальная функция, позволяющая вводить локальные нерекурсивные функции.

defun - позволяет вводить новые определения на текущем уровне.

```
(labels ( (INTERSECTION (x y)  (let* (  (N-X (null x))
                                        (MEM-CAR (member (car x) y))
                                             (INT  #'intersection)          )

(flet  ((f-tail         (fn sx sy)                  (apply fn (list (cdr sx) sy)) )
        (cons-f-tail (fn sx sy)
                     (cons (car sx) (apply fn (list (cdr sx) sy)) )))

(COND   (N-X            NIL)
                    (MEM-CAR (cons-f-tail INT x y) )
                    (T              (f-tail   INT x y))   )           )))

(defun UNION (x y)   (let   ( (a-x (CAR x))
                                    (d-x (CDR x))   )

        (COND  ((NULL x)              y)
                    ((MEMBER a-x y) (UNION d-x y)  )
                    (T                      (CONS a-x (UNION d-x y)) )   )     ))

(INTERSECTION- '(A1 A2 A3) '(A1 A3 A5))
(UNION- '(X Y Z) '(U V W X))
))
```

Функции на машинном языке (низкоуровневые)

Некоторые функции вместо определений с помощью S-выражений закодированы как **замкнутые машинные подпрограммы**. Такая функция будет иметь особый индикатор в списке свойств с указателем, который позволяет интерпретатору связаться с подпрограммой. Существует три случая, в которых низкоуровневая подпрограмма может быть включена в систему:

1) Подпрограмма **закодирована внутри** Лисп-системы.

2) Функция **кодируется пользователем вручную** на языке типа ассемблера.

3) Функция сначала определяется с помощью S-выражения, затем **транслируется компилятором**. Компилированные функции могут выполняться в 2-100 раз быстрее, чем интерпретироваться.

Обычно EVAL вычисляет аргументы функций до применения к ним функций с помощью APPLY. Таким образом, если EVAL задано (CONS X Y), то сначала вычисляются X и Y, а потом над полученными значениями работает CONS. Но если EVAL задано (QOUTE X), то X не будет вычисляться. QUOTE - специальная форма, которая препятствует вычислению своих аргументов.

Специальная форма отличается от других функций двумя свойствами. Ее аргументы не вычисляются, пока она **сама не просмотрит свои аргументы**. COND, например, имеет свой особый способ вычисления аргументов с использованием EVCON. Второе отличие заключается в том, что **специальные формы могут иметь неограниченное число аргументов**.

Самоприменимость функций

Возможность использования **безымянных определений функций** не вполне очевидна для вспомогательных рекурсивных функций, так как их имена нужны в их собственных определениях. Но при необходимости и определение рекурсивной функции можно привести к форме, не зависящей от ее имени. Такие имена формально работают как связанные переменные, т.е. их смысл не зависит от имени - нечто вроде местоимения. Это позволяет выполнять систематическую замену названия функции на другие символы или выражения.

Определение рекурсивной функции можно преобразовать к безымянной форме. Техника **эквивалентных преобразований** позволяет поддерживать целостность системы функций втягиванием безымянных вспомогательных функций внутрь тела основного определения. Верно и обратное: любую конструкцию из лямбда-выражений можно преобразовать в систему отдельных функций.

Такое, целостно свернутое, определение позволяет получить технику функциональных определений и их преобразований, которое позволяет рассматривать решение задачи с той точки зрения, с какой это удобно при постановке задачи, достигать естественную степень подробности, гибкости и мобильности.

Специальная фукнция FUNCTION обеспечивает доступ к функциональному объекту, представленному S-выражением, а функция FUNCALL обеспечивает применение функции к произвольному числу ее аргументов.

(funcall f a1 a2 ...) = (apply f (list a1 a2 ...))

Разрастание числа функций, манипулирующих функциями в Clisp, связано с реализационным различием структурного представления данных и представляемых ими функций.

Программы для Лисп-интерпретатора.

Цель этой части - помочь избежать некоторых общих ошибок.

Пример 5.1. (CAR '(A B)) = (CAR (QUOTE(A B)))

Функция: CAR
Аргументы: ((A B))

Значение есть A. Заметим, что интерпретатор ожидает список аргументов. Единственным аргументом для CAR является (A B). Добавочная пара скобок возникает, т.к. APPLY подается список аргументов.

Можно написать (LAMBDA (X) (CAR X)) вместо просто CAR. Это корректно, но не является необходимым.

Пример 5.2. (CONS 'A '(B . C))

Функция: CONS
Аргументы: (A (B . C))

Результат (A . (B . C)) программа печати выведет как (A B . C)

Пример 5.4. (CONS '(CAR (QUOTE (A . B))) '(CDR (QUOTE (C . D))))

Функция: CONS
Аргументы: ((CAR (QUOTE (A . B))) (CDR (QUOTE (C . D))))

Значением такого вычисления будет

((CAR (QUOTE (A . B))) CDR (QUOTE (C . D)))

Скорее всего, это совсем не то, чего ожидал новичок. Он рассчитывал вместо (CAR (QUOTE (A . B)) получить A и увидеть (A . D) в качестве итогового значения CONS. Интерпретатор настроен не на список уже готовых значений аргументов, а на список выражений, которые будут вычисляться до вызова функции. Кроме очевидного стирания апострофов

(CONS (CAR (QUOTE (A . B))) (CDR (QUOTE (C . D))))

ниже приведены еще три правильных способа записи нужной формы. Первый состоит в том, что CAR и CDR части функции задаются с помощью LAMBDA в определении функции. Второй заключается в переносе CONS в аргументы и вычислении их с помощью EVAL при пустом a-списке. Третий - в принудительном выполнении константных действий в представлении аргументов

((LAMBDA (X Y) (CONS (CAR X) (CDR Y))) '(A . B) '(C . D))

Функция: (LAMBDA (X Y) (CONS (CAR X) (CDR Y)))
Аргументы: ((A . B)(C . D))

(EVAL '(CONS (CAR (QUOTE (A . B))) (CDR (QUOTE (C . D)))) Nil)

Функция: EVAL
Аргументы: ((CONS (CAR (QUOTE (A . B))) (CDR (QUOTE (C . D)))) Nil)

Значением того и другого является (A . D)

((LAMBDA (X Y) (CONS (EVAL X) (EVAL Y))) '(CAR (QUOTE (A . B)))
'(CDR (QUOTE (C . D))))

Функция: (LAMBDA (X Y) (CONS (EVAL X) (EVAL Y)))
Аргументы: ((CAR (QUOTE (A . B))) (CDR (QUOTE (C . D))))

Решения этого примера показывают, что **грань между функциями и данными достаточно условна -** одни и те же вычисления можно осуществить при разном распределении промежуточных вычислений внутри выражения, передвигая эту грань.

Лекция 6. **Свойства атомов и категории функций**

Методы расширения функциональных построений применены для моделирования привычного операторно-процедурного стиля программирования и техники работы с глобальными определениями. Демонстрируется еще один важный метод - обобщение базовой схемы обработки символьных выражений и представленных с их помощью функциональных форм на основе списков свойств атомов. В результате можно собирать и специализировать функционально полное определение гибкого и расширяемого интерпретатора для языка программирования на примере Лиспа, написанном на Лиспе. Акцент на возможности варьирования семантики функций и пополнения семантического базиса с целью автоматизации выполненных построений в процессе исследования границ класса решаемых задач и конкретизации методов их решения.

Prog-выражения и циклы

Существует большое число чисто теоретических работ, исследовавших соотношения между потенциалом того и другого подхода и пришедших к заключению о формальной сводимости в обе стороны при некоторых непринципиальных ограничениях на технику программирования. Методика сведения императивных программ в функциональные заключается в определении правил разметки или переписывания схемы программы в функциональные формы. Переход от функциональных программ к императивным технически сложнее: используется интерпретация формул над некоторой специально устроенной абстрактной машиной [3,4]. На практике переложение функциональных программ в императивные выполнить проще, чем наоборот – может не хватать широты понятий.

С практической точки зрения любые конструкции стандартных языков программирования могут быть введены как функции, дополняющие исходную систему программирования, что делает их вполне легальными средствами в рамках функционального подхода. Надо лишь четко уяснить цену такого дополнения и его преимущества, обычно связанные с наследованием решений и

привлечением пользователей. В первых реализациях Лиспа были сразу предложены специальные формы и структуры данных, служащие мостом между разными стилями программирования, а заодно смягчающие недостатки исходной, слишком идеализированной, схемы интерпретации S-выражений, выстроенной для учебных и исследовательских целей. Важнейшее такого рода средство, выдержавшее испытание временем - prog-форма, списки свойств атома и деструктивные операции, расширяющие язык программирования так, что становятся возможными оптимизирующие преобразования структур данных, программ и процессов, а главное – раскрутка систем программирования [1].

Применение prog-выражений позволяет писать “паскалеподобные” программы, состоящие из операторов, предназначенных для исполнения. (Точнее “алголоподобные”, т.к. появились лет за десять до паскаля. Но теперь более известен паскаль.)

Для примера prog-выражения приводится императивное определение функции **Length** *), сканирующей список и вычисляющей число элементов на верхнем уровне списка. Значение функции Length - целое число. Программу можно примерно описать следующими словами:

*) *Примечание.* Стилизация примера от МакКарти [1].

"Это функция одного аргумента L.
Она реализуется программой с двумя рабочими переменными u и v.
Записать число 0 в v.
Записать аргумент L в u.
A: Если u содержит NIL, то программа выполнена и значением является то, что сейчас записано в v.
Записать в u cdr от того, что сейчас в u.
Записать в v на единицу больше того, что сейчас записано в v.
Перейти к A"

Эту программу можно записать в виде Паскаль-программы с несколькими подходящими типами данных и функциями. Строкам описанной выше

программы в предположении, что существует библиотека Лисп-функций над списками на Паскале, соответствуют строки определения функции:

```
function LENGTH (L: list) : integer;

        var U: list;
            V: integer;
begin
    V := 0;
    U := l;
A:    if null (U) then LENGTH := V;
    U := cdr (U);
    V := V+1;
        goto A;
end;
```

Переписывая в виде S-выражения, получаем программу:

```
(defun
LENGTH (lambda (L)
            (prog (U V)
    (setq V 0)
    (setq U L)
A     (cond  ((null U)(return V)))
    (setq U (cdr U))
    (setq V (+ 1 V))
(go  A)  )))  ))

(LENGTH '(A B C D))
(LENGTH ‘((X . Y) A CAR (N B) (X Y Z)))
```

Последние две строки содержат тесты. Их значения четыре и пять, соответственно.

Prog-форма имеет структуру, **подобную определениям функций и процедур в Паскале**: (PROG, список рабочих переменных, последовательность операторов и атомов ...) Атом в списке является **меткой**, локализующей оператор, расположенный вслед за ним. В приведенном примере метка A локализует оператор, начинающийся с "COND".

Первый список после символа PROG называется **списком рабочих переменных**. При отсутствии таковых должно быть написано NIL или (). С рабочими переменными обращаются примерно как со **связанными переменными**, но они не могут быть связаны ни с какими значениями через lambda. Значение каждой рабочей переменной есть NIL, до тех пор, пока ей не будет присвоено что-нибудь другое.

Для **присваивания рабочей переменной** применяется форма **SET**. Чтобы присвоить переменной pi значение 3.14 пишется (SET (QUOTE PI)3.14). **SETQ** подобна SET, но она еще и **блокирует вычисление первого аргумента**. Поэтому (SETQ PI 3.14) - запись того же присваивания. SETQ обычно удобнее. **SET и SETQ могут изменять значения любых переменных** из а-списка более внешних функций. Значением SET и SETQ является значение их второго аргумента.

Обычно операторы выполняются последовательно. Выполнение оператора понимается как его вычисление при текущем а-списке и отбрасывание его значения. Операторы программы часто выполняются в большей степени ради действия, чем ради значения.

GO-форма, используемая для указания перехода (GO A) указывает, что программа продолжается оператором, помеченным атомом A, причем это A может быть и из более внешнего prog.

Условные выражения в качестве операторов программы обладают полезными особенностями. Если ни одно из пропозициональных выражений не истинно, то вместо указания на ошибку, происходящего во всех других случаях, программа продолжается оператором, следующим за условным выражением. Это справедливо лишь для условного выражения, находящегося на верхнем уровне Prog.

RETURN - нормальный конец программы. Аргумент return вычисляется, что и является значением программы. Никакие последующие операторы не вычисляются.

Формы Go, Set, Return могут применяться как операторы лишь на верхнем уровне PROG или внутри COND, находящегося на верхнем уровне PROG.

Если программа прошла все свои операторы, она оканчивается со значением NIL.

Prog-выражение, как и другие Лисп-функции, **может быть рекурсивным**.

Функция REV, обращающая список и все подсписки, столь же естественно пишется с помощью рекурсивного Prog-выражения.

```
function rev (x: list) :List
      var y, z: list;
begin
 A: if null (x) Then rev := y;
     z := cdr (x);
   if atom (z) then goto B;
     z := rev (z);
 B:   y := cons (z, y);
     x := cdr (x);
                goto A
end;
```

Функция rev обращает все уровни списка, так что rev от (A ((B C) D)) даст ((D (C B))A) .

```
(DEFUN rev (x) (prog (y z)

A  (COND ((null x)(return y)))
      (setq z (CDR x))
   (COND ((ATOM z)(goto B)))
```

```
      (setq z (rev z))

B  (setq y (CONS z y))
   (setq x (CDR x))
   (goto A)
))
```

Для того чтобы форма prog была полностью законна, необходима возможность дополнять а-список рабочими переменными. Кроме того, операторы этой формы требуют специального расширения языка - в него включаются формы go, set и return, не известные вне prog. Атомы, играющие роль меток, работают как **указатели помеченного блока**.

Кроме того, уточнен механизм условных выражений: отсутствие истинного предиката не препятствует формированию значения **cond-оператора**, т.к. все операторы игнорируют выработанное значение. Это позволяет считать, что значением является Nil. Такое доопределение условного выражения приглянулось и в области обычных функций, где оно часто дает компактные формулы для рекурсии по списку.

В принципе, SET и SETQ могут быть реализованы с помощью а-списка примерно так же, как и поиск значения, только с копированием связей, расположенных ранее изменяемой переменной (см. функцию assign из лекции 3) . Более эффективная реализация будет описана ниже.

```
(DEFUN set (x y) (assign  x  y  Alist))
```

Обратите внимание, что введенное таким образом присваивание работает разнообразнее, чем традиционное: обеспечена **вычисляемость левой части присваивания**, т.е. можно в программе **вычислять имена переменных**, значение которых предстоит поменять.
Пример 6.1:

```
 (setq x 'y)
 (set x 'NEW)
```

```
(print x)
(print y)
```

Напечатается Y и NEW.

Списки свойств атомов и структура списков

До сих пор атом рассматривался только как **уникальный указатель**, обеспечивающий быстрое выяснение различимости имен, названий или символов. В настоящем разделе описываются **списки свойств**, которые начинаются в **указанных ячейках**.

Каждый атом имеет список свойств. Когда атом читается (вводится) впервые, тогда для него и создается список свойств. Список свойств характеризуется специальной структурой, подобной записям в Паскале, но указатели в такой записи сопровождаются **тэгами**, символизирующими тип хранимой информации. Первый элемент этой структуры расположен по адресу, который задан в указателе. Остальные элементы доступны по этому же указателю с помощью ряда специальных функций. Элементы структуры содержат различные свойства атома. Каждое **свойство помечается** атомом, называемым **индикатором**, или расположено в фиксированном поле структуры.

С середины 70-х годов возникла тенденция повышать эффективность разработкой специальных структур, отличающихся в разных реализациях. Существуют реализации, например, muLisp, допускающие работу с представлениями атома как с обычными списками посредством функций car, cdr.

Согласно стандарту Common Lisp, глобальные значения переменных и определения функций хранятся в фиксированных полях структуры атома. Они доступны с помощью специальных функций, symbol-function и symbol-value. Список произвольных свойств можно получить с использованием функции symbol-plist. Функция remprop в Clisp удаляет лишь первое вхождение заданного свойства. Новое свойство можно ввести формой вида:

```
(setf (get ATOM  INDICATOR ) PROPERTY )
```

Числа представляются в Лиспе как специальный тип атома. Атом такого типа состоит из указателя с тэгом, специфицирующим слово как число, тип этого числа и адрес собственно числа произвольной длины. В отличие от обычного атома одинаковые числа при хранении не совмещаются.

До этого момента списки рассматривались на уровне текстового ввода-вывода. В настоящем разделе анализируется кодовое представление списков внутри машины и мусорщик, обеспечивающий повторное использование памяти.

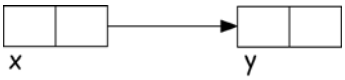
Представление структуры списка

Отформатировано

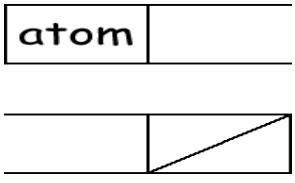
В машине списки хранятся не как последовательности символов, а как структурные формы, построенные из машинных слов как частей деревьев, подобно записям в Паскале при реализации односвязных списков. При изображении структуры списка машинное слово рисуется как прямоугольник,

разделенный на две части: адрес и декремент. Каждая из частей занимает фиксированное число разрядов, представляющее тэг и адрес.

Если декремент слова “x” указывает на слово “y”, то это можно выразить стрелками на схеме:



Теперь можно дать правило представления S-выражений в машине. Представление атомов будет пояснено ниже. В тех случаях, когда машинное слово в адресе или декременте содержит указатель на атом, данный атом отобразится как запись в соответствующем прямоугольнике:

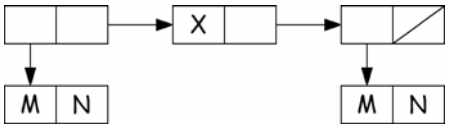


Правило представления неатомных S-выражений - начало со слова, содержащего указатель на car выражения в адресе и указатель на cdr в декременте. Ниже нарисовано несколько схем S-выражений. По соглашению NIL обозначается как перечеркнутый по диагонали прямоугольник.

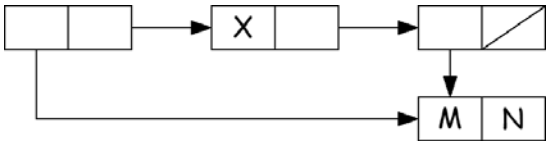
(A . B)

Непосредственная польза от сопоставления графического вида с представлением списков в памяти поясняется при рассмотрении функций, работающих со списками, на следующем примере из [1]:

((M . N) X (M . N))

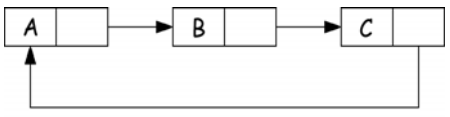


Возможное для списков использование общих фрагментов
((M . N) X (M . N)) может быть представлено графически:



В точности такую структуру непосредственно текстом представить невозможно, но ее можно построить с помощью выражений:
(let ((a '(M . N))) (setq b (list a 'X a)))
((lambda (a) (list a 'X a))'(M . N))

Циклические списки обычно не поддерживаются. Такие списки не могут быть введены псевдо-функцией read, однако они могут возникнуть как результат вычисления некоторых функций, точнее, применения структуроразрушающих или деструктивных функций. Печатное изображение таких списков имеет неограниченную длину. Например, структура



может распечатываться как (A B C A B C ...).

Преимущества структур списков для хранения S-выражений в памяти:

1) Размер и даже число выражений, с которыми программа будет иметь дело, можно не предсказывать. Кроме того, можно не организовать для размещения выражений блоки памяти фиксированной длины.

2) Ячейки можно переносить в список свободной памяти, как только исчезнет необходимость в них. Даже возврат одной ячейки в список имеет значение, но если выражения хранятся линейно, то организовать использование лишнего или освободившегося пространства из блоков ячеек трудно.

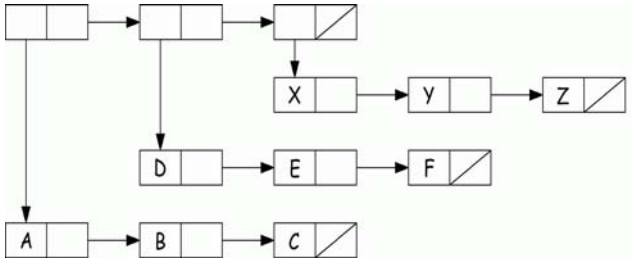
3) Выражения, являющиеся продолжением нескольких выражений, могут быть предоставлены только однажды.

Ниже следует простой пример, иллюстрирующий точность построения структур списка. Показаны два типа структур списка и описаны на Лиспе функции для преобразования одного типа в другой.

Предполагается, что дан список вида:

L1 = ((A B C)(D E F) ... (X Y Z))

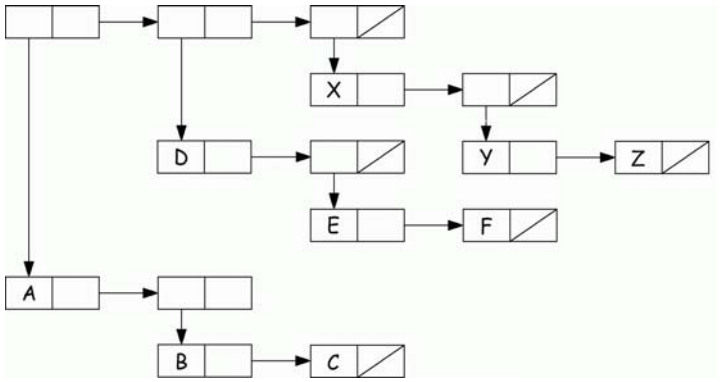
представленный как …



и нужно построить список вида

L2 = ((A (B C))(D (E F)) ... (X (Y Z)))

представленный как …



Рассмотрим типичную подструктуру (A (B C)) второго списка.
Она может быть построена из A, B и C с помощью

(cons ‘A (cons (cons ‘B (cons ‘C NIL)) NIL))

или, используя функцию list, можно то же самое записать как

(list ‘A (list ‘B ‘C))

В любом случае дан список x из трех атомов x = (A B C), аргументы A, B и C, используемые в предыдущем построении, можно найти как

A = (car x)
B = (cadr x)
C = (caddr x)

Первый шаг в получении L2 из L1 - это определение функции grp, строящей (X (Y Z)) из списка вида (X Y Z).

(grp x) = (list (car x) (list (cadr x) (caddr x)))

Здесь grp применяется к списку L1 в предположении, что L1 заданного вида.

Для достижения цели новая функция mltgrp определяется как

```
(mltgrp L) = (COND ((null L) NIL)
                  (T (cons (grp (car L)) (mltgrp (cdr L)) )))
```

Итак, mltgrp, применяемая к списку L1, перебирает (X Y Z) по очереди и применяет к ним grp, чтобы установить их новую форму (X (Y Z)) до тех пор, пока не завершится список L1 и не построится новый список L2.

Деструктивные (разрушающие) преобразования структуры списков

Отформатировано

Теория рекурсивных функций, изложенная в лекции 2, будет упоминаться далее как строгий, чистый или элементарный Лисп. Хотя этот язык универсален в смысле теории вычислимых функций от символьных выражений, он непрактичен как система программирования без дополнительного инструмента, увеличивающего выразительную силу и эффективность языка.

В частности, элементарный Лисп не имеет возможности модифицировать структуру списка. Единственная базисная функция, влияющая на структуру списка - это cons, а она не изменяет существующие списки, только создает новые. Функции, описанные в чистом Лиспе, такие как subst, в действительности не модифицируют свои аргументы, но делают модификации, копируя оригинал.

Элементарный Лисп работает как расширяемая система, в которой информация как бы никогда не пропадает. Set внутри Prog лишь формально смягчает это свойство, сохраняя ассоциативный список и моделируя присваивания теми же CONS.

