Э.Хювёнен, И.Сеппянен

МИР ЛИСПА. Т.2: МЕТОДЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

Двухтомник финских специалистов, содержащий введение в язык Лисп, методы и системы программирования. Этот язык широко известен и применяется в задачах символьной обработки информации, обработки естественных языков, искусственного интеллекта, экспертных систем, систем логического программирования. Изложение языка и примеры основаны на последней версии, которая станет стандартом языка. В книге приведены конкретные задачи с ответами и решениями. Во 2-м томе изложены методы и системы программирования.

Для программистов разной квалификации, для всех, использующих язык Лисп.

Содержание

УКАЗАТЕЛЬ ИМЕН И СОКРАЩЕНИЙ

В этом приложении собраны встречающиеся в тексте имена и сокращения. Одновременно оно может служить указателем авторов по перечням литературы. С его помощью можно найти литературу, которая вследствие разбиения по темам приведена в разных разделах. Символы и зарезервированные слова Коммон Лиспа образуют отдельный указатель (приложение 1).

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Указатель составлен из встречающихся в тексте понятий и терминов, а также из специальных терминов языка Лисп. Имена и. сокращения собраны в свой указатель (приложение 2) также, как символы Ком мои Лиспа и имена, определяемые в примерах программирования (приложение 1).

Опасность не в том, что машина начинает уподобляться человеку, а в том, что человек превращается в подобие машины.

Сидней Харрис

ВВЕДЕНИЕ

- **Скачок в развитии вычислительной техники**
- **Лисп – основа искусственного интеллекта**
- **Учебник Лиспа на финском языке**
- **Язык Лисп и функциональное программирование**
- **Методы программирования**
- **Среда программирования**
- **Примеры программ**
- **Развитие Лисп-культуры и Лисп-систем**
- **На кого расчитана книга**
- **Терминология**
- **Иконология**
- **Благодарности**

Скачок в развитии вычислительной техники

Цель данной книги – дать читателю представление о языке программирования Лисп, символьной обработке и о мире искусственного интеллекта. Эти идеи

и базирующаяся на них вычислительная техника нового поколения машин открывают дорогу новому технологическому и промышленному скачку. Через искусственный интеллект, непосредственное общение с машиной на естественном языке и использование экспертных систем вычислительная техника в состоянии справиться с задачами, которые раньше мог решить только человек.

Вычислительные машины и автоматы позволяют механизировать решение как задач рутинного характера, так и интеллектуальных задач. Развитие вычислительной техники происходит скачкообразно. Скачок, наблюдаемый в данный момент, качественный и более

глубокий по своему характеру, чем все предыдущие. Системы обработки данных вторгаются в области, традиционно считавшиеся сферой деятельности человека, где для решения различных проблем требовались компетентность и знания, имеющиеся у человека.

Лисп – основа искусственного интеллекта

Лисп – важнейший язык, используемый при символьной обработке и в исследованиях по искусственному интеллекту. Эти работы, начатые в США в MIT уже в середине 50-х годов, проводилось преимущественно на языке Лисп. Бóльшая часть имеющихся на рынке программ символьной обработки, систем искусственного интеллекта и программ работы с естественным языком написана на Лиспе. Многие методы, используемые в области искусственного интеллекта, основаны на особых свойствах языка Лисп. Лисп составляет основу для обучения методам искусственного интеллекта, исследованиям и практическому применению этой области, иными словами, Лисп вводит в мир символьной обработки и искусственного интеллекта.

Учебник Лиспа на финском языке

В основу книги положены лекции, прочитанные в 1981 г. на отделении общего языкознания Хельсинкского университета, и серия из 11 статей, опубликованных в журнале "Prosessori" (Процессор) в 1982–1983 гг., а также материалы некоторых других наших публикаций. Рабочая версия книги и ее первое издание были использованы в качестве учебника в учебном центре вычислительной техники фирмы Nokia (NITEC).

Язык Лисп и функциональное программирование

В первой части книги, посвященной миру Лиспа, рассматриваются важнейшие сферы применения символьной обработки и искусственного интеллекта и отличия языка Лисп от других языков программирования. Изложены понятия, языковые конструкции и

механизмы языка. Основные методы программирования на Лиспе – функциональное и рекурсивное программирование – рассматриваются более детально и систематично с подразделением типов функций и форм рекурсии на отдельные виды, что является новшеством по сравнению с предшествующими публикациями по Лиспу. Первая часть книги может использоваться для самостоятельного изучения как языка Лисп, так и функционального программирования.