Теперь же Лисп обобщается с точки зрения структуры списка добавлением структуроразрушающих средств - **деструктивных базисных операций** над списками rplaca и rplacd. Эти операции могут применяться для замены адреса или

декремента любого слова в списке. Они используются ради их действия так же, как и ради их значения и называются псевдо-функциями.

(rplaca x y) заменяет адресную часть x на y. Ее значение - x, но x, несколько отличающийся от того, что было раньше. На языке значений rplaca можно описать равенством

(rplaca x y) = (cons y (cdr x))

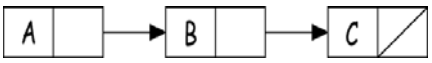
Но действие различное: никакие cons не вызываются и новые слова не создаются.

(rplacd x y) заменяет декремент x на y.

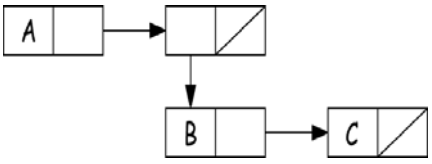
Эти операции должны применяться с осторожностью! Они могут в корне преобразить существующие определения и основную память. Их применение **может породить циклические списки**, возможно, влекущие **бесконечную печать** или **выглядящие бесконечными** для таких функций как equal и subst.

Деструктивные функции используются при реализации списков свойств атома и ряда эффективных, но **небезопасных, функций** Clisp-а, таких как nconc, mapc и т.п.

Для примера рассмотрим функцию mltgrp. Это **преобразующая список функция**, которая преобразует копию своего аргумента. Подфункция grp

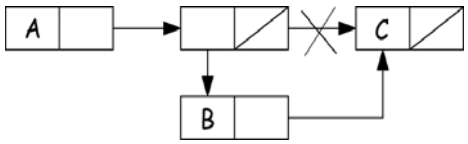


реорганизует подструктуру (A B C) в структуру (A (B C)) из тех же атомов:



Исходная функция grp делает это, создавая новую структуру и используя четыре cons. Из-за того, что в оригинале только три слова, по крайней мере, один cons

необходим, а grp можно переписать с помощью rplaca и rplacd. Изменение состоит в следующем:



Пусть новое машинное слово строит (cons (cadr x) (cddr x)) . Тогда указатель на него заготавливает форма:

(rplaca (cdr x) (cons (cadr x) (cddr x)))

Другое изменение состоит из удаления указателя из второго слова на третье. Оно выполняется как (rplaca (cdr x) NIL).

Функция pgrp теперь определяется как соотношение:

(pgrp x) = (rplacd (rplaca (cdr x) (cons (cadr x) (cddr)x))) NIL)

Функция pgrp используется, в сущности, ради ее действия. Ее значением, неиспользуемым, является подструктура ((B C)). Поэтому для mltgrp достаточно, чтобы pgrp выполнялось, а ее значение игнорировалось. Поскольку верхний уровень не должен копироваться, mltgrp не обязана основываться на cons.

```
(pmltgrp L) =(COND ((null L) NIL)
                   (T (prog2 (pgrp (cdr L)) (pmltgrp (cdr L)) )))
```

prog2 - функция, вычисляющая свои аргументы. Ее значение - второй аргумент. Значение pmltgrp - NIL. pgrp и pmltgrp - псевдо-функции.

Список свободной памяти и сбор мусора

В любой момент времени только часть памяти, отведенной для структур списков,

действительно используется для хранения S-выражений. Остальные ячейки организованы в простой список, называемый **списком свободной памяти**.

Первые реализации действовали по следующей схеме [1]:

Определенный регистр FREE содержит информацию о первой ячейке списка свободной памяти. Когда требуется слово для формирования дополнительной структуры списка, берется первое слово списка свободной памяти, а код в регистре FREE заменяется на информацию о втором слове списка свободной памяти. Не требуется никаких программных средствдля того, чтобы пользователь программировал **возврат ячеек** в список свободной памяти. Этот возврат **происходит автоматически** при пропуске программы, где бы ни исчерпался список свободной памяти.
Такая программа, восстанавливающая память, называется "**сбор мусора**".

Любая часть структуры списка, доступная программе, рассматривается как **активный список** и не затрагивается при сборе мусора. Активные списки доступны программе через некоторые **фиксированные наборы базисных ячеек**, таких как ячейки списка атомов и ячейки, вмещающие частичные результаты Лисп-выражений. Структуры списков могут быть произвольной длины, но каждое активное слово должно быть достижимо из базисной ячейки цепью car-cdr.

Любая ячейка, не достижимая таким образом, недоступна для программы и не активна, поэтому ее содержимое не представляет интереса. Неактивные, то есть недоступные программе ячейки восстанавливаются в списке свободной памяти следующим образом. Во-первых, каждая ячейка, к которой можно получить доступ через цепь car-cdr, **метится** установлением служебного бита. Где бы ни выявилось помеченное слово в цепи во время процесса пометки, ясно, что остаток раскручиваемого списка содержит уже помеченные ячейки. Затем предпринимается **линейное выметание осовободившегося пространства**, собирая непомеченные ячейки в новый список свободной памяти и устраняя пометку активных ячеек.

Такая реализация экономична в отношении памяти, но она имеет ряд неприятных следствий - непредсказуемые длинноты при поиске очередной порции ячеек и "перегрев системы", если такие порции слишком малы для продолжения счета. По мере роста производительности оборудования разработаны более простые и менее обременительные алгоритмы повторного использования памяти на базе параллельных процессов и профилактического копирования активных структур данных в дополнительные блоки памяти. Такие методы не требуют сложной разметки и анализа достижимости. Содержательная аналогия с мастерским мытьем посуды, то есть не допуская переполнения раковины, вполне отражает метод stop&copy, принятый в современных реализациях.

Гибкий интерпретатор

В качестве примера повышения гибкости определений приведено упрощенное определение Лисп-интерпретатора на Лиспе, полученное из М-выражения, приведенного **Дж. Мак-Карти в описании Lisp 1.5** [1]. Уменьшена диагностичность, нет предвычислений и формы Prog.

Лисп хорошо приспособлен к **оптимизации программ**. Любые совпадающие подвыражения можно локализовать и вынести за скобки, как можно заметить по передаче значения "(ASSOC e al)" в нижеприведенном определении EVAL.

Определения функций хранятся в ассоциативном списке, как и значения переменных.
Функция **SUBR -** вызывает примитивы, реализованные другими, обычно низкоуровневыми, средствами. **ERROR** – выдает сообщения об ошибках и сведения о контексте вычислений, способствующие поиску источника ошибки.
Уточнена работа с функциональными аргументами:

```
(DEFUN EVAL  (e al )
    (COND

     ((EQ e NIL ) NIL )
     ((ATOM e ) ((LAMBDA (v )
```

```
                (COND (v (CDR v ) )
                      (T (ERROR 'undefvalue )) )
                ) (ASSOC e al ) )
               )

      ((EQ (CAR e) 'QUOTE )    (CAR (CDR e )) )
      ((EQ (CAR e) 'FUNCTION ) (LIST 'FUNARG (CADR fn ) al ) )
      ((EQ (CAR e) 'COND  )    (EVCON (CDR e ) al ) )
      (T (apply (CAR e) (evlis (CDR e) al ) al ) )
) )

(DEFUN APPLY (fn args al )
    (COND

      ((EQ e NIL ) NIL )
      ((ATOM fn )
         (COND

           ((MEMBER fn '(CAR CDR CONS ATOM EQ )) (SUBR fn agrs al ))
           (T (APPLY (EVAL fn al ) args al ))
      )    )

      ((EQ (CAR fn ) 'LABEL )
           (APPLY (CADDR fn )
                args
                (CONS (CONS (CADR fn ) (CADDR fn ))
                     al ) ) )

((EQ (CAR fn ) 'FUNARG )
         (APPLY (CDR  fn ) args (CADDR  fn))
      )
      ((EQ (CAR fn ) 'LAMBDA )
                 (EVAL (CADDR fn )
                      (APPEND (PAIR (CADR fn ) args ) al ))
      (T (APPLY (EVAL fn al ) args al ))
```

```
 ) )
```

Определения assoc, append, pair, list – стандартны.

Примерно то же самое обеспечивают eval-p и apply-p, рассчитанные на использование списков свойств атома для хранения постоянных значений и функциональных определений. Индикатор **category** указывает в списке свойств атома на правило интерпретации функций, относящихся к отдельной категории, **macro** – на частный метод построения представления функции. Функция value реализует методы поиска текущего значения переменной в зависимости от контекста и свойств атомов.

```
(defun eval-p (e c)
(cond ((atom e)      (value e c))
     ((atom (car e))(cond ((get (car e) 'category)
                          ((get (car e) 'category) (cdr e) c) )
                         (T (apply-p (car e)(evlis (cdr e) c) c))
      )            )
     (T (apply-p (car e)(evlis (cdr e) c) c))
))

(defun apply-p (f args c)
(cond ((atom f)(apply-p (function f c) args c))
     ((atom (car f))(cond ((get (car f) 'macro)
                          (apply-p ((get (car f) 'macro) (cdr f) c) args c))
                         (T (apply-p (eval f e) args c))
     )             )
     (T (apply-p (eval f e) args c))
))
```

Или то же самое с вынесением общих подвыражений во вспомогательные параметры:

```
 (defun eval-p (e c)
(cond ((atom e)      (value e c))
```

```
    ((atom (car e)) ((lambda (v) (cond (v  (v(cdr e) c) )
                                       (T (apply-p (car e)(evlis (cdr e) c) c))
                                    )) (get (car e) 'category) ) )

    (T (apply-p (car e)(evlis (cdr e) c) c))
))

(defun apply-p (f args c)
(cond ((atom f)(apply-p (function f c) args c))
    ((atom (car f)) ((lambda (v) (cond (v  (apply-p (v (cdr f) c) args c))
                                       (T (apply-p (eval f e) args c))
                          )          ) (get (car f) 'macro)      )          )
    (T (apply-p (eval f e) args c))
))
```

Расширение системы программирования при таком определении интерпретации осуществляется простым введением/удалением соответствующих свойств атомов и функций.

Полученная схема интерпретации допускает разнообразные категории функций и макросредства конструирования функций, позволяет задавать различные механизмы передачи параметров функциям. Так, в языке Clisp различают, кроме обычных, обязательных и **позиционных**, - серийные, необязательные (факультативные) и ключевые параметры. **Виды параметров** обозначаются пометкой **&rest, &optional, &key,** соответственно, размещаемой **перед параметрами в lambda списке формальных аргументов**. При этом **серийный параметр должен быть последним в списке**.

```
(defun funcall (fn &rest agrs) (apply fn args))
```

Необязательный параметр может иметь начальное значение, устанавливаемое по умолчанию, т.е. если этот параметр не задан при вызове функции. При отсутствии начального значения его роль играет Nil.

Примеры 6.2:

```
(defun ex-opt (space &optional dot (line 'x)) (list space 'of dot 'and- line))
(ex-opt 'picture)
(ex-opt 'picture 'circle)
(ex-opt 'picture 'circle 'bars)
```

Ключевые параметры, являясь необязательными, **не зависят еще и от порядка вхождения** в список аргументов. Незаданные аргументы по умолчанию связываются с Nil.

```
(defun keylist (a &key x y z) (list a x y z))
(keylist 1 :y 2) ;= (1 NIL 2 NIL)
```

Примеры 6.3:

```
(defun   LENGTH (L &optional (V 0))
                (cond  ((null L) V)
                       (T (+ 1 (LENGTH (cdr L)))) ))

(LENGTH ‘(1 2) 3) = 5

(defun   REVERSE (L &optional (m Nil))
                (cond  ((null L) m)
                       (T  (REVERSE  (cdr L) (cons (car L) m)) )) )

(REVERSE ‘(1 2 3)) = (3 2 1)
(REVERSE ‘(1 2 3) ‘(5 6)) = (3 2 1 5 6)
```

Такой подход к работе параметрами часто освобождает от необходимости во вспомогательных функциях, что упрощает и определение Eval от обязательности упоминания а-списка. Если воспользоваться сводимостью (COND) к Nil и кое-где воспользоваться отображениями, то определение становится совсем компактным.

Лекция 7**. Детализация базовых функций**

Рассматривается функциональный подход к низкоуровневому программированию на уровне ассемблера, использованный при реализации языка Лисп [1]. Изучается понятие абстрактной машины (secd) для определения операционной семантики языка функционального программирования по Венской методике [8] в форме отображения абстрактного синтаксиса языка Лисп на язык абстрактной машины [3].

Ассемблер

П.Лэндин (P.J.Landin) предложил специальную абстрактную машину SECD, удобную для спецификации машинно-зависимых аспектов семантики Лиспа, которая будет рассмотрена ниже. В первых Лисп-системах для реализации ядра и встроенных операций использовался специальный Лисп-ассемблер LAP [1]. Это обеспечило Лисп-интерфейс для доступа ко всем возможностям оборудования в стиле работы с ассемблером, но по форме как с обычными символьными данными.

Абстрактная Лисп-машина. Система команд

Встраиваемые в ядро интерпретатора операции должны соответствовать стандартным правилам доступа к параметрам и размещения выработанного результата. Таким же правилам должен подчиняться и компилируемый код. Это позволяет формально считать равноправными встроенные и программируемые функции. Компилятор по исходному тексту программы строит код программы, эквивалентный тексту. Особенности процесса компиляции достаточно сложны даже для простых языков, поэтому характеристика результата компиляции часто задается в терминах языково-ориентированных абстрактных машин. Такой подход полезен для решения ряда технологических проблем разработки программных систем (мобильность, надежность, независимость от архитектур и т.п.)

При сравнении императивного и функционального подходов к программированию, П.Лэндин (P.J.Landin) предложил специальную абстрактную машину SECD, удобную для спецификации машинно-зависимых аспектов семантики Лиспа. Подробное описание этой машины можно найти в книгах Хендерсона и Филда-Харрисона по функциональному программированию [3,4].

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). Регистры приспособлены к хранению выражений в форме атомов или списков. Состояние машины полностью определяется содержимым этих регистров. Поэтому функционирование машины можно описать достаточно точно в терминах изменения содержимого регистров при выполнении команд, что выражается следующим образом:

s e c d => s' e' c' d' - переход от старого состояния к новому.

Для характеристики встроенных команд Лисп-интепретатора и результата компиляции программ базового Лиспа понадобятся следующие команды:

LD - ввод данного из контекста в стек;

LDC - ввод константы из программы в стек;
LDF - ввод определения функции в стек;
AP - применение функции, определение которой уже в стеке;
RTN - возврат из определения функции к вызвавшей ее программе;
SEL - ветвление в зависимости от активного (верхнего) значения стека;
JOIN - переход к общей точке после ветвления;
CAR - первый элемент из активного значения стека;
CDR - без первого элемента активное значение стека;
CONS - формирование узла по двум верхним значениям стека;
ATOM - неделимость (атомарность) верхнего элемента стека;
EQ - равенство двух верхних значений стека;
SUB1 - вычитание 1 из верхнего элемента стека;
ADD1 - прибавление 1 к верхнему элементу стека;
STOP - останов.

Стек устроен традиционно по схеме <первый пришел, последний ушел>. Каждая команда абстрактной машины "знает" число используемых при ее работе элементов стека, которые она удаляет из стека и вместо них размещает выработанный результат. Исполняются команды по очереди, начиная с первой в регистре управляющей программы. Машина прекращает работу при выполнении команды "останов", которая формально характеризуется отсутствием изменений в состоянии машины:

s e (STOP) d -> s e (STOP) d

Следуя Хендерсону, для четкого отделения обрабатываемых элементов от остальной части списка будем использовать следующие обозначения: (x . l) - первый элемент списка - x, остальные в списке l. (x y . l) - первый элемент списка - x, второй элемент списка - y, остальные в списке l и т.д. Теперь мы можем методично описать эффекты всех перечисленных выше команд.

```
 s e (LDC q . c) d          -> (q . s) e c d
(a . s) e (ADD1 . c) d      -> (a+1 . s) e c d
(a . s) e (SUB1 . c) d      -> (a-1 . s) e c d
(a b . s) e (CONS . c) d    -> ((a . b) . s) e c d
((a . b) . s) e (CAR . c) d -> (a . s) e c d
((a . b) . s) e (CDR . c) d -> (b . s) e c d
(a . s) e (ATOM . c) d      -> (t . s) e c d
(a b . s) e (EQ . c) d      -> (t . s) e c d
```

где t - логическое значение.

Для доступа к значениям, расположенным в контексте, можно определить специальную функцию N-th, выделяющую из списка элемент с заданным номером N в предположении, что длина списка превосходит заданный адрес.

```
(DEFINE N-th (n list )
   (COND
     ((EQ n 0 )(CAR list ))
     (T (N-th (SUB1 n ) (CDR list ) ))  ))
```

Продолжаем описание команд Лисп-машины.

s e (LD n . c) d -> (x . s) e c d , где x - это значение (N-th n e)

При реализации ветвлений управляющая программа соответствует следующему шаблону:

(... SEL (... JOIN) (... JOIN) ...)

Ветви размещены в подсписках с завершителем JOIN, после которых следует общая часть вычислений. Для большей надежности на время выполнения ветви общая часть сохраняется в дампе - резервной памяти, а по завершении ветви - восстанавливается в регистре управляющей памяти.

(t . s) e (SEL c1 c0 . c) d -> s e ct (c . d)

s e (JOIN) (c . d) -> s e c d

где ct - это c1 или c0 в зависимости от истинностного значения t.

Организация вызова процедур требует защиты контекста от локальных изменений, происходящих при интерпретации тела определения. Для этого при вводе определения функции в стек создается специальная структура, содержащая код определения функции и копию текущего состояния контекста. До этого в стеке должны быть размещены фактические параметры процедуры. Завершается процедура командой RTN, восстанавливающей регистры и размещающей в стеке результат процедуры.

s e (LDF f . c) d -> ((f . e) . s) e c d
((f . ef) vf . s) e (AP . c) d
-> NIL (vf . ef) f (s e c . d)
(x) e (RTN) (s e c . d) -> (x . s) e c d

где f - тело определения, ef - контекст в момент вызова функции,
vf - фактические параметры для вызова функции, x - результат функции.

Упражнение 7.1.
Программа имеет вид:

(LD 3 ADD1 LDC 128 EQ STOP)

Напишите последовательность состояний стека при работе программы и сформулируйте, что она делает.

Ответ:
Данная программа проверяет, меньше ли на 1 значение, хранящееся в контексте по адресу 3, чем заданная в программе константа 128. При ее работе стек проходит следующие состояния:

NIL (3) (4) (128 4) (NIL)

Упражнение 7.2.
Напишите управляющую программу, дающую результат, эквивалентный следующим выражениям:

```
(CADR e )
(EQ (CAR e) 'QUOTE )
(COND ((EQ n 0 )(CAR l )) (T (CONS (SUB1 n ) (CDR l ) ))  ))
```

(Адреса значений e, n, l можно обозначить как @e, @n, @l, соответственно.)

Ответ:

```
( LD @e CDR CAR )
( LD @e CAR LDC QUOTE EQ )
( LD @n LDc 0 EQ SEL (LD @l CAR JOIN ) (LD @n SUB1 LD @l CDR CONS JOIN ))
```

Упражнение 7.3.
Напишите спецификацию команды SET, сохраняющей активное значение стека в контексте по заданному в программе адресу в предположении, что длина списка превосходит заданный адрес.

Выполнение упражнение 7.3:
Нужна функция, заменяющая в списке указанный старый элемент новым.

```
(DEFINE ASS (e n list )
   (COND
     ((EQ n 0 )(CONS e (CDR l )) )
     (T (CONS (CAR l )(ASS e (SUB1 n ) (CDR l ) )))  ))
```

Тогда можно описать команду SET следующим образом:

```
   (x . s) e (SET n . c) d -> s xne c d
```

где xne = (ASS x n e) - новое состояние контекста.

Функциональная модель процессора абстрактной машины.

На этом можно было бы завершить описание реализационного минимума языка Лисп, послужившего основой для функционального программирования. Но мы рассмотрим более детально еще ряд технических аспектов, иллюстрирующих возможности функционального подхода к задачам системного программирования, включая конструирование компиляторов и моделирование парадигм программирования.

Здесь же определим общий цикл выполнения программ на SECD, что даст возможность поэкспериментировать с абстрактной машиной . Вышеописанная система команд способна выполнять результаты компиляции функциональных программ, написанных на элементарном Лисп. В следующей лекции более подробно будет описан компилятор. А пока можно поупражняться с расширениями SECD.

Объявление системы команд машины и их определения:

```
(put 'a 'SYM '(lambda () (setq st (cons (caar st)
                                        (cdr st)))
      ))
```

```
(put 'd 'SYM '(lambda () (setq st (cons (cadar st)
                                        (cdr st)))
      ))
(put 'at 'SYM '(lambda () (setq st (cons
                                    (atom (car st))
                                    (cdr st)))
      ))
(put 'co  'SYM'(lambda () (setq st (cons
                                   (cons (car st) (cadr st))
                                   (cddr st)))
      ))
(put 'equ 'SYM '(lambda () (setq st (cons
                                    (eq (car st) (cadr st))
                                    (cddr st)))
      ))
(put 'sum 'SYM '(lambda () (setq st (cons
                                    (+ (car st) (cadr st))
                                    (cddr st)))
      ))
(put 'def 'SYM '(lambda () (setq st (cons
                                    (- (car st) (cadr st))
                                    (cddr st)))
      ))
(put 'mlt 'SYM '(lambda () (setq st (cons
                                    (* (car st) (cadr st))
                                    (cddr st)))
      ))

(put 'ldc 'SYM '(lambda ()
; CP - продолжение программы вслед за LDC
                    (setq st (cons (car cp)
                                   st))
                    (setq cp (cdr cp))
; CP - без константы, переданной в стек
))

; Определение интерпретатора машины

(defun secd (lambda ()(cond ((null cp)
                                    (print "?end-of-program!"))
                     ((eq (car cp) 'STOP)
                       (print "!normal-finish!"))
                     ((get (car cp)'SYM)
                      (command (car cp) (cdr cp))
                      (secd))
                     (T (print "?error-command!")
                       (setq cp (cdr cp))
                       (secd))
                  )))
(defun command (lambda (acp dcp) (setq cp dcp)
                        (print acp)
                        (apply (get acp 'SYM)'())
```

```
                    (prsecd)
                    (read)
          ))

;  Вывод на экран состояния машины
(defun prsecd (lambda()(prst)(prcp)))
(defun prst (lambda ()(print (list "stack:=" st ))))
(defun prcp (lambda ()(print (list "control:=" cp ))))

; Задание состояния машины :
;         ST  - стек
;         CP  - управляющая программа
;         ENV - контекст
;         DM  - память для восстановления состояния
(setq st '())
(setq cp '(ldc 1 ldc 2 ldc 3 mlt def ldc 4 equ stop ))
(secd)
(prsecd)
(read)
(system)

(setq st '(a ((1 2) (4 5)) 3 6 7 8 9 11 13 12 14 21 25 9 1 0))
(setq cp '(at de co at equ stop sum def mlt stop))
(prsecd)
(secd)
(prsecd)
 (setq st (cddr st))
(setq cp (cdr cp))
(prsecd)
(secd)
(prsecd)
 (system)
 (apply (get 'a 'SYM)'())
(prst)
(setq st (cdr st))
(apply (get 'd 'SYM)'())
(prst)
(setq st (cdr st))
 (apply (get 'at 'SYM)'())
(prst)
(setq st (cdr st))
 (apply (get 'co 'SYM)'())
(prst)
 (apply (get 'at 'SYM)'())
(prst)
(setq st (cdr st))
 (setq st (cdr st))
 (apply (get 'equ 'SYM)'())
(prst)
(setq st (cdr st))
 (apply (get 'sum 'SYM)'())
(prst)
```

```
(setq st (cdr st))
 (apply (get 'def 'SYM)'())
(prst)
(setq st (cdr st))
 (apply (get 'mlt 'SYM)'())
(prst)
(setq st '(12 12 12) )
(prst)
(setq st (cdr st))
 (apply (get 'equ'SYM)'())
(prst)
 (system)
```

Лекция 8. **Компиляция функциональных программ**

В данной лекции изучаются требования к компиляции функциональных программ и строится определение компилятора. Для Лиспа такое определение написано на Лиспе, как и определение интерпретатора в виде отображения абстрактного синтаксиса языка на язык абстрактной машины [8]. Разложение программы на функции с разным уровнем отладки является отправной точкой при выборе оптимизационных решений. Компиляция программ рассматривается как один из методов оптимизации процессов, осуществляемый как символьное преобразование – трансляция с исходного языка высокого уровня на язык низкого уровня, допускающий оптимизирующую кодогенерацию.

Компилятор и требования к коду программы

Описанная в предыдущей лекции абстрактная машина SECD удобна для **спецификации низкоуровневых аспектов** семантики Лиспа. Такой подход позволяет не загромождать определение компилятора малосущественными деталями, добиться его прозрачности, но главное, такое определение может быть **машинно-независимым и переносимым [3]**.

Для функционального стиля в программировании характерно стремление снять непринципиальные ограничения на применение и построение функций. Для этого приходится сдерживать привычные для многих областей применения разграничения, а также смягчать стандартные границы при организации процессов в системах программирования. В этом отношении следует отметить:

1) единое пространство функций, их аргументов и всех обозначений, роль которых определяется по контексту при интерпретации форм;

2) использование функциональных переменных, значение которых конструируется (вычисляется) в процессе их интерпретации. Это позволяет вводить частичные определения, уточняемые по мере необходимости;

3) самоопределение основных механизмов символьной обработки и, следовательно, открытый характер системы программирования, поддерживающей функциональное программирование;

4) мягкость пространственно-временных ограничений, без точных численных оценок по отдельным параметрам;

5) поощрение рекурсивных определений;

6) предельная **уточняемость и детализируемость определений**, управление временем их существования и выполнения.

Лисп-компилятор - это программа, написанная на Лисп, которая транслирует С-выражения, определяющие функции, в машинные подпрограммы. Это средство **оптимизации**, позволяющее
программам работать во много раз быстрее, чем было бы при простой интерпретации.

Когда компилятор вызывается для компиляции функций, он находит определение функции в списке свойств названия функции. Компилятор транслирует найденное С-

выражение в С-выражение, которое представляет подпрограмму на языке ассемблера. Ассемблер после этого ассемблирует код программы. Затем в списке свойств размещается ссылка на код программы.

Опыт показывает, что скомпилированная программа может работать в 2-100 раз быстрее, чем интерпретируемая программа, в зависимости от ее природы. Скомпилированные программы могут быть и экономичнее с точки зрения расхода памяти, требуя 50-80% от полного объема.

Основная часть компилятора - транслятор или функция, отображающая С-выражения, которые обозначают функции, на язык ассемблера. Единственное основание для того, чтобы рассматривать компилятор
как псевдо-функцию, состоит в том, что он изменяет свойства названий функций по завершении компиляции.

Лисп-компилятор имеет уникальную историю. Он развивался пошаговым образом (**метод раскрутки**) [1].

1) Компилятор был написан и отлажен как Лисп-программа, состоящая из набора определений функций в виде С-выражений. Это позволяет компилировать любые Лисп-программы и строить специализированные расширения Лисп-интерпретатора.

2) Компилятору была дана команда скомпилировать себя самого. Данная операция называется раскруткой (**bootstrapping**). На это потребовалось более 5 минут на IBM 7090, поскольку значительная часть компилятора в основном интерпретировалась. В результате было создано эффективное расширение Лисп-системы, способное компилировать Лисп-программы и строить расширения Лисп-интерпретатора.

3) Чтобы исключить повторение медленной раскрутки при дальнейших шагах установки системы, весь **код компилятора** заново был введен на языке ассемблера.

4) Была создана системная лента, и компилятор стал загружаемым на уровне ассемблера.

5) Процесс раскрутки был повторен до полного формирования кода Лисп-системы.

Компилятор вызывается псевдо-функцией **Compile**. Аргумент Compile - список названий функций, которые следует компилировать. Каждый атом в списке должен иметь определение функции в своем списке свойств до компиляции.

Обработка каждой функции происходит в три шага. Во-первых, С-выражение, задающее функцию, транслируется в текст на уровень ассемблера. При отсутствии С-выражения, компилятор сообщает об этом и переходит к другой функции. Во-вторых, текст программы на уровне ассемблера транслируется в код программы. И, наконец, если никаких ошибок не обнаружено, то С-выражение функции может быть удалено из списков свойств. Когда некоторые ошибки указывают на появление необъявленной переменной, компилятор предупреждает об этом и продолжает работу. Такая **диагностика** будет дополнительно уточнена при анализе значений переменной.

При написании большой Лисп-программы лучше отлаживать отдельные функции, используя интерпретатор, а **компилировать только те из них, которые уже хорошо изучены**.

Программист, планирующий применять компилятор, должен обратить внимание на следующие моменты.

1) Нет необходимости компилировать все функции, которые используются лишь эпизодически. Интерпретатору доступны скомпилированные функции. Компилированные функции, использующие интерпретируемые функции, могут вычислять их непосредственно при счете.

2) Порядок выполнения компиляции не имеет значения. Даже нет необходимости определять все функции до тех пор, пока они не понадобятся при счете. (Исключение из этого правила - специальные формы. Они должны быть определены до того, как компилируется их вызов.)

3) При динамическом использовании Label результирующая функция не может быть скомпилирована полностью.

4) **Свободные переменные** в компилируемых функциях должны объявляться до компиляции функций.(См. лекцию 7).

Последнее требование проясняет роль типового контроля в стандартных, ориентированных на компиляцию без интерпретации, системах программирования. Компиляция всех объектов осуществляется без анализа фактических данных, а это и означает, что на момент компиляции переменные, как правило, являются свободными. Интерпретация располагает более полной информацией, связывающей необходимые для вычислений переменные с конкретными значениями, тип которых определен.

Компилятор и ассемблер могут быть удалены из **комплекта Лисп-системы**. В целом существует **механизм пакетов**, позволяющий управлять составом функций и объектов, включаемых в комплект. При удалении части системы освободившаяся память может быть присоединена к списку свободной памяти. Имеются реализации, в которых выделено минимальное ядро системы, все остальные функции загружаются по мере необходимости, а мусорщик может рассматривать память от неиспользуемых функций как свободную.

Требования к компиляции Лисп-программ

Рассмотрим особенности Лисп-программ, которые необходимо учесть при определении компилятора и подготовке программ к компиляции.

Прежде всего - **свободные переменные**. Переменная связана, если она встречается в списке lambda или prog, а также let, do, loop и т.п. Все несвязанные переменные свободны.

Пример8.1:

```
(LAMBDA (A) (PROG (B)
                  S  (SETQ B A)
                     (COND ((NULL B) (RETURN C)))
                     (SETQ C (CONS (CAR A) C))
                       (GO S) ))
```

A и B - связанные переменные, C - свободная переменная.

Когда переменная используется как свободная, это значит, что она должна быть связана в другой функции на более высоком уровне. При интерпретации функций может быть обнаружена переменная, не связанная вообще, о чем система известит пользователя соответсвующим **диагностическим сообщением** об ошибке.

При компиляции новые диагностические сообщения не появляются, а переменная получает значение Nil.

Существуют разные типы переменных в компилируемых функциях: обычные и специальные - **Special**. Тип Special необходимо объявить до компиляции. Все необъявленные переменные рассматриваются как обычные.

При трансляции функций в подпрограммы концепция переменных отображается в **распределение памяти**, в которой размещаются значения аргументов. Для обычных переменных распределение памяти - это стек. Другие функции не могут знать адреса таких переменных, что и не позволяет рассматривать их как свободные.

SPECIAL-переменные размещаются в списке свойств. Это допускает реализацию с заданием **фиксированных адресов ячеек**. Когда такая ячейка связана, можно старое значение вытолкнуть в стек, а новое разместить по старому адресу. При выходе из области действия текущей связи старое значение восстанавливается. При использовании такой свободной переменной адрес SPECIAL- ячейки может быть доступен другим функциям.

SPECIAL-переменные объявляются псевдо-функцией **DECLARE** с индикатором SPECIAL, вслед за которым расположен список переменных. Эта функция устанавливает индикатор SPECIAL в списках свойств перечисленных имен и создает ячейку для хранения значения переменной. Эта псевдо-функция для компилятора весьма существенна. Она может действовать и при интерпретации, и при прогоне компилированных программ.

(declare ... (special v1 v2 ...) ...)

При повторном объявлении одной и той же SPECIAL-переменной компилятор выделит другую ячейку, формально это будут различные переменные.

Специальные переменные удобны для коммуникации между компилируемыми программами, но не всегда могут четко служить коммуникации между интерпретируемыми и компилируемыми программами.

Еще один тонкий аспект - **функциональные константы.**
Рассмотрим следующее определение, использующее C-выражение:

```
(YDOT (LAMBDA (X Y) (MAPLIST X (FUNCTION
                    (LAMBDA (J) (CONS (CAR J) Y)) ))))
```

```
(ydot (A B C D) X)
;=((A . X) (B . X) (C . X) (D . X))
```

За словом **FUNCTIO**N располагается функциональная константа. Если ее вынести как самостоятельную функцию, то формально J - связанная переменная, а Y - свободная. Не исключено, что свободная переменная где-то будет объявлена как специальная или общедоступная, хотя она должна быть связана внутри ydot.

Теперь про **функциональные аргументы.**
MAPLIST может быть определен следующим образом:

```
(MAPLIST (LAMBDA (L FN) (COND
                          ((NULL L) NIL)
                          (T (CONS (FN L) (MAPLIST  (CDR L)
FN))) )))
```

Переменная FN - это **связанный функциональный аргумент**. Дело в том, что его значение - определение функции.

Трассировка скомпилированных программ
Trace - работает с компилированными программами согласно следующим ограничениям:

1) Trace может быть объявлена после компиляции и не требует повторной компиляции программы.

2) Trace не отслеживает прямые переходы на функции, созданные в обход атомов, т.е. выделенные на уровне ассемблера или кода без встраивания в общую информационную систему Лиспа – таблицу символов, хранящую свойства атомов.

"**В правильном выражении всегда достаточно скобок**, чтобы избежать синтаксической неоднозначности "[3]. Используя это утверждение Хендерсона как эпиграф, а в Лиспе со скобками все в порядке, можно уверенно приступить к определению собственно компилятора.

Компиляция. Операционная семантика

Функциональный подход к программированию наиболее убедительно выражен в **Венской методике определения языков программирования**. Эта методика разработана в конце 60-х годов [8]. Основная идея - использование **абстрактного синтаксиса и абстрактной машины** при определении **семантики** языка программирования. **Конкретный синтаксис** языка отображается в абстрактный - **анализ**, а абстрактная машина может быть реализована с помощью конкретной машины - **кодогенерация**, причем и то, и другое может иметь небольшой объем и невысокую сложность. Сущность определения языка концентрируется в виде так называемой **семантической функции языка**, выполняющей переход от абстрактного синтаксиса к абстрактной машине - **трансляцию**.

При такой архитектуре компилятор можно рассматривать как три комплекта функций, обеспечивающих анализ программы, ее трансляцию и кодогенерацию. Главная задача анализа - обнаружить основные понятия и выделить, вывести или вычислить по тексту программы значения компонентов структуры, представляющей собой абстрактный синтаксис программы. Эта работа сводится к набору **распознавателей и**

селекторов, названия которых могут быть выбраны в зависимости от смысла понятий, составляющих программу, а **реализация варьируется** в зависимости от конкретного синтаксиса языка. Тогда при любом конкретном синтаксисе разбор программы выполняется тем же самым определением, что и анализ ее абстрактного представления, которое играет роль спецификации. Любое определение анализа выглядит как перебор распознавателей, передающих управление композициям из селекторов, выбирающих существенные **компоненты** из анализируемой программы и заполняющих поля определенной структуры или значениями, или программами их вычисления. Содержимое полей предназначено для генерации кода программы, эквивалентной исходному тексту программы,
а заодно и ее абстрактной структуре.

Например, если форму PROG рассматривать как представление абстрактного синтаксиса для подмножества языка Паскаль, содержащего переменные, константы, арифметические операции и сравнения, пустой оператор, присваивание, последовательное исполнение операторов, условный оператор и безусловный переход goto, то необходим набор распознавателей, выявляющих эти понятия, и селекторов, выделяющих их характеристики. Селекторы
имеют смысл лишь при истинности соответствующего распознавателя.

Использование списочных форм в качестве абстрактного синтаксиса позволяет все распознаватели свести к анализу головы списка

Таблица 8.1. Абстрактный синтаксис операторов

(перем X)	x
(конст C)	c
(плюс A1 A2)	(A1 + A2)
(равно A1 A2)	(A1 = A2)
(пусто)	
(присвоить X A)	x := a;
(шаги S1 S2)	S1; S2;
(если P ST SF)	if p then ST else SF;
(пока P S)	while p do S;

- сокращение для шаблона цикла

Все селекторы сводятся к композиции car-cdr, выполняемой после соответствующего распознавателя. Так, в приведенных выше формах поля X, C, A1, S1, P можно выделить селектором, определенным как (lambda (fm) (car(cdr fm))) – выделение второго элемента списка, а поля A2, S2, ST, S, расположенные третьими в списках - как (lambda (fm) (car(cdr(cdr fm)))), поле SF - как (lambda (fm) (car(cdr(cdr(cdr fm))))). Такие определения практически не требуют
отладки, работают с первого предъявления.

Определение семантической функции, обеспечивающей корректную трансляцию абстрактного синтаксиса программы в ее **абстрактный код**, требует реализации соответствия между именами и их значениями в зависимости от контекста и **предшествующих определений**.

При интерпретации такое соответствие представлялось ассоциативным списком, в котором хранятся связи вида Имя-Смысл, преобразуемые по принципу стека,

естественно отражающего динамику вызова функций. При компиляции не принято сохранять имена на время исполнения программы: их роль выполняют сопоставленные именам адреса. Поэтому вместо а-списка вида ((а . 1)(в . 2)(с . 3)...) применяется два списка (а в с ...) и (1 2 3 ...), хранящих имена и их значения на согласованных позициях. Обрабатываются эти два списка синхронно: оба удлиняются или сокращаются на одинаковое число элементов.

Можно переписать Eval-Apply с соответствующей коррекцией и определить функцию подстановки, заменяющую имена их значениями из синхронного списка.

Определение Eval-Apply особенно компактно в стиле р-списка. Иерархию определений можно организовать с помощью блоков Flet со списками определений для шаблонов **(перем конст плюс равно)** и отдельно для **(пусто присвоить шаги если пока).**

Важно обратить внимание на учет изменения контекста при последовательном выполнении шагов программы, а также на несовпадение порядка в тексте с очередностью выполнения композиций функций.
Формально операторы могут рассматриваться как функции, преобразующие полное состояние памяти V. Пусть функция E списочному представлению оператора сопоставляет эквивалентную ему Лисп-функцию, вызываемую в контексте (declare (special N)).

Таблица 8.2. Примеры функциональной реализации операторов и выражений.

c (конст C)	(lambda (v)c)
x (перем X)	(lambda (v) (assoc-i X N v)) N - свободная переменная, задающая список имен, известных в программе
(A1 + A2 (плюс A1 A2)	(lambda (v) (+(E A1)(У A2)))
(A1 = A2) (равно A1 A2)	(lambda (v)(=(E A1)(У A2)))
(пусто)	(lambda (v)v) Состояние памяти неизменно
x := a; (присвоить X A)	Замена происходит по указанному адресу (lambda (v)(replace N v X (E A)))
S1; S2; (шаги S1 S2)	(lambda (v) (E S2 (E S1 v)))
if e then ST else SF;	(lambda (v) (funcall (cond (((E P)v)

(если P ST SF)	(E S1)) (T(E S2))) v)
while e do S; (пока P S)	Циклу соответствует безымянная функция, строящая внутренний контекст: (lambda (W) ((lambda (v) (cond (((E P)v)(w ((E S)v))) (T v))) (lambda (v) (cond (((E P)v) (w ((E S)v))) (T v)))))

Денотационная семантика ЯП (математическая) определяет семантику языка в терминах соотношений между множествами. При задании **операционной семантики** важно отследить корректность обработки порождаемых структур данных, что может быть сформулировано как **свойство чистого результата**. Согласно этому свойству задана четкая дисциплина манипуляций со стеком при обработке данных: каждая операция берет из стека в точности все свои аргументы и обязательно располагает в нем свой единственный результат.

При определении компилятора на уровне абстрактной машины должно быть выделено описание реализационного минимума языка Лисп, послужившего базой для раскрутки Лиспа и основой для функционального программирования. Необходимо лишь ввести некоторые ограничения, гарантирующие при переходе к низкоуровневому программированию сохранение важнейших свойств функциональных программ. Эти ограничения формулируются как чистый **результат правильного выражения:**

- все аргументы убраны из стека;

- результат выражения записан в стек.

Определение Лисп-компилятора на Лиспе

```
(defun compile-(s)(append (comp- s Nil)'(Ap Stop)))

(defun comp- (S N)(cond

  ((atom S)   (list 'LD (adr S N)))

  ((eq (car S)'QUOTE) (list 'LDC (cadr S)))
  ((eq (car S)'CONS)    (append (comp-(caddr S)N) (comp-(cadr S)N) 'CONS))
  ((eq (car S)'CAR)      (append (comp-(cadr S)N) 'CAR))
  ((eq (car S)'+)          (append (comp-(cadr S)N) (comp-(caddr S)N) 'ADD))

  ((eq (car S)'IF)        (let  ( (then (list (comp-(caddr S)N) '(JOIN)))
                                  (else (list (comp-(cadddr S)N) '(JOIN))))
                               (append (comp-(cadr S)N) (list 'SEL then else))))

  ((eq (car S)'LAMBDA)  (list 'LDF (comp-(caddr S) (append (cadr S) N)) 'RTN))

  ((eq (car S)'LET)        (let*   ((args (value (cddr S)))
```

```
                                        (mem (cons (var (cddr S)) N))
                                        (body (append (comp-(cadr S)mem) 'RTN)))
                                    ((append (map #'(lambda(x)(comp- x N)) args)
                                             (list body 'AP)))))

   ((eq (car S)'LABEL) (let* ((args (value (cddr S)))
                                    (mem (cons (var (cddr S)) N))
                                    (body (append (comp-(cadr S)mem) 'RTN)))
                                ((append '(DUM) (map #'(lambda(x)(comp- x mem))
args)
                                                 (list 'LDF body 'RAP)))))

   (T (append (map #'(lambda(x)(comp- x N)) (cdr S))
              (list body 'AP)) )
))
```

Лекция 9. **Реализационные решения**

Приведены принципы реализации системы функционального программирования и описаны структуры данных, удобные для динамической обработки информации. Проиллюстрированы методы “сборки мусора” и других реализационных механизмов функциональных языков программирования, давющих на современном оборудовании достаточно высокие эксплуатационные характеристики. Рассмотрена комплектация ядра системы, технические детали организации ее рабочего цикла и функциональные средства оперативного мониторинга фактического состава системы.

Сборка системы и ее рабочий цикл

Моделирование Лиспа или другого языка программирования на идеальном базовом Лиспе вполне служит иллюстрацией определения **операционной семантики** языков программирования [8].

Традиционно система программирования для языка Лисп содержит пару интерпретатор-компилятор. Интерпретатор содержит элементы, реализация которых описывается в трансляционных терминах: внутреннее представление, структура памяти, реализация двоичных деревьев и т. п. Компилированная программа содержит интерпретируемые элементы: например, обращения к файловой системе и другим элементам ОС. На практике достоинства интерпретации проявляются при отладке программ, а преимущества компиляции - при эксплуатации готового программного продукта. Подробное обсуждение этой темы заслуживает отдельного разговора.

Теория программирования утверждает, что определение компилятора может быть выведено из определения интерпретатора методами смешанных вычислений [9]. Компилятор по такой методике получается как остаточная программа при смешанном вычислении интерпретатора. Теоретически для определения языка программирования достаточно построить определение интерпретатора, хотя практичность реальной системы программирования обычно обеспечивается оптимизирующей компиляцией и кодогенерацией программ. [9] Но здесь речь идет не об эффективном компиляторе, а лишь о понятном описании семантики языка программирования.

Ядро интерпретатора языка Лисп может быть реализовано следующим образом:

- выбирается реализация списков в виде бинарных деревьев, листья которого рассматриваются как атомы, а узлы используются для выстраивания списков (левые ветви - элементы списка, правые - продолжение списка или конец списка, т.е. пустой список);

- выбирается реализация атомов как объектов, внутренняя структура которых при определении и исполнении функций не всегда существенна, но при необходимости доступна специальным операциям;

- встраивается специальный атом NIL, являющийся реализацией пустого списка ();

- встраивается операция, связывающая различные данные с атомами, и ассоциируется с атомом LABEL;

- определяются правила доступа к параметрам встроенных операций с размещением их результата и встраивается специальная операция (монитор), выполняющая применение операций к правильно размещенным аргументам (SUBR);

- встраивается операция, строящая из атомов и списков более сложные структуры (списки и узлы из любых элементов), и ассоциируется с атомом CONS;

- встраиваются операции, выполняющие разбор и анализ структур, и ассоциируются с атомами CAR, CDR, ATOM, EQ, представляющими эти операции;

- встраиваются специальные операции (псевдо-функции), выполняющие блокировку вычислений, выбор ветви и конструирование определений функций,
и ассоциируются с атомами QUOTE, COND и LAMBDA, соответственно;

- универсальная функция, сбственно интерпретатор, ассоциируется с EVAL;

- определяется **рабочий цикл** передачи данных интерпретатору и вывода результата интерпретации.

Такое ядро представляет собой машинно-зависимую часть Лисп-интерпретатора. Встраивание операции в ядро системы - это включение в его реализацию исполнимого кода, который является реализацией этой операции. Адрес такого кода ассоциируется с именем атома, с помощью которого будет организовано выполнение операции при интерпретации программ.

При ассоциировании атомов с произвольной информацией можно использовать специально организованный ассоциативный список, построенный из пар, содержащих атомы и их определения. Например, ассоциативный список

((T . T) (NIL . NIL))

обеспечивает значение T и NIL в соответствии с семантикой базового Лиспа, список

((ОДИН . 1) (ДВА . 2))

дает символьные имена числовым значениям, а список

((ГОЛОВА . CAR) (ХВОСТ . CDR) (УЗЕЛ . CONS))

- синонимы для обозначения базовых операций Лиспа.

Ассоциативный список работает как стек: при многократных определениях работает самое новое, т.е. расположенное ближе к началу списка. Если мы знаем адреса кода операций, встроенных в ядро системы, то можем соответствие между именами операций и адресами их кода хранить в ассоциативном списке. Можно считать, что начальное состояние ассоциативного списка содержит таблицу соответствия между именами и адресами операций.

Основой определения интерпретатора является функция EVAL (evaluation), вычисляющая произвольные выражения языка с учетом состояния ассоциативного списка AL. Спецификация такой функции для базового Лиспа может быть проиллюстрирована следующими примерами:

```
(EVAL NIL AL )                  => NIL
(EVAL T AL )                    => T
(EVAL 'X AL )                   => (CADR (ASSOC X AL))
(EVAL '(QOUTE EXPR ) AL )               => EXPR
(EVAL '(COND ((T YES) ... )) AL )       => YES
(EVAL '(CSETQ X Y EXPR ) AL )           => (EVAL EXPR (CONS '(X Y ) AL ))
(EVAL '(CAR A) '((A (1 2 3))(NIL NIL)) )  => 1
```

Иначе выражения получают значение в результате применения некоторой функции, стоящей в голове списка, к ее аргументам, что выполняется другой важной частью определения интерпретатора - функцией APPLY. Для ее работы необходима функция, вычисляющая значения аргументов с учетом состояния ассоциативного списка.