Основой изложения нами был выбран диалект Коммон Лисп, ставший "де-факто" промышленным стандартом языка Лисп. В книге представлены все важнейшие языковые формы и свойства конструкций Коммон Лиспа, а также типы функций и данных. Материал изложен не в виде справочного руководства, а в логически последовательной и поэтому пригодной для обучения форме. Даются пояснения для рассматриваемых в книге понятий и методов, которые в справочных руководствах обычно не освещаются. В то же время сведено к минимуму количество трудночитаемой системной технической информации. Чтобы можно было использовать книгу и как справочное руководство, в конце ее приведено краткое описание всего Коммон Лиспа.

Однако изложение не ограничивается рамками Коммон Лиспа. По мере надобности приводятся сведения и о важнейших свойствах и особенностях других систем и расширениях стандарта.

Методы программирования

В этой части рассмотрим прежде всего методы, технику и системы программирования.

Вначале познакомимся с методом программирования, основанном на использовании операторов. Наряду с функциональным программированием это основной используемый в Лиспе метод программирования, который поддерживается многими конструкциями языка. Далее рассмотрим программирование, управляемое данными, опирающееся в основном на присущую

Лиспу возможность трактовать данные как программу и наоборот.

Ближе к методам искусственного интеллекта подводит техника программирования, известная под названием сопоставления с образцом. Идея сопоставления с образцом состоит в распознавании данных с помощью представленных в общем виде образцов. Этот метод, в частности, нашел применение, в "понимающих человеческий язык" автоматах и других системах. Сопоставление с образцом можно использовать в продукционном программировании, являющемся основным методом реализации экспертных систем.

Переходя к программированию систем, базирующихся на логике, рассмотрим так называемую унификацию, или взаимное приведение двух структур к общему виду. С помощью процедур унификации и поиска построим программу, реализующую на Лиспе небольшой интерпретатор Пролога. Созданная система – это пример системы, включающей язык декларативного представления знаний, действия с которыми можно определить с помощью логических правил, не задавая алгоритмического описания этих действий, как это обычно делается в программировании.

После этого рассматриваются включенное в состав многих современных Лисп-систем объектно-ориентированное программирование и как пример такой системы объектно-ориентированная система Flavor, являющаяся расширением Коммон Лиспа. Явными аналогами объектно-ориентированной организации данных и методов программирования являются используемые в представлении знаний фреймовые структуры и механизмы наследования в семантических сетях, процедурное представление (демоны) и так далее. Объектно-ориентированная модель может с успехом использоваться и в системном программировании, и в моделировании.

В заключение описания методов программирования приведем обзор языковых конструкций и методов, а также критериев, используемых при оценке стиля программирования.

Среда программирования

Одно из важнейших преимуществ Лиспа перед другими языками программирования – наличие доступной из системы среды программирования, обладающей широкими возможностями. Стандарт Коммон Лиспа определяет среду программирования лишь в части вызова различных подсистем, не касаясь конкретно их функционирования. В главе 3 рассмотрим первичную среду Коммон Лиспа и для примера наиболее развитые лисповские среды Интерлиспа и Зеталиспа.

Примеры программ

В первой и второй частях книги приведены небольшие примеры лисповских функций и программ. Более подробные примеры программ, дающие одновременно представление о широте сферы применения предлагаемых Лиспом и символьной обработкой возможностей, сосредоточены в гл. 4 с целью сведения в единую систему различных конструкций и методов программирования.

Чтобы дать общее представление о механизмах языка и показать применимость Лиспа в системном программировании, прежде всего напишем программу, реализующую небольшой интерпретатор Лиспа на самом языке Лисп. Программа Миксима из следующего примера является "младшим братом" разработанной в MIT экспертной системы Macsyma по аналитической математике. В числе прочего представлен разбор выражений и обработка их в виде дерева посредством использования объектно–ориентированного программирования и макропрограммирования. Примером обработки естественного языка будет небольшая программа, переводящая с финского языка на псевдоязык сплетника. В этих программах в первую очередь используется традиционное функциональное программирование. Продукционное программирование будет проиллюстрировано на примере программирования экспертной системы. В качестве примера объектно-ориентированной системы запрограммируем действующую модель Солнечной системы.