При написании на базовом Лиспе определения функции EVAL согласно приведенной выше спецификации, способной от данного списочного представления выражения E перейти к его значению с учетом заданного ассоциативного списка AL, хранящего значения атомов, мы несколько отступаем от ранее данных определений, с тем чтобы более явно выделить линии сборки системы.

```
(DEFUN EVAL  (e al )
    (COND
        ((MEMBER e '(NIL T )) e )
        ((ATOM e ) ((LAMBDA (v )
                          (COND (v (CADR v ) )
                          (T (ERROR 'undefdvalue ))
                       )  )
                    (ASSOC e al )
         )              )

       ((EQ (CAR e) 'QUOTE ) (CAR (CDR e )) )
       ((EQ (CAR e) 'COND  )  (EVCON (CDR e ) al ) )
       ((EQ (CAR e) 'LET )      (EVAL (CADDDR e )
                                   (CONS (CONS (CADR e )
                                               (CONS (EVAL (CADDR e ) al ) NIL)
                                   ) al )
        )                          )

       (T (APPLY (CAR e) (EVLIS (CDR e) al ) al ) )
)   )
```

В этом функциональном определении используется имя функции APPLY, применяющей произвольную функцию к ее аргументам при заданном ассоциативном списке. ERROR – псевдо-функция, выдающая заданные диагностические сообщения.

Определение функции APPLY работает при условии, что функция SUBR осуществляет применение встроенных функций к их аргументам, заданным в виде списка значений.

```
(DEFINE APPLY  (fn args al )
    (COND
       ((EQ fn NIL) NIL)
```

```
        ((MEMBER fn '(CAR CDR CONS ATOM EQ ))
                        (SUBR (CADR (ASSOC fn al)) args ))

        ((ATOM fn ) (APPLY (EVAL fn al ) args al ))
        ((EQ (CAR fn ) 'LAMBDA )
             (EVAL (CADDR fn )
                      (APPEND (PAIR (CADR fn ) args ) al )) )
         ((EQ (CAR fn) 'LABEL )
                        (APPLY (CADDR fn) args (CONS (CDR fn ) al )) )

          (T (ERROR- 'undefdfunction))
)     )
```

Обратите внимание, что при поиске атома в ассоциативном списке в EVAL мы допускаем отсутствие связанного с атомом значения и сообщаем об этом диагностикой с помощью функции ERROR. При выборе адреса встроенной функции в APPLY мы рассчитываем, что все известные функции реализованы, и их адреса размещены в ассоциативном списке – за правильность выбора имен встроенных функций отвечает программист.

Можно еще поработать с таким определением интерпретатора и более четко локализовать его зависимость от четырех различных категорий объектов: самоопределяемые атомы - (NIL T 1 2 3 ...), базовые операции над данными языка, обрабатывающие предварительно вычисленные аргументы, - (CAR CDR CONS ATOM EQ ...), специальные функции, управляющие обработкой аргументов без их предварительного вычисления, - (QUOTE COND LET ...) и конструкторы функций, строящие функциональные значения из обычных выражений, - (LAMBDA LABEL...).

Желающие могут поэкспериментировать с самодельным интерпретатором, превращая его в модель ядра любого языка программирования, используя какую-нибудь Лисп-систему, например GNU Clisp (с точностью до имен отдельных функций).

Упражнение 9.1. Пусть READ и PRINT - встроенные функции, обеспечивающие прием с клавиатуры и вывод на экран произвольных данных языка Лисп. Напишите определение рабочего цикла интерпретации последовательности выражений.

Ответ.

```
(DEFINE CIRCLE (al ) (PRINT (EVAL (PRINT (READ )) al ))
                (CIRCLE al ) )
```

"Но оно же зациклится!" - скажете вы и будете правы.
Но это не помешает эксперименту, ведь в нашем распоряжении имеется конец файла Ctrl-Z или встроенная операция завершения процесса типа BYE, EXEC, SYSTEM либо
что-то подобное.

Упражнение 9.2. Полученные 50 строк определения Лисп-интерпретатора и его вспомогательных функций содержит 1263 символа, если не считать пробелы в начале строк. Попробуйте написать сравнимое по объему и не менее содержательное определение на каком-нибудь знакомом вам языке программирования.

Реализация динамической памяти и структур данных

После обсуждения схемы функционирования Лисп-интерпретатора и его сборки из машинно-зависимого конструктива пришло время поговорить о структурах данных, используемых при реализации списков и атомов.

Обычно при обработке программ в памяти располагаются разносортные результаты, возникающие при разборе и анализе текста программы и ее данных, при построении ее внутреннего кода и при формировании результата. В Лисп-системах традиционно при всех этих видах работ принято придерживаться принципов **логики здравого смысла**:

- новые значения строятся **на новом месте**,

- предпочитаются **интересы малых программ**,

- **автоматизируется повторное использования памяти**, что на первых шагах разработки освобождает программиста от необходимость уделять внимание вторичным по отношению к его задаче проблемам распределения памяти.

Кроме того, почти исключается необходимость рискованных присвоений, они в программах заменяются именованием.

Память обычно распределена по блокам, содержащим ряд слов, образующих структуры данных. Физический объем памяти, логическая длина данных и состав информации, полезной для продолжения вычислений, могут существенно различаться. Минимальные потери в результативности работы с памятью дает **динамическая обработка бинарных деревьев –** без простоев из-за незаполненности части полей. Каждый узел такого дерева имеет небольшой объем, достаточный для хранения двух **типизированных указателей** (CAR и CDR, левый и правый). Типизация указателей нужна для оперативного контроля соответствия данных и операций по их обработке. NIL, атомы, списки, числа, строки - все это **реализационно различимые типы данных**. (Утверждение о бестиповости Лиспа имеет отношение лишь к отсутствию акцента на статическое связывание в тексте программы имен переменных с типами их значений. Для компиляции приходится дополнять Лисп-программы сведениями о типах значений переменных, но далеко не каждая программа доживает до компиляции.) Лиспу свойственна функциональная классификация значимых типов данных, т.е. именно реализационно различимых.

Реализация бинарных деревьев или **односвязных списков** описана в классических курсах по программированию, а реализацию атомов мы рассмотрим подробнее. Эффективность приведенного выше определения интерпретатора с использованием ассоциативного списка существенно зависит от числа различимых атомов в программе. Такая зависимость обычно смягчается механизмом **функций расстановки** (хэш-функций), обеспечивающим доступ к информации по ключу. В качестве ключа используется имя атома. В результате вся связанная с атомом информация становится легко достижимой. Структура такой информации называется **списком свойств атома.** Она представляет собой чередование так называемых "индикаторов" и "значений" свойств. Число свойств атома неограничено. Свойства бывают встроенные в систему или вводимые программистом. Значения атомов, адреса встроенных операций, определяющие выражения функций - примеры встроенных

свойств. Встроенные операции типа **LET, LABELS** обычно используют списки свойств. Обработка таблицы, связывающей атомы и их списки свойств, как правило, реализационно зависима. Методы задания и изменения свойств работают подобно обычным присваиваниям. Псевдо-функция **PUT** задает индикатор и соответствующее ему новое значение свойства атома, а функция **GET** обеспечивает доступ к свойству атома, соответствующему заданному индикатору. Теперь с помощью списков свойств мы могли бы добиться **точного соответствия семантики** констант и определений функций их спецификации в базовом Лиспе, но не будем отвлекаться на это.

Самым интересным, можно сказать революционным, механизмом работы с памятью в Лиспе, бесспорно, стала "**сборка мусора**". С начала 60-х годов методам такой работы посвящены многочисленные исследования, которые продолжаются до наших дней и сильно активизировались в связи с включением похожего механизма в реализацию языка Java.

Общая идея всех таких методов достаточно проста:

- пока памяти хватает, о ней можно не беспокоиться и располагать новые данные в новых блоках памяти;

- если памяти вдруг не оказалось, то надо выполнить "сборку мусора", в процессе которой, возможно, найдутся ставшие бесполезными для программы блоки;

- если память нашлась, ее снова **можно тратить беззаботно**.

Специальная программа "сбор мусора" выполняет **анализ достижимости** всех блоков памяти просто пометкой узлов, видимых из конечного числа рабочих регистров системы программирования. К таким регистрам относятся промежуточные результаты вычислений, активная часть стека, ассоциативный список, таблица атомов и др. После пометки все непомеченные узлы объединяются в **список свободной памяти**, передающий их для повторного использования новым вызовам функции **CONS**. Такая автоматизация не лишена недостатков, но они обнаруживаются лишь в сравнительно сложных процессах, требования которых мы сейчас сознательно не учитываем.

Обычно с машиной связывается представление о блоках информации фиксированного объема, таких как слова, байты, регистры. Функциональное программирование нацелено на более крупные построения - **структуры данных не ограниченной заранее длины**.

Значительный резерв производительности функциональных программ дают деструктивные функции, являющиеся формальными аналогами чистых функций, но строящих свой результат в той же памяти, в которой размещены аргументы. Выигрыш получается ценой снижения надежности.

Таблица 9.1. Соответствие строгих функций и их деструктивных эквивалентов

Append	nconc
Subst	nsubst
Remove	delete
Reverse	nreverse
Union	nunion

Реальный состав системы и внешний мир

Реальный состав системы и возможности ее компонентов можно исследовать с помощью специальных функций, предоставляющих информацию о включенных в систему объектах и их свойствах.

(**apropos** ‘nm ‘package) – печатает информацию обо всех символах, имена которых содержат подстроку “nm”. Второй аргумент, если он указан, ограничивает эту информации заданным пакетом.

(**describe** ‘fn) – дает описание роли объекта в системе.

(**symbol-plist** ‘fn) – выдает перечень всех свойств объекта.

(**documentation** ‘fn ‘function) – выдает документацию по объекту. При желании ее можно задавать вместе с символьным определением функций как строки-комментарии, начинающиеся с двойной точки с запятой “;;”, расположенного сразу вслед за первой строкой вида “(DEFUN name-function”.

(**dribble** ‘path) - направляет в файл протокол работы с Лисп-интерпретатором.

(**step** expr) – обеспечивает пошаговый режим интерпретации выражения с выдачей разультатов каждого шага.

(**setq** inpt (open file-in :direction :input)) – заведение переменной для обозначения открытого потока.

(read inpt) – чтение из файла, открытого как поток.

(print (print eval-val prtcl) outpt) – запись данного eval-val в два разных файла.

(open file-in :direction **:input**) - открытие файла на чтение.

Далее следуют три варианта открытия файлов на запись:

(open "output" :direction **:output** :if-exists :rename :if-does-not-exist :create)
(open "protocol" :direction **:output** :if-exists :overwrite :if-does-not-exist :create)
(open "history" :direction **:output** :if-exists :append :if-does-not-exist :create)

(close prtcl) – закрытие потока.

Особенности работы с файлами, основные приемы их открытия, задания специфики их функционирования и обмена данными с обычными символьными объектами иллюстрирует пример организации учебного цикла работы с Clisp, использующего пошаговую интерпретацию программ.

```
(defun eval-protocol () (prog (eval-val)
; выражения припасены в файле "input.lsp"

metka
 (print '> prtcl)
```

```
 (setq eval-val (eval
              (list 'STEP
; пошаговое вычисление выражения
      (print (print
            ( if (eq 'eof (setq rinpt
                              (read inpt NIL 'eof) ))
               (return(close inpt))
               rinpt)
       prtcl) hstry)
  )))

; прочитанное записывается в файлы "protocol" и "history"
(print '- prtcl)
;(print eval-val)
(print (print eval-val
              prtcl) outpt)
; результат интерпретации в файлах "protocol" и "output"
(go metka)
))

(defun help ( function-name )
 (ed (string function-name )) )

(defun step1 (file-in)
(prog ()

    (setq inpt (open file-in :direction :input ))
  (setq outpt (open "output" :direction :output :if-exists :rename
                                    :if-does-not-exist :create))
 (setq prtcl (open "protocol" :direction :output
                                   :if-exists :overwrite
                                   :if-does-not-exist :create))
 (setq hstry (open "history" :direction :output :if-exists :append
                                     :if-does-not-exist :create ))
 (print '"****** new-session ******" hstry)

; цикл прервется по концу файла ввода
(eval-protocol)

(close prtcl)
(close hstry)
(close outpt)
))

(step1 "input.lsp")
; интерпретируемые  выражения припасены в файле "input.lsp"
```

Лекция 10. От ФП к ООП

Анализируется содержательное родство между функциональным (ФП) и объектно-ориентированным (ООП) программированием. Рассмотрены основные принципы ОО-программирования и проанализированы схемы их реализации в рамках функционального программирования на базе ряда структур данных на примере простой модели ОО-языка, встраиваемого в Лисп. Отмечены особенности CLOS, поддерживающей ООП на GNU Clsip [7]. Реализация методов обработки объектов заданного класса сводится к отдельной категории функций, вызов которых управляется анализом принадлежности аргумента классу.

Общее представление о декомпозиции программ

Сборка программы из автономно развивающихся компонентов опирается на формулировку достигаемой ими цели, понимание которой гарантирует не только корректность полученного результата, но и рациональность его использования. Формулировать цели частей программы - процесс нетривиальный. В его основе лежат весьма различные подходы к классификации понятий.

ООП объединяет в рамках единой методики организации программ классификацию на базе таких понятий как **класс объектов, структура данных и тип значений**. Тип значений обычно отражает спектр низкоуровневых реализационных средств, учет которых обеспечивает **эффективность кода программы**, получаемого при компиляции. Структура данных обеспечивает **конструктивность построений**, гарантирует доступ к частям, из которых выстроено данное любой сложности. Класс объектов характеризуется **понятным контекстом**, в котором предполагается их корректная обработка. Обычно контекст содержит определения, структуру объектов и их свойства.

Текст программы одновременно представляет и динамику управления процессами, и схему информационных потоков, порождаемых при исполнении программы. Кроме того, последовательность написания программы и ее модификации по мере уточнения решаемой задачи могут соответствовать логике, существенно отличающейся от процесса выбора системных и реализационных решений. **Функциональное программирование скрывает** сложность таких деталей управления процессами путем **сведения его к общим функциям**, преобразующим любые аргументы в определенные результаты. Модификации осуществляются в процессе изменения состава ветвей непосредственно в определении каждой функции.

ООП структурирует множество частных методов, используемых в программе, в соответствии **с иерархией классов объектов**, обрабатываемых этими методами, реализуемыми с помощью функций и процедур, в предположении, что определяемые в программе **построения могут локально видоизменяться** при сохранении основных общих схем информационной обработки. Это позволяет выполнять модификации объявлением новых подклассов и дописыванием методов обработки объектов отдельных классов без радикальных изменений в ранее отлаженном тексте программы.

При анализе задач, решаемых в терминах объектов, некоторая деятельность описывается так, что постепенно продумывается **все, что можно делать с объектом** данного класса. Потом в программе достаточно лишь упоминать методы обработки объекта. Если методов много, то они структурированы по иерархии классов, что

позволяет автоматизировать выбор конкретного метода. На каждом уровне иерархии можно немного варьировать набор методов и структуру объектов. Таким образом, **описание программы декомпозируется на интерфейс и реализацию**, причем интерфейс скрывает сложность реализации так, что можно обозревать лишь необходимый для использования минимум средств работы с объектом.

Типичная гипотеза при программировании работы с объектами:
Объект не изменен, если на него не было воздействий из программы.

Но реальность зачастую требует понимания и учета более сложных обстоятельств, что может существенно продлить время жизни программы или ее компонентов. В таком случае удобно предоставлять объектам большую свободу, сближающую их с понятием **субъекта, описание которого содержит все, что он может делать**. Программа может давать ему команды-сообщения и получать ответы-результаты.

Связь методов с классами объектов позволяет вводить одноименные методы над разными классами объектов (**полиморфизм**), что упрощает представление логики управления: на уровне текста программы можно не выражать распознавание принадлежности объекта классу, это сделает система программирования. Таким образом обычно реализовано сложение, одинаково изображаемое для чисел, строк, векторов, множеств и т.п. Фактически субъектом является суперкласс, объединяющий классы объектов, обрабатываемые одноименными методами, т.е. функциями одного семейства. Так, при организации сложения можно считать, что существует суперкласс “слагаемые”, которое умеют складываться с другими слагаемыми. При этом они используют **семейство функций**, реализующих сложение. В зависимости от полноты семейства результат может быть получен или не получен. Семейство легко пополняется добавлением одноименных функций с новыми комбинациями типов параметров.

Дальнейшее развитие подходов к декомпозиции программ связано с выделением отдельных аспектов и шагов при решении сложных задач. Понятие **аспекта** связано с различием точек зрения, позволяющим описывать решение всей задачи, но отражать в описании только видимые детали. По мере изменения точек зрения могут проступать новые детали, до тех пор, пока дальнейшая детализация не утрачивает смысл, т.е. улучшение трудно заметить или цена его слишком высока. Так, представление символьной информации в Лиспе выделено в отдельный аспект, независимый от распределения памяти, вычисление значений четко отделено от компиляции программ, понятие связывания имен с их определениями и свойствами не зависит от выбора механизмов реализации контекстов исполнения конструкций программы.

Понятие **шага** обычно связывается с процессом раскрутки программ, оправданным в тех случаях, когда целостное решение задачи не может гарантировать получение приемлемого результата в нужный срок – это влечет за собой непредсказуемо большие трудозатраты.

Удобный подход к организации программ “отдельная работа отдельно программируется и отдельно выполняется” успешно показал себя при развитии операционной системы UNIX [ ] как работоспособный принцип **декомпозиции** программ. Но существуют задачи, например реализация систем программирования, в которых прямое следование такому принципу может противоречить требованиям к производительности. Возможен компромисс “отдельная работа программируется отдельно, а выполняется взаимосвязано с другими работами” [А. Л. Фуксман], что

требует совмещения декомпозиции программ с методами сборки - **комплексации** или **интеграции** программ из компонентов. Рассматривая комплексацию как еще одну “отдельную” работу, описываемую, например, в терминах управления процессами, можно констатировать, что эта работа больше сказывается на требованиях к уровню квалификации программиста, чем на объеме программирования. При достаточно объективной типизации данных и процессов, возникающих при декомпозиции и сборке программ определенного класса, строят библиотеки типовых компонентов и разрабатывают компонентные технологии разработки программных продуктов - Corba, COM/DCOM, UML и т.п.. Одна из проблем применения таких библиотек – их обширность.

При реализации экспериментальных языков и систем программирования цель применения **раскрутки** - минимизация трудозатрат, основанная на учете формальной избыточности средств языков программирования. Можно выделить небольшое **ядро**, на основе которого методично программируется все остальное. Каждый шаг реализации по схеме раскрутки должен обеспечивать:

- уменьшение трудоемкости последующих шагов,
- отладку прототипов сложных компонентов,
- подготовку демонстрационного материала.

Выбор конретных шагов можно соотнести с декомпозицией определения языка программирования на синтаксические и семантические, функциональные и машинно-ориентированные, языково-ориентированные и системные аспекты. При такой декомпозиции можно на первых шагах как бы “снять” синтаксическое и семантическое разнообразие языка, как имеющее чисто технический характер. Именно в этом смысл выделения элементарного Лиспа. Такая методика может быть успешна при освоении любого класса, информацию о котором можно представить в виде частично формализуемых текстовых и графовых форм.

Дальнейшие шаги раскрутки можно упорядочить по актуальности реализации компонентов, обеспечивающих положительную оценку системы пользователем. Это позволяет развить представление о принципах декомпозиции программ более созвучно ООП: “ отдельная работа обнаруживается независимо от остальных работ”. Наиболее устойчивая и значимая классификация работ по реализации системы программирования может быть установлена как обеспечение механизмов надежного функционирования информационных систем – “отдельная работа – это отдельное средство повышения надежности программирования”. Ряд таких средств можно выделить в любом языке программирования: вычисления, статическое и динамическое управление процессами, логика выбора хода обработки информации, дисциплина именования памяти и доступа к расположенным в ней данным, правила укрупненных воздействий на блоки данных и иерархию процессов, диагностика и обработка событий – перечень открытый.

ООП на Лиспе

При переходе от обычного стандартного программирования с ООП связывают радикальное изменение способа организации программ. Это изменение произошло под давлением роста мощности оборудования. ООП взламывает традиционное программирование по многим направлениям. Вместо создания отдельной программы,

оперирующей массой данных, приходится разбираться с данными, которые сами обладают поведением, а программа сводится к простому взаимодействию новой категории данных - "объекты".

Чтобы сравнить дистанцию с функциональным программированием, далее рассмотрим самодельный встроенный в Лисп **объектно-ориентированный язык** (ОО-язык), обеспечивающий основы ООП. Встраивание ОО-языка - идеальный пример, показывающий характерное применение функционального программирования, при котором типичные понятия ООП отображаются в фундаментальные абстракции.

Практически ООП - это организация программ в терминах методов, классов, экземпляров и наследования. Почему стоит писать программы таким способом? Основной выигрыш - **программы легче изменять**. Если мы хотим изменить способ манипулирования каким-либо объектом некоторого класса, то мы изменяем лишь метод этого класса. Если мы хотим сделать нечто подобное объекту, но отличающееся в отдельных чертах, мы можем создать подкласс объектов, а уже для подкласса поменять его отдельные черты. Если программа написана тщательно, то можно добиться того, чтобы все такие модификации производились даже без просмотра ранее написанного исходного текста программы.

В Лисп есть разные способы размещать **коллекции свойств**. Кроме уже рассмотренного механизма «список свойств атома» существует более общий механизм - **хэш-таблицы.** Можно размещать свойства как записи в нее. В [7] приведен блестящий пример реализации ООП на базе хэш-таблиц. Тогда отдельное свойство можно получать из таблицы как формы:

```
(gethash 'color obj)
```

Поскольку функции являются данными объекта, постольку мы должны их размещать так же, как и свойства. Это значит, что мы должны завести еще методы, позволяющие вызывать данный метод объекта как исполнение свойства данного объекта.

```
(funcall (gethash 'move obj) obj 10)
```

Мы можем определить под эту идею синтаксис подобный языку **Smalltalk:**

```
(defun tell (obj message &rest args)
(apply (gethash messmage obj) obj args))
```

что позволяет сказать объекту, чтобы он переместился на 10 шагов, в форме:

```
(tell obj 'move 10)
```

Фактически успех наследования обеспечивает единственная особенность Лиспа: все это работает благодаря реализации рекурсивной версии функции GETHASH. Таким образом, определение данной функции нам сразу дает все три основные черты ООП.

Техническая легкость результата - это трюк, но не программистский трюк, а концептуальный. Не будем забывать о том, что Лисп по своей природе уже был ОО-языком, или даже чем-то более общим изначально. Все что нужно в Лиспе для ОО-

программирования - это создать новый интерфейс для уже существующих в Лиспе абстракций.

До сих пор речь шла о простом наследовании - объект имел только одного предка. Но можно получать и **множественное наследование** построением списка свойств предков со слегка измененной rget. При простом наследовании, когда нам надо выбрать некоторое свойство объекта, мы сразу ищем рекурсивно его предшественников. Если собственно объект не содержит искомую информацию, мы обозреваем его предка и т.д. При множественном наследовании мы хотим делаем примерно то же самое, но работа усложняется тем, что предшественники объекта могут образовывать граф вместо дерева.

Логически нет необходимости особо различать классы, объекты и их вхождения - удобно все обрабатывать по общей схеме. Но на практике весьма непроизводительно всякий раз просматривать сотни предшественников, если известно, что достаточно менее десятка. Такую оптимизацию выполняет механизм экземпляров. **Экземпляр** имеет одного предка – его класс. Его обработка не требует списка предшественников. Это не ведет к потере гибкости, т.к. при необходимости можно переопределить состояние объекта и как бы конвертировать экземпляр в класс.

Реализация ООП с помощью хэш-таблиц обладает слегка парадоксальной окраской: гибкость у нее больше, чем надо, и за большую цену, чем можно позволить. Уравновесить это может подобная реализация на базе простых векторов. Этот переход показывает, как функциональное программирование дает **новое качество** "на лету". В опорной реализации фактически не было реализационного разделения объектов на экземпляры и классы. Экземпляр – это был просто класс с одним-единственным предком. При переходе к векторной реализации разделение на классы и экземпляры становится реальным. В ней становится невозможным превращать экземпляры в классы простым изменением свойства.

Модель ОО-языка

Более прозрачная модель ООП получается на базе обычных списков свойств, заодно иллюстрирующая глубинное родство ФП и ООП:

```
(defun classes (cl) (cond
                (cl (cons (cdar cl) (classes (cdr cl)))) ))

; вывод формулы классов аргументов из определения параметров метода
; Nil - произвольный класс

(defun argum (cl) (cond
              (cl (cons (caar cl) (argum (cdr cl)))) ))

; вывод списка имен аргументов из определения параметров метода

(defun defmet (FMN c-as expr)
               (setf (get FMN 'category) 'METHOD)
               (setq ML (cons(cons(cons FMN (classes c-as))
                                  (list 'lambda (argum c-as) expr) ) ML))
       FMN )
```

```
; объявление метода и расслоение его определения
; для удобства сопоставления с классами аргументов

(defun defcl (NCL SCL FCL ) ; имя, суперкласс и поля/слоты класса
            (setq ALLCL (cons NCL ALLCL))
            (set NCL (append FCL SCL))  )

; значением класса является список его полей, возможно, со значениями

(defun ev-cl (vargs) (cond

; вывод формата фактических аргументов для поиска метода их обработки

            (vargs (cons (cond
                              ((member (caar vargs) ALLCL) (caar vargs))  )
                         (ev-cl (cdr vargs))))  ))
; Nil если не класс

(defun m-assoc (pm meli) (cond (meli (cond ((equal (caar meli) pm)(cdar meli))
                          (T(m-assoc pm (cdr meli)))))))

; поиск подходящего метода, соответствующего формату классов данных

(defun method (MN args &optional c)
    (apply (m-assoc (cons mn (ev-cl args)) ML)
           args c))

; если метода не нашлось, в программе следует выполнить приведение
; параметров к нужному классу

(defun instance (class &optional cp) (cond

; подобно Let безымянная копия контекста

              (class (cond ((atom (car class))(instance (cdr class) cp))
                           ((assoc (caar class) cp) (instance (cdr class) cp))
                           (T(instance (cdr class) (cons (car class) cp)))
))    ) cp)

(defun slot (obj fld) (assoc fld obj))

; значение поля объекта
```

Остается лишь слегка подкорректировать определение Лисп-интерпретатора, заодно используя необязательные параметры, осовобождающие от простейших вспомогательных определений, например от обязательного вхождения накопителей типа ассоциативного списка.

```
(DEFUN evcon- (c &optional a)
;                       |_________ключ, объявляющий
;                                  необязательные параметры
  (COND
```

```
   ((eval-p (car (car  c)) a) (eval-p (car (cdr (car c))) a) )
   ( T          (evcon- (cdr c) a)     ) ))

(DEFUN evlis- (m &optional a)
 (COND
   ((EQ m Nil)   Nil)
   ( T (cons (eval-p (car m) a)
                (evlis- (cdr m) a)    ) )   ))

(defun eval-p (e &optional c)
(cond ((atom e)      (value e c))
     ((atom (car e))(cond

            ((eq (car e) 'QUOTE) (car (cdr e)))
            ((eq (car e) 'COND)  (evcon- (cdr e) a))
            ((get (car e) 'METHOD)
               (method (car e) (evlis(cdr e)) c) )
            ( T (apply-p (car e)(evlis- (cdr e) c) c))
      )           )
     (T (apply-p (car e)(evlis- (cdr e) c) c))
))

(defun apply-p  (f args &optional c)
(cond ((atom f) (apply-p (function f c) args c))
     ((atom (car f))(cond ((get (car f) 'macro)
                      (apply-p (apply-p (get (car f) 'macro) (cdr f) c) args c))

(T (apply-p (eval f c) args c))
     )             )
     (T (apply-p (eval f c) args c))
))

(print (eval-p 1))
(print (eval-p 'a))
(print (eval-p '(quote b)))
(print (eval-p '(cond (Nil 6)(T 88)  )))
(print (eval-p '(car '(3 2))))
```

Средства ООП в CLOS на базе стандарта Clisp

Показанный в [7] пример работает по первому аргументу (выбор подходящего метода рассчитан на то, что достаточно разобраться с одним аргументом), CLOS делает это на всех аргументах, причем с рядом вспомогательных средств, обеспечивающих гибкий перебор методов и анализ классов объектов.

Классы и экземпляры объектов

(**defclass** ob () (f1 f2 ...))

Это означает, что каждое вхождение объекта будет иметь поля-слоты f1 f2 ...
(**Слот** – это поле записи или списка свойств.)

Чтобы сделать **представлителя класса,** мы вызываем общую функцию:

(setf c (**make-instance** 'ob))

Чтобы задать **значение поля,** используем специальную функцию:

(setf (**slot-value** c) 1223)

До этого значения полей были не определены.

Свойства слотов

Простейшее определение слота - это его имя.
Но в общем случае слот может содержать список свойств.
Внешне свойства слота **специфицируются как ключевые параметры функции.**

Это позволяет задавать **начальные значения.**

Можно объявить слот **совместно используемым.**

:allocation :class

Изменение такого слота будет **доступно всем экземплярам** объектов класса.

:documentation - свойство слота

Можно задать тип элементов, заполняющих слот.

Суперкласс

Нет необходимости все новые слоты создавать в каждом классе

```
;oop-compile

(defclass expr ()
((type :accessor td) (sd :accessor ft))
(:documentation "C-expression"))

(defclass un (expr)
;                  \_____суперкласс для унарных форм

  ((type :accessor td) ;; можно убрать ???
   (sd :accessor ft))  ;; можно убрать ???
(:documentation "quote car *other *adr"))

(defclass bin (expr)
    ((type :accessor td)
     (sd :accessor ft)
     (sdd :accessor sd)  )
(:documentation "cons + lambda let"))
```

```
(defclass trio (expr)
  ((type :accessor td)
   (sd :accessor ft)
; (bin) ;; не объявлять sdd ???
   (sdd :accessor sd)
   (sddd :accessor td) )
(:documentation "if label"))

(defmethod texrp ((x expr) (nt atom))
    (setf (slot-value x 'type) nt)
    (setf (td x) nt) ;;--;; variant
(:documentation "объявляем тип выражения"))

(defmethod spread ((hd (eql 'QUOTE))
                   (tl expr))
  (let ( (x (make-instance 'un)) )
        (setf (ft x) (car tl))
        (setf (td x) hd)
) (:documentation "распаковка выражения"))

(defmethod compl ((hd (eql 'QUOTE))
                   (tl expr))
                  (list 'LDC tl)
) (:documentation "сборка кода"))

(defmethod compl ((hd (eql 'CAR))
                   (tl expr))
                  (append (compl(ft tl) N) '(CAR))
) (:documentation "сборка кода"))

(defmethod spread ((hd (eql 'CONS))
                   (tl expr))
  (let ( (x (make-instance 'bin)) )
        (setf (ft x) ( car tl))
        (setf (sd x) ( cadr tl))
        (setf (td x) hd)
) (:documentation "распаковка выражения"))

(defmethod compl ((hd (eql 'CONS))
                   (tl bin) N )
                  (append (compl(sd tl) N) (compl(ft tl) N) '(CONS))
) (:documentation "сборка кода"))

(defmethod compl ((hd (eql '+))
                   (tl bin) N )
                  (append (compl(ft tl) N) (compl(sd tl) N) '(ADD))
) (:documentation "сборка кода"))

(defmethod spread ((hd (eql 'IF))
                   (tl expr))
  (let ( (x (make-instance 'trio)) )
```

```
        (setf (ft x) ( car tl))
        (setf (sd x) ( cadr tl))
        (setf (td x) ( caddr tl))
        (setf (td x) hd)
) (:documentation "распаковка выражения"))

(defmethod compl ((hd (eql 'IF))
                  (tl expr) N   )
    (let ( (then (list (compl(sd tl) N) '(JOIN)))
           (else (list (compl(td tl) N) '(JOIN))) )
         (append (compl(ft tl) N) (list 'SEL then else) )
)(:documentation "сборка кода"))

(defmethod parh ((x expt))
  (let (ftx (ft x))
    (cond
      ((atom ftx) (spread 'ADR ftx))
      ((member (car ftx) '(QUOTE CAR CONS + IF LAMBDA LABEL LET))
       (spread (car ftx) (cdr ftx))
      (T (spread 'OTHER ftx) ))
)(:documentation "шаг разбора"))

;====test==========
(setf test1 (make-instance 'expr))
(texpr test1 'expr)
(setf (slot-value test1 'sd) (read))
()

(setf e1 (make-instance 'expr))
(setf e2 (make-instance 'expr))
(setf e3 (make-instance 'expr))
(print (tf e2))
(setf (slot-value e3 'type) 'expr)
(print (tf e3))
(setf (slot-value e3 'sd) '(quote const))

(defmethod ep ((x expr))
        ((lambda (xt)
            (setf (slot-value x 'type) xt))
          (car (slot-value x 'sd) )))
(print (ep e3))
(print (tf e3))
(print (td e3))
(print (sd e3))

(defmethod ep-q ((x (eql 'quote)) (y expr))
(setf y (make-instance 'un)))
(setf (slot-value y 'type) 'quote)
(setf (slot-value y 'sd) y)
))

(print (tf (e3 'sd)))
```

```
(print (tf e1))
(print(setf (slot-value e1 'type) (tf e1)))
(setf (slot-value e2 'sd) 'atom1)
(print (tf (sd e2)))

(print(setf (slot-value e3 'sd) '(quote const)))
(print (tf e3))
```

CLOS, естественно, использует модель обобщенных функций, но мы написали независимую модель, используя более старые представления, тем самым показав, что концептуально ООР – это не более чем перефразировка идей Лиспа. ООП - это одна из вещей, которую Лисп изначально умеет делать. Для функционального стиля программирования в переходе к ООП нет ничего неожиданного. Это просто небольшая конкретизация механизмов перебора ветвей функциональных объектов.

Более интересный вопрос, что же нам еще может дать функциональный стиль и лисповская традиция реализации систем программирования?

Ответу на этот вопрос посвящены три следующие лекции.

Лекция 11. **Варианты, последовательности, множества**

Описаны приемы организации недетерминированных вычислений в рамках функционального стиля программирования. Реализация программ с возвратами, перебор вариантов, откат влекут за собой расширение семантического базиса механизмами обработки прерываний. Анализируется соответствие точности решения задач и уровня их изученности. Исследуются связь диагностической интерпретации и средств логического программирования [10]. Обработка множеств, последовательностей и хэш-таблиц дает материал для простых примеров.

Недетерминированные процессы

Есть мнение, что однозначное решение задачи в виде четкого алгоритма над хорошо организованными структурами и упорядоченными данными - результат аккуратной, тщательной работы, пытливого и вдумчивого изучения класса задач и требований к их решению. Эффективные и надежные программы в таких случаях – естественное вознаграждение. Но в ряде случаев природа задач требует свободного выбора одного из вариантов - выбор произвольного элемента множества, вероятности события при отсутствии известных закономерностей, псевдо случайные изменения в игровых обстановках и сценариях, поиск первого подходящего адреса для размещения блока данных в памяти, лингвистический анализ при переводе документации и художественных текстов и т.д. При отсутствии предпочтений все допустимые варианты равноправны, и технология их отладки и обработки должна обеспечивать формально равные шансы вычисления таких вариантов. (Похожая проблема характерна для организации обслуживания в сетях и выполнения заданий операционными системами. Все узлы и задания сети должны быть потенциально достижимы, если нет формального запрета на оперирование ими.)

Представление вариантов в чем-то подобно определению ветвлений, но без предикатов, управляющих выбором ветви. В некоторых языках, например, учебно-игрового характера, можно указать вероятность выбора варианта. В языках логического и генетического программирования считают возможным прямой перебор вариантов, сопоставляемых с образцами, и организацию возвратов при неудачном варианте.

В отличие от множества элементов, набор вариантов не требует одновременного существования всех составляющих. Поэтому и программирование вариантов можно освободить от необходимости формулировать все варианты сразу. В логическом программировании можно продумывать варианты отношений между образцами формул постепенно, накапливая реально встречающиеся сочетания, как и методы обработки классов объектов в ООП. Содержательно такой процесс похож и на уточнение набора обработчиков прерываний на уровне оборудования. Кроме основной программы, выполняющей целевую обработку данных, отлаживается коллекция диагностических реакций и процедур продолжения счета для разного рода неожиданных событий, препятствующих получению результата программы.

Обычно понятие алгоритма и программы связывают с детерминированными процессами. Но эти понятия не очень усложняются, если допустить **недетерминизм, ограниченый конечным числом вариантов**, так что в каждый момент времени из них существует

только один вариант. По смыслу выбор **варианта** похож на выбор произвольного элемента множества.

{ a | b | c } = э{ a, b, c }

Чтобы такое понятие промоделировать обычными функциональными средствами, нужны дополнительные примитивы. Например, чтобы определить **выбор произвольного элемента** из списка L, можно представить рекурсивное выражение вида:

(любой L) = {(car L)
| (любой (cdr L)) }

Если варианты в таком выражении рассматривать как равноправные компоненты, то не ясно, как предотвратить преждевременный выбор пустого списка при непустом перечне вариантов.

Чтобы решить эту задачу, вводится специальная форма **ESC (ТУПИК)**, действие которой заключается в том, что она как бы "старается" **по возможности не исполняться**. Иными словами, при выборе вариантов предпочитаются варианты, не приводящие к исполнению формы ESC. (Такая же проблема возникает при обработке пустых цепочек в грамматиках. Аналогичным образом эта проблема решена при моделировании процессов интерпретированными сетями Петри [] - соглашением о приоритете раскрашенных переходов в сравнении с пустыми.)

Уточненное таким образом определение выбора произвольного элемента списка можно представить формулой вида:

(любой L) = { (car L)
| (любой (cdr L))
| (if (nl L) ESC) }

В какой-то момент L становится пустым списком, и его разбор оказывается невозможным. Тогда действует ESC.

Следует иметь в виду, что **варианты не образуют иерархии**. Их аксиоматика подобна так называемой **упрощенной теории множеств** Принципиальная особенность - совпадение предикатов принадлежности и включения.

Другие построения, характерные для теории множеств:

{ x | P(X) } - множество элементов, обладающих свойством P.

Определение вида

(F x) = {(if (P (car L))
(cons (car L) (F (cdr L))))
| (if (nl L) esc) }

недостаточно, т.к. порождаемые варианты элементов, удовлетворяющих заданому свойству, существуют в разные моменты времени и могут не существовать одновременно. Чтобы получить **все варианты одновременно**, требуется еще один примитив ALL, обеспечивающий накопление всех реально осуществимых вариантов.

```
(F x) = (ALL {(if (P ( car L ))
                  (cons ( car L) (F ( cdr L))) )
             | (if (nl L) esc) } )
```

Пересечение множеств A и B:

```
( all ( lambda (x y) {(if (= x y) x)
                     | esc  }) (любой A) (любой B) )
```

Логические связки:

Логика McCarthy (компьютерная)

a & b

```
(if (not a) Nil b)
```

b вычисляется лишь при истинном a, что не всегда соответствует интуитивным ожиданиям.
Более надежны варианты, исключающие зависимость от порядка перебора :

```
(( lambda x {  (if (not x) Nil )  | esc }) {a | b} )
```

Аналогичная проблема возникает при построении **ветвлений**

```
(cond (p1 e1) (p2 e2 ) …)
```

```
( ( lambda L {(cond (( eval ( caar L) AL)    ( eval ( cadr L) AL ) )) | ESC })
                                    ( любой ((p1 e1)  (p2 e2) ...)))
```

Поддержка вариантов, каждый из которых может понадобиться при построении окончательного результата, находит практическое применение при организации высокопроизводительных вычислений. Например, мультиоперации можно организовать с исключением зависимости от порядка отдельных операций в **равносильных формулах**:

a+b+c = (a+b)+c = a+(b+c) = (a+c)+b

```
((lambda (x y z) {(if (< (+ x y) K) (+(+ x y) z)) | esc})
        {(a b c) | (b c a) | (c a b)})
```

В книге Хендерсона приведено обобщение абстрактной машины, поддерживающее на базовом уровне работу с вариантами с использованием дополнительного дампа, гарантирующего идентичность состояния машины при переборе вариантов [3].

Реализация недетерминированных моделей

Необходимая для такого стиля работы инструментальная поддержка обеспечивается в GNU Clisp механизмом обработки событий **throw-catch,** для которого следует задать примерно такое взаимодействие:

```
(defun vars (xl)(catch 'esc
; перебор вариантов до первого тупика
        (cond
; vars not Nil
             ((null xl)(escape))
          ((car xl) (cons (car xl)(vars (cdr xl))))
)))

(defun escape () (throw 'esc Nil))
; сигнал о попадании в тупик

(print(vars ()))
(print(vars '(a)))
(print(vars '(a b c)))
(print(vars (list 'a 'b (vars ()) 'c)))
```

В этой схеме THROW играет роль прерывания процесса, а CATCH – обработчика прерываний. Их взаимодействие синхронизировано с помощью **тега**, идентифицирующего уровень, на котором расположена **ловушка** для соответствующего прерывания. При этом есть возможность указать передаваемое “наверх” значение. Содержательно такая схема взаимодействия похожа на PROG-RETURN, с той разницей, что отсутствует зависимость от расположения в тексте программы. Получается, что в любом выражении можно выполнить разметку ветвей на нормальные и тупиковые. **Тупики** можно связать с различными тегам и выставить ловушки на заданные теги. При попадании в тупик формируется значение всей структуры, размещенной внутри ловушки.

Используя тупики и ловушки, можно собрать все беступиковые варианты или организовать перебор вариантов до первого беступикового. Первое можно сделать с помощью отображений (map), а второе - первый подходящий – слегка модифицированным evcon, можно с добавочной ловушкой на прерывание при достижении успеха. Модификация заключается в использовании другой версии eval. Ловушка нужна для приостановки вычисления независимых вариантов, когда уже найден один подходящий, т.е. не заводящий в тупик.

Более **сложно обеспечить равновероятность выбора** вариантов. Наиболее серьезно возможность такой реализации рассматривалась в проекте языка SETL [12]. Похожие механизмы используются в языках, ориентированных на конструирование игр, таких как Grow, в которых можно в качестве условия срабатывания команды указать вероятность.

В задачах искусственного интеллекта работа с семантическими сетями, используемыми в базах знаний и экспертных системах, часто формулируется в терминах фреймов-слотов (рамка-щель), что конструктивно очень похоже на работу со списками свойств. Каждый объект характеризуется набором поименованных свойств, которые, в свою очередь, могут быть любыми объектами. Анализ понятийной системы, представленной таким образом, обычно описывается в недетерминированном стиле.

Следует отметить неисчерпаемый ряд задач, при решении которых результативно используются недетерминированные модели:

1) Обоснование упорядочений в традиционных алгоритмах – выделяется доалгоритмический уровень, на котором просто анализируются таблицы возможных решений и постепенно вырабатываются комплекты упорядочивающих условий и предикатов.

2) Переформулировка задач и переопределение алгоритмов с целью исключения необоснованных упорядочений – одна из типовых задач оптимизации, особенно при переходе от обычных программ к параллельным. Приходится выяснять допустимость независимого исполнения всех ветвей и управляющих их выбором предикатов.

3) Обобщение идеи абстрактных машин с целью теоретического исследования, экспериментального моделирования и прогнозирования недетерминированных процессов на суперкомпьютерах и многопроцессорных комплексах
(многопроцессорная машина Тьюринга и т.п.).

4) Конструирование учебно-игровых программ и экспериментальных макетов, в которых скорость реализации важнее, чем производительность.

5) Описание и реализация недетерминизма в языках сверхвысокого уровня, таких как Planner, Setl, Sisal, Id, Haskell и др.

6) Недетерминированные определения разных математических функций и организация их обработки с учетом традиции понимания формул математиками.

7) Моделирование трудно формализуемых низкоуровневых эффектов, возникающих на стыке технических новинок и их массового применения как в научных исследованиях, так и в общедоступных приборах.

8) Обработка и исследование естественно языковых конструкций, речевого поведения, культурных и творческих стереотипов, социально-психологических аспектов и т.п.

9) Организация и разработка распределенных вычислений, измерений, Grid-технологий, развитие интероперабельных и телекоммуникационных систем и т.п.

Используемые при исследовании и решении таких задач модели дают богатый материал для развития нового поколения информационных систем, концептуальную основу которых можно изучать с помощью небольших функциональных программ.

Принятая при решении таких задач техника сопоставления с образцом в значительной мере может быть осуществлена как работа с необязательными параметрами, что иллюстрирует эффективная версия определения сцепления списков [7]:

```
(defun append (&optional first &rest others )
        (if (null others) first
                        (nconc (copy-list first)
                                (apply #'append others)) )
)
```

В этой версии исключено копирование первого списка, когда других списков нет, и копии сцепляемых списков производятся лишь однократно.

Обработка множеств и последовательностей

При реализации недетерминированных моделей обычно используются средства обработки множеств и последовательностей. В современных системах функционального программирования такие средства достаточно разнообразны. Здесь приведены лишь наиболее очевидные:

Member – выделяет часть списка, начиная с заданного объекта, Nil – если такого объекта в списке нет.
(member ‘a (b a c)) ;= (a c)
(member ‘d (b a c)) ;= Nil

Set-difference – строит список элементов первого аргумента, не входящих во второй аргумент. Имеет деструктивный аналог - nset-difference.

Set-exlusive-or - строит список элементов первого или второго аргумента, но не входящих в оба сразу. Имеет деструктивный аналог - nset-exlusive-or.

Union – объединение множеств - строит список элементов первого или второго аргумента. Имеет деструктивный аналог – nunion.

Intersection – пересечение множеств - строит список элементов первого, входящих во второй аргумент. Имеет деструктивный аналог – nintersection.

Delete - строит последовательность элементов второго аргумента за исключением совпадающих с первым аргументов. Имеет деструктивный аналог – remove.

(delete 1 ‘(1 2 1 3 1 4)) ;= (2 3 4)

Concatenate – строит новую последовательность заданного типа из своих аргументов, начиная со второго, при этом копирует их, кроме последнего. Для списков имеет деструктивный аналог – nconc.

Elt – выдает элемент последовательности по заданному номеру.

Find – отыскивает заданный символ в последовательности, можно управлять направлением поиска.

Sort – упорядочивает последовательность по заданному предикату.
(sort '(1 2 1 3 1 4) #'<) ;= (1 1 1 2 3 4)

Map – отображает с помощью данной функции ряд последовательностей в новую последовательность типа, заданного первым аргументом. Отображающая функция – второй аргумент. Кратность применения отображающей функции определяется длиной кратчайшего аргумента, начиная с третьего. Имеет деструктивный аналог map-into, строящий результат из первого аргумента.

Reverse – обращает последовательность. Имеет деструктивный аналог nreverse.

Position – выдает номер позиции первого вхождения заданного символа в последовательность.

Substitute – выполняет систематическую замену "старого" символа на "новый" в последовательности. Имеет деструктивный аналог - nsubstitute.

Maphash – методично применяет отображающую функцию двух аргументов к каждой паре из ключа и соответствующего значения в хэш-таблице.

Лекция 12. Управление процессами

Рассматривается эффективное обобщение процесса информационной обработки, вытекающее из возможности отложенных действий (lazy evaluation), органически присущей функциональному программированию благодаря унификации представления данных и программ. Анализируются резервы производительности обобщенных процессов и методы динамической оптимизации вычислений, приводящие к смешанным и параллельным вычислениям.

Замедленные вычисления

Средства управления процессами в функциональном программировании изначально опираются на интуитивное представление о вычислении выражений, согласно которому функция применяется к заранее вычисленным аргументам.

Ради полноты пространства вычислений, гибкости программ и результативности процессов такое представление пришлось расширить и ввести категорию специальных функций, которые "знают", когда и что из их аргументов следует вычислить. Специальные функции могут анализировать и варьировать условия, при которых вычисление аргументов необходимо. Так используется возможность манипулировать данными, представляющими выражения, и явно определять в программах позиции обращения к интерпретатору. Эта возможность применялась для поддержки стандартной программотехники и традиционных форм конструирования функциональных объектов.

Свойственная функциональному программированию тенденция к полномасштабному применению всех попадающих в поле зрения средств логически требует перехода от частных случаев к поддержке универсального механизма, т.е. от набора конкретных специальных функций к более общему аппарату управления процессами вычислений.

Результат управления проявляется в изменении некоторых оценок, например можно влиять на эффективность и надежность программ, обусловленную целостностью объемных, сложных данных, избыточностью вычислений, возможно, бесполезных выражений, необоснованной синхронизацией формально упорядоченных действий. Подобные источники неэффективности могут быть устранены достаточно простыми методами организации частичных вычислений с учетом дополнительных условий для их фактического выполнения, таких как достижимость или востребованность результата вычислений.

Любое очень объемное, сложное данное можно вычислять "по частям".
Вместо вычисления списка

(x1 x2 x3 ...)

можно вычислить x1 и построить структуру:

(x1 (рецепт вычисления остальных элементов))

Получается принципиальная экономия памяти ценой незначительного перерасхода времени на вспомогательное построение. Рецепт - это ссылка на уже существующую

программу, связанную с контекстом ее исполнения, т.е. с состоянием ассоциативного списка в момент построения рецепта.

Пример 12.1. Построение ряда целых от M до N с последующим их суммированием. Обычная формула:

```
(defun ряд_цел (M N) (cond ((> M N) Nil)
                           (T(cons M (ряд_цел (1+ M) N)))))

(defun сумма (X) (cond ((= X 0) 0)
                       (T (+ (car X)( сумма (cdr X))))) )
```

Введем специальные операции || - приостановка вычислений и @ - возобновление ранее отложенных вычислений. Избежать целостного представления ряда целых можно, изменив
формулу:

```
(defun ряд_цел (M N) (cond ((> M N) Nil)
                           (T(cons M ( || (ряд_цел (1+ M) N))))))

(defun сумма (X) (cond ((= X 0) 0)
                       (T (+ (car X)( @ ( сумма (cdr X)))) ))
```

Чтобы исключить повторное вычисление совпадающих рецептов, в его внутреннее представление вводится флаг, имеющий значение T - истина для уже выполненых рецептов, F - ложь для невыполненных.

Тогда в выражении (all (cons { 1 | 2 } || (цел 3 100)) второй аргумент cons выполнится только для одного варианта, а для второго подставится готовый результат. Таким образом, рецепт имеет вид:

```
{ ( F e AL )
| ( T X ) },
```

где X = (eval e AL).

Это заодно позволяет распространить понятие данных на бесконечные, рекурсивно-вычислимые множества. Например, можно работать с рядом целых, больших чем N.

```
(defun цел (M) (cons M ( || (цел (1+ M) ))))
```

Можно из организованного таким образом списка выбирать нужное количество элементов, например первые K элементов можно получить по формуле:

```
(defun первые (K Int) (cond ((= Int Nil) Nil)
                            ((= K 0) Nil)
                       (T (cons (car Int)( первые ( @ (cdr Int))) )) ))
```

Эффект таких приостанавливаемых и возобновляемых вычислений получается путем следующей реализации операций || и @:

||e = > (lambda () e)
@e = > (e),

что при интерпретации приводит к связыванию функционального аргумента с ассоциативным списком для операции || и к вызову функции EVAL для операции @.

Обычно в языках программирования различают вызовы по значению, по имени и по ссылке. Техника приостановки и возобновления функций может быть названа вызовом по необходимости.

В некоторых языках программирования, таких как язык SAIL и Hope - lazy evaluation основная модель вычислений.

Наиболее частый вариант - приостановка аргументов всех определенных пользователем функций и операции CONS. В таком случае порождаются многократные приостановки, что требует итеративного возобновления до непосредственно исполняемого рецепта.

Более подробно о тонкостях определения ленивых вычислений рассказано в книге Хендерсона [3].

Смешанные вычисления

Идея смешанных вычислений с точки зрения реализации близка технике ленивых вычислений, но сложилась концептуально из несколько иных предпосылок [АПЕ]. Рассматривается пара Программа-Данные при недостаточных данных, отображаемая в так называемую остаточную программу, которая может дать нужный результат, когда даны недостающие данные. Для определения такого отображения понадобилась разметка действий программы на исполнимые и задерживаемые. Если такую разметку не связывать с отсутствием данных, то получается модель, практически подобная вычислениям с задержками и возобновлением.

Не всегда неопределенность части данных мешает организовать вычисление. Рассмотрим

(If (< X Y) Z T)

или эквивалент

if X < Y then Z else T

Если X и Y не определены, но известно, что X лежит в интервале [1, 4], а Y в интервале [5, 6],
то логическое выражение X<Y определено, и можно сделать вывод относительно выбора ветви условного выражения и, возможно, получить его значение.

Изучение смешанных вычислений может исходить из разных толкований понятия частичности, т.е. функций, определенных не на всей области их существования.

Первые работы Lombardi в этой области посвящены частичным вычислениям, т.е. обработке частично определенных выражений над числами. Реализация такой обработки на Лиспе осуществляла выполнимые операции и строила из полученных частичных результатов и невыполнимых операций некоторый промежуточный результат - выражение, доопределив которое, можно получить полный результат.

В.Э.Иткин оценивал **частичность как практичный критерий эффективности** организации деятельности.

При подготовке программ на Лиспе неопределенность часто представляют пустым списком, предполагая, что в него просто не успели что-то записать. Такое представление не всегда достаточно корректно и может потребовать дополнительных соглашений при обработке данных, по смыслу допускающих NIL в качестве определенного значения. Так, при реализации Lisp 1.5 введено соглашение, что значение атома в списке свойств хранится упакованым в список [1].

В работах по семантике стандартных языков программирования принято сведение к неопределенности значений любых операций, зависящих от неопределенных данных.

Это приводит на практике к необоснованным потерям части определенной информации и результатов.

A_1+...+A_100_000_000_000 + неопределенность -> неопределенность

Можно обратить внимание, что невелика практическая разница в уровне определенности данных вида (A **…**) и (A F), где F - рецепт вычисления, про который не всегда известно, приведет ли он к получению результата. Поэтому лучше было бы неопределенные данные "накрывать" рецептами, использующими специальные функции, нацеленные на раскрытие неопределенностей. Например, роль такой функции может сыграть запрос у пользователя дополнительной информации:

(A **…**) => (A **.** ||(read))

В определении интерпретатора обработка неопределенностей сосредоточена в функции ERROR.

```
(defun eval (e AL)
 …
    ((assoc e AL)(cdr (assoc e AL)))
    (T(ERROR '"неопределенная переменная"))
…
)
```

В определение функции ERROR можно включить обращение к READ, обрамленное сообщением о ситуации с информацией о контексте.

```
(defun apply (f  args  AL)
 …
     ((assoc f AL)(apply (cdr (assoc f AL))(evlis args AL)AL))
     (T (ERROR ‘"неопределенная функция"))
…
)
```

При отладке сложных комплексов часто неразработанные определения замещают временными "заглушками", которые помогают разобраться в будущей программе по частям. Такую работу можно стандартизировать заданием предварительных определений функций в виде отображения типа аргументов в тип результата. Соответственно, исполнение предопределенной таким образом функции можно интерпретировать как проверку аргументов на соответствие типу аргументов и выдачу в качестве результата вариантов значения, принадлежащего типу результата.

При небольшом числе значений заданного типа, например истинностные значения, может быть целесообразным полный перебор таких значений с последующим выбором реальной альтернативы пользователем.

```
(cond (e  r)(T g)) => (assoc e (list (cons T (eval r AL))
                                     (cons Nil (eval g AL))) )
```

Таким образом выполнятся обе ветви, их результаты ассоциируются с различными значениями заданого типа, что позволяет получить нужный результат, как только доопределится ранее не определенное значение. Это позволяет избежать повторного выполнения предшествующих вычислений, если их объем достаточно велик.

Применение библиотечных процедур, зависящих от слишком большого числа параметров, можно упростить для пользователя построением проекций на типовые комплекты трудно задаваемых параметров, понимаемых как определение режима работы процедуры.

```
(defun f  (x y z a b c … v t u) (g …))
(defun Fi (x y z ) (f x y z ai bi ci … vi ti ui))
```
Примерно это и делает необязательный параметр вида &optional.

Такое построение можно рассматривать как декомпозицию, разделение, сортировку на выполнимые и невыполнимые действия, при которой выполнимые действия в тексте определения замещаются их результатом, а невыполнимые преобразуются в остаточные, что все вместе образует проекцию процедуры на заданную часть ее параметров.

Многие выражения по смыслу используемых в них операций иногда определены при частичной определенности их операндов, что часто используется при оптимизаци кода программ.

Пример 12.2.

X * 0 = 0
car (A …) = A

X*1 = X при любом X
X-X = 0
X/X = 1 и т.п.

Представление функции в некоторых точках при отладке можно задать ассоциативной таблицей:

(setq f '((a1 . r1)(a2 . r2)(a3 . r3) **…**))
(defun f (x) (assoc x f))

В такое точечное определение легко добавлять недостающие пары, соответствующие нужным демонстрационным тестам при макетировании программ для согласования их функций на начальных этапах разработки, о чем еще будет речь в лекции 14.

Итак, мы получили некоторое число схем, различных с точки зрения управления вычислениями, полезных в разных ситуациях:

- частичные
- интервальные
- многовариантные
- предопределения
- точечные
- проекции

Возможны и другие, обеспечивающие оптимизацию, компиляцию, предвычисления, макрогенерацию текста программы, что в перспективе может покрыть полное пространство обработки программ в рамках единой методики. Например, основой единого подхода может быть так называемый трансформационный подход, заключающийся в сведении смешанных вычислений к преобразованию программ посредством набора базовых трансформаций.

Асинхронные процессы и параллелизм

Полное представление об асинхронных процессах, их эффективности и проблемах организации дают работы по сетям Петри .

Заметное место среди языков функционального программирования занимают языки параллельного программирования. Рассмотрим один из довольно известных - язык функционального программирования SISAL [11].

Название языка расшифровывется как "Streams and Iterations in a Single Assignment Language", сам он представляет собой дальнейшее развития языка VAL, известного в середине 70-х годов. Среди целей разработки языка SISAL следует отметить наиболее характерные, связанные с функциональным стилем программирования.

- Создание универсального функционального языка.
- Разработка техники оптимизации для параллельных программ.
- Достижение эффективности исполнения, сравнимой с языками типа Fortran и C.
- Внедрение функционального стиля в процессы разработки больших программ.

Эти цели содателей языка SISAL подтверждают, что функциональные языки способствуют разработке корректных параллельных программ. Одна из причин заключается в том, что функциональные программы свободны от побочних эффектов и ошибок, зависящих от реального времени. Это существенно снижает сложность отладки. Результаты переносимы на разные архитектуры, операционные системы или инструментальное окружение. В отличие от императивных языков, функциональные языки уменьшают нагрузку на кодирование, в них проще анализировать информационные потоки и схемы управления.

Легко создать функциональную программу, которая является безусловно параллельной, если ее можно писать, освободившись от большинства сложностей параллельного программирования , связанных с выражением частичных отношений порядка между отдельными операциями уровня аппаратуры. Пользователь Sisal-а получает возможность сконцентрироваться на конструировании алгоритмов и разработке программ в терминах крупноблочных и регулярно организованных построений, опираясь на естественный параллелизм уровня постановки задачи.

Начнем с примера программы:

1. Вычисление числа пи.

```
For                              % инициирование цикла
   Approx := 1.0;
      Sign := 1.0;
   Denom := 1.0;
         i := 1

 while i <= Cycles do          % предусловие завершения цикла

   Sign := -Sign;                          %  однократные
   Denom := Denom + 2.0;                   %  присваивания
   Approx := Approx + Sign / Denom;        %  образуют
   i := i + 1                              %  тело цикла

   returns Approx * 4.0
% выбор и вычисление результата цикла
   end for
```

2. Это выражение также вычисляет число пи.

```
   for i in [1..Cycles/2] do
% пространство параллельно исполнимых итераций

      val := 1.0/real(4*i-3) - 1.0/real(4*i-1);
% тело цикла, для каждого i исполняемое независимо

   returns sum( val )
% выбор и свертка результатов всех итераций цикла
```

```
    end for * 4.0
% вычисление результата выражения
```

Это выражение вычисляет сумму всех вычисленных значений val и умножает результат на 4.0.

3, 4. В for-выражениях операции dot и cross могут порождать пары индексов при формировании пространства итерирования:

```
for i in [1..2] dot j in [3..4] do
% для пар индексов [1,3] и [2,4]

    returns  product (i+j)
% произведение сумм

    end for                              %  = 24


for i in [1..2] cross  j in [3..4] do
% для пар [1,3],  [1,4],  [2,3] и [2,4]

    returns product (i+j)
% произведение сумм

    end for                               %  = 600
```

5. Итеративное for-выражение с обменом данными между итерациями:

```
    for
      I := 1
    while I < S do
      K := I;
      I := old I + 2;
% значение из предыдущей итерации

      J := K + I;
    returns product(I+J)
    end for
```

Как это свойственно языкам фукнционального программирования, и язык математически правильный - функции отображают аргументы в результаты без побочных эффектов, и программа строится как выражение, вырабатывающее значение. Наиболее интересна форма параллельного цикла. Она включает в себя три части: генератор пространства итераций, тело цикла и формирователь возвращаемых значений.

SISAL-программа представляет собой набор функций, допускающих частичное применение, т.е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных

аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений.

```
function Sum (N); % Сумма квадратов
  result (+ ( sqw (1 .. N)));
```

Обычно рассматривают оптимизации, обеспечивающие устранение неиспользуемого кода, чистку циклов, слияние общих подвыражений, перенос участков повторяемости для обеспечения однородности распараллеливаемых ветвей, раскрутку или разбиение цикла, втягивание константных вычислений, уменьшение силы операций, удаление копий агрегатных конструкций и др.

Лекция 13. Функции высших порядков

Рассматривается аппарат функций высших порядков при организации высококвалифицированных процессов информационной обработки, использующей формализацию и спецификацию данных, таких как синтаксический анализ, кодогенерация, конструирование интерпретаторов и компиляторов по формальному определению реализуемого языка – так называемые синтаксически управлямые методы информационной обработки .

Ранжирование функций

Применение функций высших порядков естественным образом завершает освоение функционального программирования как логической системы, допускающей конструирование функциональных объектов при решении задач регулярной обработки формализованной информации. Подобные задачи возникают при реализации и настройке сложных информационных систем, таких как операционные системы, системы программирования, текстовые и графические процессоры, системы управления базами данных, поддержки проектов и т.п.

Функции высших порядков используют другие функции в качестве аргументов или вырабатывают в качестве результатов.

```
(defun mul-N (N) #'(lambda (x) (* x N)))
; конструктор семейства функций, множащих аргумент на N

 (funcall (mul-N 25) 7)
; применение частной функции, умножающей на 25
```

Правильность выражений с такими функциями требует корректной подстановки параметров и учета **ранга функции**, определяющего возможность манипулирования функциональными значениями. Функции можно ранжировать на основе так называемых **типовых выражений**, представляющих области определения и значения функций. Например,

```
x+1 : Number -> Number

x+y : (Number  Number) -> Number
```

Отсутствие таких средств в языке можно в процессе программирования компенсировать соответствующими комментариями [3].

Суперпозицию функций можно характеризовать следующими типовыми выражениями:

```
S(h,g) = { при h: X -> Y, g: Y -> Z строит f=g(h) - суперпозиция }
                    : (X->Y  Y->Z) -> (X->Z)

(defun super (f g) #'(lambda (x) (funcall f(funcall g x)) ))
; конструктор суперпозиции функций
```

```
(funcall (super #'car #'cdr) '(1 2 3))
; применение суперпозиции  CAR и CDR
```

Двойное применение функции можно определить независимо или через суперпозицию – типовое выражение от этого не зависит, но оно представляет собой параметризованное выражение.

```
W f = ((lambda x)(f (f x))) = S (f,f) { дважды применяется функция }
   : (Number->Number) -> (Number->Number)
```
или более точно:
```
   : (X->X) -> (X->X),
```
где X - произвольный тип значения.

Типовое выражение представляет зависимость от этого типа - **параметризованный тип значения**.

```
 (defun duble (f) #'(lambda (x) (funcall f(funcall f x)) ))
; конструктор двойного применения функции

(funcall (duble #'car) '(((1) 2) 3))                    ;= (1)

(defun duble (f) (funcall #'super f f))
; двойное применение функции через суперпозицию

(funcall (duble #'car) '(((A B) B) C))          ; = (A B)
```

Можно ввести обозначения:

```
  Atom - атомы,
  Number - число,
  List (X) - NIL или списки из элементов типа X,
  Bool - NIL или T,`
  Some - любой объект.
```

Соответственно пишутся типовые выражения для элементарных функций:

```
cons : (X List (X)) -> List (X)
car  : List (X) -> X
cdr  : List (X) -> List (X)
eq   : (Atom Atom) -> Bool
at   : Some -> Bool
     : (Atom -> T) & (List (X) -> NIL)
nl   : Some -> Bool
     : (NIL -> T) & (Atom \=NIL -> NIL) & (List (X)\=NIL -> NIL)
```

Таким же образом можно специфицировать и универсальную функцию:

```
eval [e, al] : (Some List( (Atom . Some ) )) -> Some
```

```
        |  |
        | List( (Atom .  Some) )
Some {могут попасть и неправильные выражения }

apply [fn, (a1 a2 …), al] : (List(Some ) -> Some
                                List(Some )
                                List((Atom . Some))  ) -> Some
          |      |             |
          |      |           List((Atom . Some))
          |     List(Some )
(List(Some ) -> Some
```

Отображающий функционал также может характеризоваться типовым выражением:

```
map [x, f] : ( List(X)  (X->Y) ) -> List(Y)

(defun map- (x f) (cond (x (cons (funcall f (car x))
                           (map- (cdr x) f )))))
(map- '((1) (2) (3)) #'car )
```

Можно построить функцию, непосредственно преобразующую свой функциональный аргумент в новую функцию.

```
mapf [f] : List(X->Y) ->( List(X) -> List(Y))

(defun mapf (f)  #'(lambda (x)
    (cond (x (cons (funcall f (car x))
                (funcall (mapf f ) (cdr x)) )))  ))
(funcall (mapf #'car ) '((1) (2) (3)) )
```

Аргумент может быть списком функций, результаты которых следует собрать в общий список.

```
manyfun [lf] : List(X->Y) -> (X -> List(Y))
                  |                  |       |__список результатов функций
                  |                  |__тип аргумента отдельной функции
                  |_________список функций

 (defun manyfun (lf)  #'(lambda (x)
                              (cond (lf (cons (funcall (car lf) x)
                                              (funcall (manyfun (cdr lf)) x) )))  ))
 (funcall (manyfun '(car cdr length)) '(1 f (2 T) (3 D e)) )
```

Таким образом можно как бы «просачивать» определения функций над простыми данными по структурам данных и тем самым распространять простые функции на сложные данные подобно матричной арифметике. Такой стиль работы характерен для теории комбинаторов и языка FORTH. Похожие построения предлагаются Бэкусом в его программной статье о функциональном стиле программирования и в языке APL, ориентированном на обработку матриц.

Существует ряд языков функционального программирования, требующих или допускающих спецификацию объектов, что, кроме дисциплины программирования, дает средства для корректной работы с пакетами, сопряжения с модулями на других языках, оптимизирующих преобразований, распараллеливания и верификации программ (Sisal, ML и др.).

Конструирование распознавателей

Результативность функций высших порядков Хендерсон показывает на модельной задаче построения **распознавателя контекстно-свободного языка** [3]. В качестве примера такого языка рассмотрен синтаксис понятия "слог", образованный по правилам из гласных и согласных звуков, что можно представить **грамматикой** вида:

<а-гр> ::= A | A <а-гр>

<в-гр> ::= B | B <в-гр>

<слог> ::= <а-гр> <в-гр>
| <в-гр> <а-гр>
| <в-гр> <а-гр> <в-гр>

В этой грамматике "A" и "B" - терминальные символы, "слог", "а-гр" и "в-гр" - нетерминальные символы (метапонятия), "слог" – основное понятие. Необходимо быстро построить предикат is-syllable, выделяющий списки, представляющие правильно построенные слоги в соответствии с приведенными правилами.

Такое построение можно выполнить с помощью ряда функций высокого порядка, конструирующих распознаватели для альтернатив и цепочек из понятий, к которым сводится определение грамматики языка. Предполагается, что каждому правилу будет соответствовать свой распознающий предикат. Для простоты ограничимся случаями из пар альтернатив и двухзвенных цепочек.

Пусть тексты этого языка представляются списками из однобуквенных атомов A и B. Допустим, имеются предикаты is-A и is-B, выделяющие одноэлементные списки (A) и (B), соответственно.

```
(defun is-a (x)(cond ((eq(car x) 'a)
                      (null (cdr x))) ))   ; распознаватель A

(defun is-b (x)(cond ((eq(car x) 'b)
                      (null (cdr x))) )) ; распознаватель B
```

Типовые ранги этих функций одинаковы: List (X) -> Bool. Таким же должен быть и ранг результирующей функции is-syllable. При ее построении будет применена вспомогательная функция более высокого порядка is-alt, которая из произвольных предикатов конструирует новый предикат, перебирающий варианты правил и выдающий Nil, если ни одно из них не подходит. Функция is-alt может быть определена следующим образом:

```
(defun is-alt (p q) #'(lambda (x) (cond ((funcall p x )T)
; конструктор распознавателя альтернатив
                                        ((funcall q x) T)
                                        (T Nil))))
```

Ее типовый ранг имеет вид:

(List(X)->Bool List(X)->Bool) -> List(X)->Bool

Можно использовать эквивалент:

```
 (defun is-alt (p q) (lambda (x) (if (funcall p x) T (funcall q x)) ))
```

Предикат both, работающий как логическая связка “и”, можно реализовать как обычную функцию с типовым рангом (Bool Bool) -> Bool.

```
(defun both (x y) (cond ( x y)(T Nil)) )
; проверка одновременности условий
```

Еще одна вспомогательная функция высокого порядка is-chain из произвольных предикатов конструирует новый предикат, выясняющий, не выделяют ли исходные предикаты смежные звенья цепочки. Типовый ранг этой функции должен быть таким же, как у is-alt, т.к. их результаты используются при разборе и анализе текста в одинаковых позициях.

```
(defun is-chain (p q) #'(lambda (x )
 ; конструктор распознавателя  цепочек
            (cond ((null x) (both (funcall p x) (funcall q nil)) )
; пустая цепочка
                  ((both (funcall p x) (funcall q nil)) T)
; префикс без суффикса
                ((both (funcall p Nil) (funcall q x)) T)
; суффикс без префикса
                   ((both (funcall p (cons (car x)Nil)) (funcall q (cdr x)) ) T)
; допустимое разбиение
       (T(funcall (is-chain (lambda(y)(funcall p(cons(car x)y)))
                                                q ) (cdr x) ))     )))
; сдвиг границы разбиения вправо
```

Из данного распознавателя is-a можно бы и без функций высших порядков построить распознаватель is-a-gr, распознающий группу из любого числа символов A:

```
 (defun is-a-gr (x ) (if x
; распознаватель цепочек из A
                         (cond ((eq (car x) 'a)  (is-a-tl (cdr x)) )
; <a-гр>    ::= A | A <a-гр>
                         (t  nil) ) Nil))
```

```
(defun is-a-tl (x)(cond ((null x)T)((eq (car x)'A)(is-a-tl (cdr x )) ))))
; хвост цепочки из A
```

Но использование конструкторов is-alt и is-chain, показанное на примере распознавателя is-b-gr, позволяет построить определение, синтаксически подобное правилу грамматики:

```
(defun is-b-gr (x )  (funcall (is-alt  #'is-b (is-chain #'is-b #'is-b-gr)) x ))
; распознаватель цепочек из B
;          <в-гр>    ::=                    B |              B      <в-гр>
```

Теперь опробованные приемы конструирования распознавателей применяем к построению функции is-syllable, активно опираясь на чисто внешнее, синтаксическое подобие определению заданной грамматики:

```
(defun is-syllable (x )
; распознаватель слога
        (funcall (is-alt  (is-chain  #'is-b-gr  #'is-a-gr)
;   BA
                          (is-alt  (is-chain #'is-a-gr #'is-b-gr)
;  AB
                                   (is-chain #'is-b-gr (is-chain #'is-a-gr #'is-b-gr))
; BAB
                          )          ) x ))

(is-syllable '(a  b))
(is-syllable '(b a))
(is-syllable '(b a b ))
(is-syllable '(b b b b a a b b ))
```

Сопоставляя правила и полученное определение распознавателя, можно убедиться, что собственно конструирование распознавателя осуществляется и модернизируется сравнительно быстро: достаточно свести распознаваемый язык композиции альтернатив и цепочек.

```
<слог>    ::=  <в-гр> <а-гр>
               | <а-гр> <в-гр>
               | <в-гр> <а-гр> <в-гр>
```

```
(defun is-syllable (x )
; распознаватель слога
; <слог>    ::=
          (funcall (is-alt  (is-chain  #'is-b-gr  #'is-a-gr)
;   BA
;                                        <в-гр>    <а-гр>
                                (is-alt  (is-chain #'is-a-gr #'is-b-gr)
;  AB
;                                       |                <а-гр> <b-гр>
                                         (is-chain #'is-b-gr (is-chain #'is-a-gr #'is-b-gr))
; BAB
```

; | <в-гр> <а-гр> <в-гр>

)) x))

Результат сопоставления показывает, что достигнуто синтаксическое подобие определения грамматики и построенного распознавателя. Это значит, что определение можно автоматически отобразить в такой распознаватель. Отображение – функция высокого порядка, вырабатывающая в качестве результата распознаватель языка, порождаемого исходной грамматикой.

Таблица: 13.1 Определение распознавателя языка, синтаксически подобное грамматики языка.

Распознаватель	(defun is-alt (p q) #'(lambda (x) (cond ((funcall p x)T) ((funcall q x) T) (T Nil)))) (defun is-chain (p q) #'(lambda (x) (cond ((null x) nil) ((both(funcall p x) (funcall q nil)) T) ((both(funcall p Nil) (funcall q x)) T) ((both(funcall p (cons (car x)Nil)) (funcall q (cdr x))) T) (T(funcall (is-chain (lambda(y) (funcall p(cons(car x)y))) (lambda(y)(funcall q y))) (cdr x)))))) (defun is-syllable (x) (funcall (is-alt (is-chain #'is-**b-gr** #'is-**a-gr**) (is-alt (is-chain #'is-**a-gr** #'is-**b-gr**) (is-chain #'is-**b-gr** (is-chain #'is-**a-gr** #'is-**b-gr**)))) x))
Грамматика	<слог> ::= <**в-гр**> <**а-гр**> | <**а-гр**> <**в-гр**> | <**в-гр**> <**а-гр**> <**в-гр**>

Преобразование определений

Конечно, построенное выше определение не отличается эффективностью. Обычно синтаксические формулы приводят к **нормализованной форме**, гарантирующей полезные свойства распознавателей и удобство их построения. Выбор нормализованной формы и процесс нормализации обосновывается доказательными построениями, на практике воспринимаемыми как эквивалентные преобразования. **Преобразования формул** – еще один интересный класс задач символьной обработки. Для демонстрации рассмотрим модель реализации функций **свертки** текстов. При подходящем выборе обозначений такие функции можно применять для преобразования синтаксических формул с целью приведения к нормализованной форме.

Пусть свертки системы текстов представлены в стиле самоописания подобно формам Бекуса-Наура списком вида:

```
(
(Тексты (Имя Вариант ...)...)
 ; первое имя - обозначение системы текстов
; за ним следуют варианты поименованных текстов

(Вариант Элемент ...)
; Вариант представляет собой последовательность Элементов

(Элемент Имя Лексема (Варианты))
; Элемент – это или Имя, или Лексема, или Варианты в скобках
)
```

Для системы текстов “((м а ш и н а)(м а ш а)(ш и н а))” можно дать свертку вида:

```
(
(пример (ма ((ш н)
                (ш а))
          ( ш н   )      )
 (н ина)
)
```

Построение свертки системы текстов выполняется функциями unic, ass-all, swin, gram, bnf :

```
(defun unic (vac) (remove-duplicates (mapcar 'car vac) ))
;; список уникальных начал

(defun ass-all (Key Vac)
;; список всех вариантов продолжения,
;; что может идти за ключом
          (cond
             ((Null Vac) Nil)
             ((eq (caar Vac) Key) (cons (cdar Vac)
                                  (ass-all Key (cdr Vac)) ))
             (T (ass-all Key (cdr Vac)) )
)          )

(defun swin (key varl) (cond
;; очередной шаг свертки
;; снять скобки при отсутствии вариантов
                ((null (cdr varl))(cons key (car varl)))
                (T (list key (gram varl)) )
))
(defun gram (ltext)
;; левая свертка, если нашлись общие начала
  ( (lambda (lt) (cond
                ((eq (length lt)(length ltext)) ltext)
                (T (mapcar
```

```
                    #'(lambda (k) (swin k (ass-all k ltext ) ))
                          lt  )
   )          )  ) (unic ltext)
) )
```

```
(defun bnf (main ltext binds) (cons (cons main (gram ltext)) binds))
```

В результате синтаксические формулы можно приводить к нормализованному виду, пригодному для конструирования **эффективного распознавателя** с грамматикой текста. Организованные таким образом свернутые формы текстов могут играть роль словарей, грамматик языка, макетов программ и других древообразных структур данных, приспособленных к обработке рекурсивными функциями.

Обратные преобразования представляют не меньший интерес. Их можно использовать как генераторы тестов для синтаксических анализаторов или перечисления маршрутов в графе и других задач, решение которых сводится к обходу деревьев.

Построение **развертки**, т.е. системы текстов по их свернутому представлению, выполняется функциями names, words, lexs, d-lex, d-names, h-all, all-t, pred, sb-nm, chain, level1, lang.

Функции names, words и lexs задают алфавит и разбивают его на терминальные и нетерминальные символы на основе анализа их позиций в определении.

```
(defun names (vac) (mapcar 'car vac))
;; определяемые символы

(defun words (vac) (cond
;; используемые символы
      ((null vac) NIL)
      ((atom vac) (cons vac NIL ))
      (T (union (words(car vac)) (words (cdr vac)))) ))

(defun lexs (vac) (set-difference (words vac) (names vac)))
;; неопределяемые лексемы
```

Функции d-lex и d-names формируют нечто вроде встроенной базы данных, хранящей определения символов для удобства дальнейшей работы.

```
(defun d-lex ( llex)
;; самоопределение терминалов
   (mapcar #'(lambda (x) (set x x) ) llex) )

(defun  d-names ( llex)
;; определение нетерминалов
   (mapcar #'(lambda (x) (set (car x )(cdr x )) ) llex) )
```

Функции h-all, all-t и pred раскрывают слияния общих фрагментов системы текстов.

```
(defun h-all (h lt)
;; подстановка голов
    (mapcar #'(lambda (a)
                (cond
                   ((atom h) (cons h a))
                   (T (append  h a))      )
             ) lt) )

(defun all-t (lt tl)
;; подстановка хвостов
    (mapcar #'(lambda (d)
       (cond
              ((atom d) (cons d tl))
            (T(append d tl))
       )     ) lt) )

(defun pred (bnf tl)
;; присоединение предшественников
  (level1 (mapcar #'(lambda (z) (chain z tl )) bnf) ))
```

Функции sb-nm, chain и LeveL1 строят развернутые, линейные тексты из частей, выполняя подстановку определений, сборку и выравнивание.

```
(defun sb-nm (elm tl)
;; подстановка определений имен
    (cond
      ((atom (eval elm)) (h-all (eval elm) tl))
      (T (chain (eval elm) tl))
)    )

(defun chain (chl tl)
;; сборка цепочек
  (cond
    ((null chl) tl)
    ((atom chl) (sb-nm chl tl))

    ((atom (car chl))
          (sb-nm (car chl) (chain (cdr chl) tl) ))

    (T (pred (all-t (car chl) (cdr chl)) tl)) ))

(defun level1 (ll)
;; выравниваие
  (cond
    ((null ll)NIL)
    (T (append (car ll) (level1 (cdr ll)) )) ))
```

На основе приведенных вспомогательных функций общая схема развертки языка по заданному его определению (свертке) может быть выполнена функцией lang:

```
(defun lang ( frm )
;; вывод заданной системы текстов
 (d-lex (lexs frm))
 (d-names frm)
 (pred (eval (caar frm)) '(())
)               )
```

Вот и тесты к этой задаче, предложенные И.Н. Скопиным, справедливо предположившим, что для решения задач синтаксически управляемой обработки текстов хорошо подходит функциональный стиль программирования на Лиспе:

```
 (lang (print (bnf 'vars
                   '((m a s h a)(m a s h i n a)(s h i n a))
                         '((n (i n a))) )))

 (lang '((vars (m a ((s h a)(s h n))) (s h n) ) (n (i n a)) ) )
```

Цель преобразования синтаксических формул при определении анализаторов и компиляторов можно проиллюстрировать на схеме рекурсивного определения понятия “Идентификатор”:

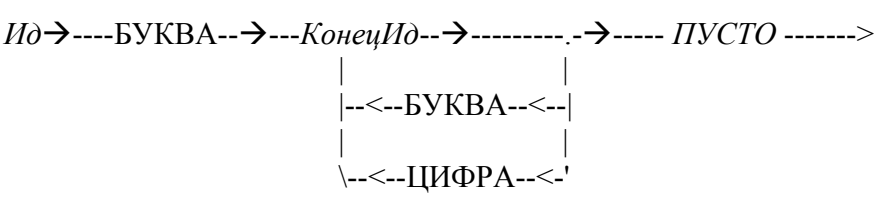
Ид ::= БУКВА
| *Ид* БУКВА
| *Ид* ЦИФРА

Удобное для эффективного синтаксического разбора определение имеет вид:

Ид ::= БУКВА | БУКВА *КонецИд*

КонецИд ::= БУКВА *КонецИд*
| ЦИФРА *КонецИд*
| *ПУСТО*

Синтаксическая диаграмма анализатора



Этот пример показывает, что удобные для анализа формулы приведены к виду, когда каждую альтернативу можно выбрать по одному текущему символу. Система CLOS поддерживает ООП с выделением методов для одноэлементных классов, распознаваемых простым сравнением. Тем самым обеспечено удобное построение программ над структурами, подобными нормализованным формам.

Например, определение:

<а-гр> ::= A | A <а-гр>

<в-гр> ::= B | B <в-гр>

<слог> ::= <а-гр> <в-гр>
| <в-гр> <а-гр>
| <в-гр> <а-гр> <в-гр>

можно привести к виду, не требующему возвратов при анализе (в фигурных скобках - внутренние альтернативы):

<а-гр> ::= A <а-кон>
<а-кон> ::= <пусто> | A <а-кон>

<в-гр> ::= B <в-кон>
<в-кон> ::= <пусто> | B <в-кон>

<слог> ::= A <а-кон> B <в-кон>
| B <в-кон> A <а-кон> <в-кон>

Если программирование сводит алгоритм решения задачи к программе из определенной последовательности шагов, то конструирование строит программу решения задачи из решений типовых вспомогательных задач. Для задачи реализации языка программирования ключевой (но не единственной) типовой задачей является определение реализуемого языка. Ее решение открывает возможности автоматизированного конструирования анализаторов и компиляторов. Автоматизацию конструирования системы программирования обеспечивают методы синтаксического управления обработкой информации и методы смешанных/частичных вычислений, позволяющие выводить определение компилятора программ из определения интерпретатора.

Все это хорошо изученные задачи, имеющие надежные решения, знания которых достаточно для создания своих языков программирования и проведения экспериментов с программами на своих языках. Существует ряд программных инструментов, поддерживающих автоматизацию процесса создания и реализации языков программирования и более общих информационных систем обработки формализованной информации, например YACC, LEX, Bison, Flex, основные идеи применения которых достаточно близки изложенным выше методам обработки формул и текстов.

Лекция 14. Макеты программ и тесты

Техника функционального программирования иллюстрируется примерами поддержки полного жизненного цикла программ с помощью быстрого прототипирования и спецификации программ. В этом плане существенна возможность введения частично определенных функций, варьируемых и уточняемых определений, а также специализация интерпретатора программ с целью учета уровня достоверности определений. Рассматриваются примеры построения прототипов системы, опережающего детальную разработку алгоритмов и отладку программ. Основой является процесс уточнения информации о решаемой задаче, продемонстрированный на отдельных примерах и схемах с привлечением частичных функций на доступных типах данных с доведением до полных функций, приспособленных к обработке произвольных данных.

Построение теорий при разработке программ

Принимая аксиоматическую теорию множеств за образец грамотно разработанной теории, попробуем проанализировать доказательные положения, полезные при обосновании и выполнении программистских проектов.

Многие построения в теории множеств выполнены над **кумулятивной иерархией** множеств, инициированной некоторым множеством объектов не множественной природы и пустого множества посредством операции объединения множеств. Кроме того, над множествами определены операции пересечения, дополнения, равенства, вхождения и включения, удовлетворяющие небольшому набору аксиом разной сложности.

Аналогично, структуры, такие как S-выражения, выстроены над атомами, не структурируемыми на компоненты, и пустого списка NIL, посредством операции CONS – консолидации. Над S-выражениями определены операции, позволяющие разбирать структуры на компоненты, сравнивать и анализировать структуры, отличать атомы от структур и пустой список от других данных. Элементарные операции подчинены аксиомам, обеспечивающим обратимость информационной обработки, и техника программирования на уровне строгих функций поддерживает прозрачность определений и скорость отладки.

Рассматривая программы и программные системы как формы представления знаний, трудно удержаться от попытки исследования **динамики представления знаний** на основе аналогии с развитием программ и программных систем.

Движущими силами этого развития являются: необходимость разных видов **эффективной** деятельности, потребность в **уточнении** представления знаний и установление новой информации, которая раньше не попадала в поле зрения или наблюдатель не был готов ее понять. Динамика представления знаний сводится к переходу от одного представления к другому.

Успешность эффективной деятельности ограничена "пропускной способностью" поля зрения. Это ограничение систематически преодолевается посредством обобщения, приводящего к представлениям более высокого порядка - представлениям более мощным, более организованным, например к процедурам, функциям, фреймам, шаблонам, макросам. Последовательность шагов обобщения можно называть **индуктивным развитием** представления знаний. В методике программирования индуктивное развитие соответствует восходящим методам, "снизу вверх". Как правило, индуктивное развитие имеет некоторые пределы. Такие пределы при возрастании **меры информативности** используемых средств рассматриваются Д.Скоттом [5]. Интересен случай, когда пределом является теория, достаточная для порождения всей достоверной информации, установленной на данный момент времени. При разработке программ роль такого предела играет система программирования.

В результате индуктивного развития представления знаний наблюдается тенденция к возрастанию доли средств **декларативного** характера (таких как описания, отношения, формирователи, типы, фреймы, семантические сети, иерархии понятий, аксиоматические системы) в сравнении с долей средств **процедурного** характера (таких как действия, операции, операторы, процедуры, интерпретаторы, задания). Эта тенденция обуславливает рост эффективности применения дедуктивных методов и может рассматриваться как стимул к переходу от индуктивного **развития** к **дедуктивному**. Дедуктивный вывод осуществляет переход от потенциальных знаний к актуальным. Традиционно для этих целей в системах искусственного интеллекта используется метод

резолюций, системы продукций и другие средства. **Чередование стадий** индуктивного и дедуктивного развития можно рассматривать как обоснование выбора метода программирования в зависимости от уровня развития знаний о решаемой задаче (зрелость, **уровень изученности**).

Применение развиваемых таким образом представлений может потребовать возврата к менее структурированным средствам (например, для упрощения обратной связи с областью, породившей решаемые задачи или для более тонкой детализации реализационных решений). Такой переход является **конкретизацией** представления знаний. В методике программирования конкретизация соответствует нисходящим методам "сверху вниз".

Независимо осуществляемое развитие приводит к задаче установления **эквивалентности** между различными системами представления знаний. При решении этой задачи возникают предпосылки для целенаправленного дедуктивного развития, что приводит к выравниванию потенциала систем (вводятся недостающие понятия, выполняются аналогичные построения, реализуются подобные инструменты). Таким образом, выделено четыре типа переходов: индуктивное и дедуктивное развитие, конкретизация и выравнивание. Эта классификация сопоставима с классификацией трансформаций программ в теории смешанных вычислений, предложенной А.П.Ершовым [9].

Макетирование функций

При разработке больших программ, особенно по нисходящей методике, необходимость в тестировании и отладке возникает намного раньше, чем подготовлен текст программы. **Макет** программы – это некоторый предварительный ее текст, допускающий уточнение – **доопределение.**

Простейший макет может быть создан из небольшой коллекции **тестов**, иллюстрирующих поведение программы в наиболее важных точках. Выбор таких точек – необходимая работа, результаты которой многократно используются на всех фазах жизненного цикла программы: при конструировании алгоритмов, автономном тестировании компонентов программы, комплексной отладке программы, демонстрации

программы всем заинтересованным лицам, при ее эксплуатации и развитии. Функционирование простых макетов особенно легко реализуется в языках, обладающих унификацией структур данных и функциональных объектов, таких как Лисп и Сетл.

Сетл – язык сверхвысокого уровня, представляет собой попытку практичного использования теоретико-множественных понятий в практике программирования [12]. Согласно концепции этого языка, понятие “функция” обладает двойственной природой. Функция может быть представлена в алгоритмическом стиле – определением процедуры, выполнение которой сопоставляет результат допустимому аргументу. Но столь же правомерно представление функции в виде **графика**, отображающего аргументы в результаты. Оба представления могут существовать одновременно – это всего лишь две реализации одной функции. Графическое понимание функции включает в себя и **табличную реализацию** подобно математическим таблицам Брадиса. Кроме того **график функции** не обязан быть линией – это может быть фигура произвольных очертаний. Следовательно, аргументу может соответствовать множество результатов, лежащих на пересечении вертикали с этой фигурой – графиком функции. При такой трактовке нет ничего удивительного в постепенном накоплении или построении графика функции. Можно задать небольшое множество точек графика, а потом постепенно его пополнять. По замыслу Дж.Шварца, автора языка SETL, такая методика может выполнять роль оптимизации особо сложных вычислений.

Более формальный макет может быть построен из **спецификаций** функций в виде типовых выражений, задающих описание типов аргументов и результатов. Такой макет может работать как “заглушка” для нереализованных компонентов. Вместо них может работать универсальная функция, проверяющая соответствие фактических аргументов предписанному типу данных и вырабатывающая в качестве результата произвольное данное, соответствующее описанию результата. Такой механизм будет более эффективен в паре с простым макетом из тестов, если результат выбирать из коллекции тестов.

Мемо-функции и тестирование

Не менее ценные следствия из унификации структурных значений и функциональных объектов дает накопительный, кумулятивный эффект ряда сеансов обработки

рекурсивных программ, содержащих общие компоненты. Допустимость совместного хранения функциональных определений и тестов для их проверки в общей структуре, например в списке свойств атома, именующего функцию, позволяет строить **технологические макеты** с множественными определениями, коллекциями тестов и спецификаций, а также с документацией. Такие макеты пригодны для поддержки **полного жизненного цикла программы**. Они позволяют организовывать оперативное сравнение результатов при обновлении системы функций. На такой основе возможно автоматическое тестирование программ. С практической точки зрения технологические макеты – универсальный инструмент динамической оптимизации прикладных систем.

Представим, что вычисление каждой рекурсивной функции сопровождается сохранением пары <аргумент, результат>. После этого можно запустить в дело слегка измененное правило интерпретации функций. Изменение заключается в следующем: прежде чем применять функцию к фактическому аргументу, выполняется проверка, нет ли для этого аргумента уже вычисленного результата. Готовый результат и есть результат функции, а в противном случае все работает как обычно. Механизм сохранения насчитанных результатов функций назван “**мемо-функции**” [4]. Естественно, основанием для его применения является достаточная сложность и частота обработки. Примечательная особенность данного метода – любая сложность очень частых вычислений стремится со временем к линейной **:)**

Лекция 15. Парадигмы программирования

Подводится итог изучению основ функциональное программирование и особенностей его применения. Анализируются наиболее очевидные закономерности применения языков программирования, отражающие расширение класса решаемых задач, прогресс элементной базы и рост квалификации программистов. Рассматриваются ключевые моменты развития парадигм программирования и анализируются закономерности в процессе реализационного освоения новых областей обработки информации. Приведен небольшой обзор парадигм программирования. Для желающих поэкспериментировать дана справка о реализационных особенностях GNU Clisp [6,7]

Итоги и выводы

Согласно рекомендациям специалистов по обучению информатике, функциональное программирование (ФП) входит в число основных подходов к обучению инфоматике в университетах (наряду с алгоритмическим, императивным, аппаратным, объектным и обзорно-ознакомительным). В целом средства и методы ФП образуют два слоя. Глубинный слой - **локальное программирование строгих функций**, безотходных структур данных, обратимых контекстов, регулярных отображений, корректных функций высших порядков, универсальных функций и средств управления вычислениями. Внешний слой - **функциональное моделирование** широкого спектра **парадигм программирования**, обеспечивающее другие сферы программирования прототипами, дающее подход к оценке функциональности информационных систем и их компонентов.

Глубинный слой дает **концептуальную основу** для применения и определения функций во всей полноте этого понятия, для его развития и выбора реализационных решений при разработке систем ФП, включая привлечение стандартной программотехники и деструктивных функций. Внешний слой открывает **перспективы повышения уровня используемых конструкций** на

базе моделирования основных механизмов системного, низкоуровневого, оптимизационного, логического, высокопроизводительного, ООП и других подходов к разработке программ. Каждый из подходов по своему расширяет понятие "функция", варьирует правила применения и реализации функций, конкретизирует расширения и специализацию систем ФП.

Фактически термин "функциональное программирование" используется при объединении в систему методов решения классов задач, обладающих исследовательскими аспектами, что влечет за собой **необходимость развития полученных решений**. Система предполагает **общую логику уточнения решаемых задач и формализацию обобщенных решений** на основе специально выбранных базовых конструкций.

1) **Базовые конструкции** определяются как строгие функции.

2) Общая логика развития задачи сводится к процессу раскрутки полного решения как набора **шагов по расширению набора функций и повышению их потенциала использованием отображений**, что обеспечивается надежными средствами языка функционального программирования (ЯФП) и быстро отлаживается на базе систем ФП (СФП), приспособленных к интерпретации программ.

3) При необходимости выполняются **формальные преобразования программ**, (например, компиляция), обеспечивающие улучшение эксплуатационных характеристик, связанных с процессами исполнения программ.

4) Важный критерий качества ФП - **полнота системы функций и универсальность определений**, дающая возможность синтаксически управляемой обработки данных с помощью ФВП, что существенно повышает надежность программирования.

5) Разработка ИС средствами функций высших порядков (ФВП) успешно выполняет **роль прототипа** для реализации другими, более распространенными средствами.

Оттолкнувшись от интуитивного представления о понятии «функция», мы для начала ограничились **однозначными функциями**, но разрешили предельно широкое толкование понятия “значение”, включающее понятие “структура данных”.

1) Ориентируясь на **рекурсивные определения функций**, мы ввели несложную схему, достаточно удобную для построения формул, задающих функциональные определения. При этом отмечено качественное различие между элементарными функциями, задаваемыми неформально вне метаязыка, и остальными функциями, определяемыми формулами языка, использующими обозначения элементарных функций. В качестве примера рассмотрен **элементарный Лисп**.

2) Затем было конкретизировано основное множество значений функций как множество списков и атомов, на котором определены алгебра и логика, достаточные для обработки и анализа таких значений. Представления функций отображены в это множество и определена **универсальная функция**, по списочному представлению функции и ее аргументов строящая результат.

3) Рассмотрены примеры структур данных, полезных при реализации списков, интерпретируемых как функции (стеки, пары, блоки, односвязные списки, расстановочные таблицы и др.).

4) Изучено расширение функционального языка, достаточное для императивно процедурного стиля программирования, что обеспечивает помимо нисходящей методики разработки программ и восходящую, более естественную для несложных задач.

5) Определена абстрактная машина, содержащая реализацию элементарных функция языка и поддерживающая интерпретацию функционального языка, и

проанализирован **компилятор** с абстрактного синтаксиса функционального языка программирования на языково-ориентированную абстрактную машину.

Таким образом, завершена **нисходящая линия определения** языка функционального программирования, позволяющая во всех деталях представлять один возможный процесс применения функций на уровне интуитивных понятий, структур данных и машинного кода. Затем была выполнена серия обобщений представления о процессах применения функций, т.е. осуществлена **восходящая линия определения** функционального языка.

Этот процесс не всегда удовлетворителен по эффективности с разных точек зрения. Повышение эффективности обычно требует развития размерности пространства, в котором рассматриваются оптимизируемые понятия.

6) Исследованы возможные направления развития как по вертикали, так и по горизонтали. Для этого изучены традиционные решения по организации структур данных и систем программирования для поддержки функционального программирования, а также списки свойств атомов и деструктивные функции.

7) Неоднозначные функции могут рассматриваться через понятие отношения. При изучении идей ООП показано, что идеи эти весьма близки и легко поддерживаются базовыми средствами функционального программирования. Достаточно лишь продемонстрировать технику работы с классами и экземплярами объектов и способы определения методов их обработки.

8) Рассмотрено понятие вариантов или альтернативных процессов, успешного выполнения одного из которых достаточно для построения результата функции. Реализация таких процессов потребовала уточнить определение абстрактной машины, чтобы обеспечить сохранение состояния памяти для выхода из тупиковых ситуаций. Такое же уточнение необходимо для обеспечения диагностичности.

9) Представлены методы управления процессами вычисления функций и средства, обеспечивающие выбор времени выполнения отдельных шагов процесса, соответствующих конкретным формулам. Организация таких

процессов обеспечивается совместным хранением данных и рецептов их получения, что заодно снимает требование конечности представления данных. Достаточно конечности рецепта. Техника работы с рецептами, заключающаяся в приостановке и возобновлении программ, не требует специальных реализационных механизмов.

10) **Функции высших порядков** показаны как инструмент естественной модуляризации программ, например, техникой продолжений, достаточной для достижения подобия представления функций и обрабатываемых ими типов данных. Иллюстрация такого подобия на задаче построения синтаксического анализатора позволяет замкнуть изученные механизмы на исходный язык программирования.

Практические аспекты

Требования к учебно-экспериментальной практике можно условно разделить на четыре группы, различающиеся по целям:

- элементарное **знакомство** с концепциями ФП,
- **эксперименты** по системному программированию на базе ФВП,
- **моделирование** изучаемых приложений средствами СФП,
- изучение средств СФП как практического **инструмента**.

Первая из этих целей - знакомство - может быть достигнута на уровне **концептуального минимума**, достаточно далекого от решения технических проблем. В круге общеизвестных задач, пригодных для показа изучаемых явлений, на уровне регулярной обработки небольших текстов.

Вторая цель – эксперимент – гораздо чувствительнее к **максимальному потенциалу** реализации СФП, к ее гибкости и переносимости. Самоприменимость языков функционального программирования здесь

гарантирует очевидные преимущества в сравнении со стандартными и производственными языками.

Третья цель – функциональное моделирование – обеспечивается как **практичный компромисс**, учитывающий разного рода обстоятельства: исторически сложившиеся стандарты, профессиональные стереотипы и жаргон, уровень квалификации заинтересованных лиц и многое другое, что порождает новые языки и их диалекты.

Четвертая цель – реальный инструмент – требует методично выбранного **приемлемого баланса** между уровнем изученности класса решаемых задач и уровнем организованности комплекта средств, имеющихся в СФП.

Элементарный Лисп, описанный как **Pure Lisp** Дж. Мак-Карти, идеально соответствует цели знакомства с ФП, его базовые средства доступны практически в любой реализации основных диалектов Лиспа. Навыки и понимание основ обработки структурированных данных на уровне элементарного Лиспа пригодятся при работе с любой СФП. Конструирование функций средствами чистого Лиспа доставляет интеллектуальное удовольствие, оно сродни решению математических головоломок. Благодаря функциональной полноте Лиспа, изучение других инструментов ФП, а также основных средств проектирования и программирования, можно обосновывать и понимать через программирование на Лиспе.

Во всей полноте идеи функционального программирования поддержаны в проекте **Lisp 1.5**, выполненном Дж. Мак-Карти и его коллегами [1]. В этом исключительно мощном языке не только реализованы основные средства, обеспечившие практичность и результативность функционального программирования, но и впервые опробован целый ряд поразительно точных построений, ценных как концептуально, так и методически и конструктивно, понимание и осмысление которых

слишком отстает от практики применения. Понятийно-функциональный потенциал языка Lisp 1.5 в значительной мере унаследован стандартом Common Lisp, но многие идеи пока не получили достойного развития. Вероятно, это дело будущего - для нового поколения системных программистов.

По мере накопления опыта реализации СФП на базе Лиспа и других языков сформированы обширные библиотеки функций, весьма эффективно поддерживающих обработку основных структур данных - списков, векторов, множеств, хэш-таблиц, а также строк, файлов, каталогов, гипертекстов, изображений. Существенно повысилась результативность системных решений в области работы с памятью, компиляцией, манипулирования пакетами функций и классами объектов. Все это доступно в современных ФСП, таких как GNU Clisp, Python, CMUCL и др., основная проблема при изучении которых – слишком много всего, разбегаются глаза, трудно выбрать первоочередное. Хочется найти пересечение со знакомыми программами и воспроизвести любимые приемы в новой стилистике – естественный путь для решения задач функционального моделирования.

С конца 70-х годов появились **Лисп-процессоры**, доказавшие, что пресловутая неэффективность функционального программирования обусловлена характеристиками оборудования, а не стилем программирования. Функциональные мини-языки хорошо показали себя и при решении задач аппаратного уровня. К середине 90-х годов появились весьма убедительные результаты [13] по динамической оптимизации процессов, и были осуществлены высокопроизводительные схемы работы с памятью на новом оборудовании. Все это превращает СФП в практичный и перспективный инструментарий.

Полученное при изучении ФП обобщенное представление о процессах выполнения функций нацеливает на решение задач в виде определения семейства процессов и выбор подходящего представителя семейства. Привлекая теоретико-множественный подход как обнадеживающую

аналогию, наполнение понятий при функциональном программировании и организации его системной поддержки осуществляется следующими преобразованиями:

- проекция в более удобное, простое, но не слишком бедное множество,
- ортогонализация независимых вспомогательных понятий,
- унификация родственных понятий,
- распространение реализуемых действий на новые области существования.

Посредством таких преобразований производится **уточнение структуры пространства**, в котором рассматривается каждое понятие. Кроме шагов расширения по горизонтали обычно происходит и реорганизация по вертикали – детализация и обобщение, при котором, как правило, возрастает число измерений исходного пространства, точнее, видимой его части.

Такая схема подтверждается самой историей развития диалектов языка Лисп и родственных ему языков программирования. (Pure Lisp, Lisp 1.5, Lisp 2, Interlisp, CommonLisp, MicroLisp, MuLisp, Sail, Hope, Miranda, Scheem, ML, GNU Clisp, CLOS, Emacs, Elisp, xLisp, Vlisp, AutoLisp, Haskell, Python, CMUCL). Не вдаваясь здесь в подробности этой истории (ее изложение заслуживает отдельного курса!), отметим лишь особенности свободно распространяемой системы GNU Clisp, которую легально можно использовать в качестве системы, поддерживающей ФП.

Стандарт **Common Lisp** в сравнении с Лиспом от Мак-Карти имеет ряд отличий, несколько влияющих на программотехнику. Прежде всего, это касается реализации списков свойств атомов. Данная структура реализована в императивном стиле, в виде таблицы с непрерывным «забыванием» информации после каждого присваивания. В результате исключено многократное связывание глобальных объектов с их определениями, а заодно и множественное объявление свойств атомов. Конечно, такие эффекты можно смоделировать, но это не столь гармонично. Другое отличие связано с механизмом контекстов

вычисления выражений. Стандарт при вычислении значений переменных по умолчанию привлекает статический контекст, иначе переменную надо объявить специальной. Третье отличие затрагивает унификацию представления функций и обрабатываемых ими значений. При внешнем подобии – и то, и другое выглядит как круглоскобочные списки, но реализационно это разные типы данных, и возникает нечто вроде приведения типа (**funcall**) в ситуациях вызова функций, конструируемых “на лету”. Имеются и другие, не столь явные отличия, которые пока не стоит упоминать. GNU Clisp, xLisp, CMUCL соответствуют стандарту Common Lisp. Документация по этим СФП доступна на сайтах любителей Лиспа и ФП, а также на многих университетских сайтах.

Развитие парадигм программирования

Знакомое нам из курса философии слово "парадигма" имеет в информатике и программировании узко профессиональный смысл, сближающий их с лингвистикой. Парадигма программирования как исходная концептуальная схема постановки проблем и их решения является инструментом грамматического описания фактов, событий, явлений и процессов, возможно, не существующих одновременно, но интуитивно объединяемых в общее понятие.

Каждая парадигма программирования имеет свой круг приверженцев и класс успешно решаемых задач. В их сфере приняты разные приоритеты при оценке качества программирования, отличаются инструменты и методы работы и соответственно - стиль мышления и изобразительные стереотипы. Нелинейность развития понятий, зависимость их обобщения от индивидуального опыта и склада ума, чувствительность к моде и внушению позволяют выбору парадигм в системе профессиональной подготовки информатиков влиять на
восприимчивость к новому.

Ведущая парадигма прикладного программирования на основе **императивного** управления и **процедурно-операторного стиля** построения программ получила популярность более пятидесяти лет назад в сфере узкопрофессиональной деятельности специалистов по организации вычислительных и информационных процессов. Последнее десятилетие резко расширило географию информатики, распространив ее

на сферу массового общения и досуга. Это меняет критерии оценки информационных систем и предпочтения в выборе средств и методов обработки информации.

Консервирующееся в наши дни доминирование одной архитектурной линии, стандартного интерфейса, типовой технологии программирования и т.д. чревато потерей маневренности при обновлении информационных технологий. Особенно уязвимы в этом отношении люди, привыкшие в учебе прочно усваивать все раз и навсегда. При изучении языков программирования подобные проблемы обходят за счет одновременного преподавания различных языков программирования или предварительного изложения основы, задающей грамматическую структуру для обобщения понятий, изменяемость которых трудно улавливается на упрощенных учебных примерах [14]. Именно такую основу дает изучение функционального программирования тем, что оно нацелено на изложение и анализ парадигм, сложившихся в практике программирования в разных областях деятельности с различным уровнем квалификации специалистов, что может быть полезно как концептуальная основа при изучении новых явлений в информатике.

Общие парадигмы программирования, сложившиеся в самом начале эры компьютерного программирования, - парадигмы прикладного, теоретического и функционального программирования в том числе, имеют наиболее устойчивый характер.

Прикладное программирование подчинено проблемной направленности, отражающей компьютеризацию информационных и вычислительных процессов численной обработки, исследованных задолго до появления ЭВМ. Именно здесь быстро проявился явный практический результат. Естественно, в таких областях программирование мало чем отличается от кодирования, для него, как правило, достаточно операторного стиля представления действий. В практике **прикладного программирования** принято доверять проверенным шаблонам и библиотекам процедур, избегать рискованных экспериментов.
Ценится точность и устойчивость научных расчетов. Язык Фортран - ветеран прикладного программирования. Лишь в последнее десятилетие он стал несколько уступать в этой области Паскалю-Си, а на суперкомпьютерах - языкам параллельного программирования, таким как Sisal.

Теоретическое программирование придерживается публикационной направленности, нацеленной на сопоставимость результатов научных экспериментов в области программирования и информатики. Программирование пытается выразить свои формальные модели, показать их значимость и фундаментальность. Эти модели унаследовали основные черты родственных математических понятий и утвердились как **алгоритмический подход** в информатике. Стремление к доказательности построений и оценка их эффективности, правдоподобия, правильности, корректности и других формализуемых отношений на схемах и текстах программ послужили основой **структурированного программирования** и других методик достижения надежности процесса разработки программ, например **грамотное программирование**. Стандартные подмножества Алгола и Паскаля, послужившие рабочим материалом для теории программирования, сменились более удобными для экспериментирования **аппликативными языками**, такими как ML, Miranda, Scheme и другие диалекты Лиспа. Теперь к ним присоединяются подмножества C и Java.

Функциональное программирование сформировалось как дань математической направленности при исследовании и развитии искусственного интеллекта и освоении новых горизонтов в информатике. Абстрактный подход к представлению информации, лаконичный, универсальный стиль построения функций, ясность обстановки исполнения для разных категорий функций, свобода рекурсивных построений, доверие интуиции математика и исследователя, уклонение от бремени преждевременного решения непринципиальных проблем распределения памяти, отказ от необоснованных ограничений на область действия определений – все это увязано Джоном Мак-Карти в идее языка Лисп [1]. Продуманность и методическая обоснованность первых реализаций Лиспа позволила быстро накопить опыт решения новых задач, подготовить их для прикладного и теоретического программирования. В настоящее время существуют сотни

функциональных языков программирования, ориентированных на разные классы задач и виды технических средств.

Основные парадигмы программирования сложились по мере возрастания сложности решаемых задач. Произошло расслоение средств и методов программирования в зависимости от глубины и общности проработки технических деталей организации процессов компьютерной обработки информации. Выделились разные стили программирования, наиболее зрелые из которых - машинно-ориентированное, системное, логическое, трансформационное, и высокопроизводительное/параллельное программирование.

Машинно-ориентированное программирование характеризуется аппаратным подходом к организации работы компьютера, нацеленным на доступ к любым возможностям оборудования. В центре внимания - конфигурация оборудования, состояние памяти, команды, передачи управления, очередность событий, исключения и неожиданности, время реакции устройств и успешность реагирования. Ассемблер в качестве предпочтительного изобразительного средства на некоторое время уступил языкам Паскаль и Си даже в области микропрограммирования, но усовершенствование пользовательского интерфейса может восстановить его позиции.

Системное программирование долгое время развивалось под прессом сервисных и заказных работ. Свойственный таким работам производственный подход опирается на предпочтение воспроизводимых процессов и стабильных программ, разрабатываемых для многократного использования. Для таких программ оправдана компиляционная схема обработки, статический анализ свойств, автоматизированная оптимизация и контроль. В этой области доминирует императивно-процедурный стиль программирования, являющийся непосредственным обобщением операторного стиля прикладного программирования. Он допускает некоторую стандартизацию и модульное программирование, но обрастает

довольно сложными построениями, спецификациями, методами тестирования, средствами интеграции программ и т.п. Жесткость требований к эффективности и надежности удовлетворяется разработкой профессионального инструментария, использующего сложные ассоциативно семантические эвристики наряду с методами синтаксически-управляемого конструирования и генерации программ. Бесспорный потенциал такого инструментария на практике ограничен трудоемкостью освоения - возникает квалификационный ценз [15].

Высокопроизводительное программирование нацелено на достижение предельно возможных характеристик при решении особо важных задач. Естественный резерв производительности компьютеров **- параллельные процессы**. Их организация требует детального учета временных отношений и неимперативного стиля управления действиями. Суперкомпьютеры, поддерживающие высокопроизводительные вычисления, потребовали особой техники системного программирования. **Графово-сетевой подход** к представлению систем и процессов для параллельных архитектур получил выражение в специализированных языках параллельного программирования и суперкомпиляторах, приспособленных для отображения абстрактной иерархии процессов уровня задач на конкретную пространственную структуру процессоров реального оборудования [11,16].

Логическое программирование возникло как упрощение функционального программирования для математиков и лингвистов, решающих задачи символьной обработки. Особенно привлекательна возможность в качестве понятийной основы использовать недетерминизм, освобождающий от преждевременных упорядочений при программировании обработки формул. Продукционный стиль порождения процессов с возвратами обладает достаточной естественностью для лингвистического подхода к уточнению формализованных знаний экспертами, снижает стартовый барьер [10].

Трансформационное программирование методологически объединило технику оптимизации программ, макрогенерации и частичных вычислений. Центральное понятие в этой области - эквивалентность информации. Она проявляется в определении преобразований программ и процессов, в поиске критериев применимости преобразований, в выборе стратегии их использования. Смешанные вычисления, отложенные действия, "ленивое" программирование, задержанные процессы и т.п. используются как методы повышения эффективности информационной обработки при некоторых дополнительно выявляемых условиях [9].

Экстенсивные подходы к программированию – естественная реакция на радикальное улучшение эксплуатационных характеристик оборудования и компьютерных сетей. Происходит переход вычислительных средств из класса технических инструментов в класс бытовых приборов. Появилась почва для обновления подходов к программированию, а также возможность реабилитации старых идей, слабо развивавшихся из-за низкой технологичности и производительности ЭВМ. Представляет интерес формирование исследовательского, эволюционного, когнитивного и адаптационного подходов к программированию, создающих перспективу рационального освоения реальных информационных ресурсов и компьютерного потенциала.

Исследовательский подход с учебно-игровым стилем профессионального, обучающего и любительского программирования может дать импульс изобретательности в совершенствовании технологии программирования, не справившейся с кризисными явлениями на прежней элементной базе.
Эволюционный подход с мобильным стилем уточнения программ достаточно явно просматривается в концепции объектно-ориентированного программирования, постепенно перерастающего в субъектно-ориентированное и даже эго-ориентированное программирование. Повторное использование определений и наследование свойств объектов могут удлинить жизненный цикл отлаживаемых информационных обстановок, повысить надежность их функционирования и простоту

применения. Когнитивный подход с **интероперабельным стилем визуально-интерфейсной разработки открытых систем** и использование новых аудио-видео средств и нестандартных устройств открывают пути активизации восприятия сложной информации и упрощения ее адекватной обработки. **Адаптационный подход** с эргономичным стилем индивидуализируемого конструирования персонифицированных информационных систем предоставляет информатикам возможность грамотного программирования, организации и обеспечения технологических процессов реального времени, чувствительных к человеческому фактору.

Направление развития парадигмы программирования отражает изменение круга лиц, заинтересованных в развитии и применении информационных систем. Многие важные для практики программирования понятия, такие как события, исключения и ошибки, потенциал, иерархия и ортогональность построений, экстраполяция и точки роста программ, измерение качества и т.д. не достигли достаточного уровня абстрагирования и формализации. Это позволяет прогнозировать развитие парадигм программирования и выбирать учебный материал на перспективу **компонентного программирования** (COM/DCOM, Corba, UML, .Net и др.). Если традиционные средства и методы выделения многократно используемых компонентов подчинялись критерию модульности, понимаемой как оптимальный выбор минимального сопряжения при максимальной функциональности, то современная элементная база допускает оперирование поликонтактными узлами, выполняющими простые операции.

Парадигма программирования в образовательном процессе является инструментом формирования профессионального поведения. Программирование прошло путь от профессиональной деятельности высококвалифицированной элиты технических специалистов и научных работников до времяпрепровождения активной части цивилизованного общества. Освоение информационных систем через понимание с целью компетентных действий и ответственного применения техники сменилось интуитивными навыками хаотичного воздействия на информационную среду со скромной надеждой на везение, без претензий на знание. Обслуживание центров коллективного пользования, профессиональная поддержка целостности информации и подготовки данных почти полностью отступили

перед самообслуживанием персональных компьютеров, независимым функционированием сетей и разнородных серверов со взаимодействием различных коммуникаций.

Противопоставление разрабатываемых программ, обрабатываемых данных и управления заданиями уступает представлению об интерфейсах, приспособленных для участия в информационных потоках подобно навигации. Прежние критерии качества: скорость, экономия памяти и надежность обработки информации - все больше заслоняются игровой привлекательностью и широтой доступа к мировым информационным ресурсам. Замкнутые программные комплексы с известными гарантиями качества и надежности форсированно вытесняются открытыми информационными комплектами с непредсказуемым развитием состава, способов хранения и обработки информации.

Эти симптомы обновления парадигмы программирования определяют направление изменений, происходящих в системе базовых понятий, в концепции информации и информатики. Тенденция использования интерпретаторов (точнее неполной компиляции) вместо компиляторов, анонсированная в концепции Java в сравнении с Си, и соблазн объектно-ориентированного программирования на фоне общепринятого императивно-процедурного стиля программирования можно рассматривать как неявное движение к функциональному стилю [2]. Моделирующая сила функциональных формул достаточна для полноценного представления разных парадигм, что позволяет на их основе экстраполировать приобретение практических навыков организации информационных процессов на будущее.

Литература

1. McCarthy J. LISP 1.5 Programming Mannual.- The MIT Press., Cambridge, 1963, 106p.

2. Гилдер Дж. Программное обеспечение: переворот грядет.- М. Открытые системы. N 3, 1996, с. 54-60

3. Хендерсон П. Функциональное программирование.- М.: Мир, 1983

4. Филд А., Харрисон П. Функциональное программирование. Перевод под редакцией В.А.Горбатова. – М. Мир, 1993, 638с

5. Скотт Д. Теория решеток, типы данных и семантика. // В кн. Данные в языках программирования. – М. Мир, 1982, с.25-53

6. Хьювенен Э., Сеппанен Й. Мир Лиспа., т.1,2, М.: Наука, 1994

7. Graham P. ANSI Common Lisp. //Prentice Hall,1996, 432p

8. Оллонгрен А. Определение языков программирования интерпретирующими автоматами. М.: Мир, 1977, 288 с.

9. Ершов А.П. Смешанные вычисления: потенциальные приложения и проблемы исследования.- Тез. Докл. Всесоюзная конф. "Методы математической логики в проблемах искусственного интеллекта и систематическое программирование", ч.2, Вильнюс, 1980, с. 26-55

10. Малпас Дж. Реляционный язык Пролог и его применение.- М.: "Наука", 1990, 463с.

11. Cann D. C. SISAL 1.2: A Brief Introduction and tutorial. Preprint UCRL-MA-110620. Lawrence Livermore National Lab., Livermore, California,May, 1992

12. Левин Д.Я. Язык сверх высокого уровня Сетл и его реализация (для БЭСМ-6). //Новосибирск: Наука, 1983

13. Knoop J. Optimal Interprocedural Program Optimization. A New Framework aand Its Application. //Springer, LNCS-1428, 1998, 288p

14. Weinberg G.M. The Psychology of Computer Programming.- New York: Van Norstand Reinhold Comp., 1971

15. Поттосин И.В. Система СОКРАТ: Окружение программирования для встроенных систем.-Новосибирск, 1992.-20с. (Препр./РАН. Сиб. отд-ние. ИСИ; N11)

16. Евстигнеев В.А. Применение теории графов в программировании.- М.: Наука, 1985