Развитие Лисп-культуры и Лисп-систем

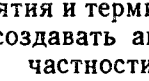
В конце книги приводится обзор развития языка Лисп и Лисп-систем в университетах, в исследовательских центрах и в промышленности США, а также дается представление о распространении Лисп-культуры по всему миру вплоть до Финляндии. Отдельно будут отмечены важнейшие переломные моменты в истории Лиспа. Книга заканчивается изложением истории и направлений развития Лисп-машин и Пролог-машин пятого поколения.

На кого расчитана книга

Книга расчитана на учащихся и исследователей как в области технических, естественных, так и гуманитарных специальностей, а также для специалистов по вычислительной технике и других близких областях. Надеемся, что эта книга будет иметь широкую сферу применения и использоваться не только в качестве учебника в университетах и высших школах, но и в качестве материала курсов повышения квалификации в промышленности и для самостоятельного обучения Лиспу, а также будет полезна специалистам и псем интересующимся проблематикой искусственного интеллекта.

Терминология

В вычислительной технике время от времени рождаются новые понятия и термины, для которых приходится искать или создавать аналоги на других языках, в частности на финском. Обычно в вычислительной технике проблемы именования наиболее сложны в быстро развивающихся областях, где еще не устоялась терминология на языке-оригинале. Особенно много новых понятий требуется и создается в программировании и технике искусственного интеллекта, где сам по себе вопрос в основном сводится к определению и формированию понятий.



Мы попытались найти для английских терминов как можно более точные и практичные соответствия. В какой степени это удалось, судить читателям. Оригинальный термин дан в скобках в момент его определения или разъяснения. Термины на финском и английском языках собраны в приложении[1]. Выбранные соответствия, конечно, не являются наилучшими. Поэтому с благодарностью примем все предложения читателей, касающиеся терминологии, а также и другие замечания и предложения по содержанию книги и изложению материала.

Иконология

Для иллюстрации и лучшего восприятия текста мы использовали взятые из различных источников и окрашенные юмором символическую графику и афоризмы. Как в тексте, так и на картинках часто повторяющимся мотивом является дерево, которое символизирует не только логические и физические конструкции Лиспа, но и воплощаемые в них деревья данных и модели мира. Дерево мира, дерево данных и дерево жизни – центральные символы картины мира, знаний, жизни, счастья и благополучия во многих странах и культурах. И в "Калевале" есть большой дуб:

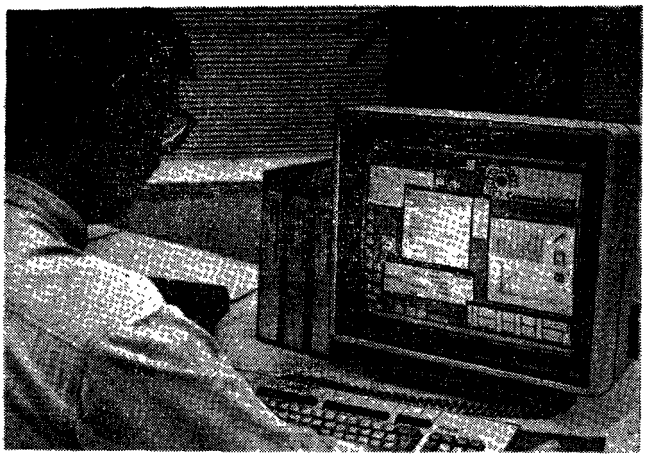
Если кто там поднял ветку,
Тот нашел навеки счастье;
Кто принес себе верхушку,
Стал навеки чародеем;
Калевала 2:191–194[2]

Дерево всегда было важным материалом, на котором тем или иным способом основывалась техника и

---

[1] Мы, естественно, приводим не финские, а рекомендуемые нами русские эквиваленты английских терминов. Впрочем, к ним применимы все те оговорки, которые сделаны авторами книги в отношении финских терминов. *–Прим. ред.*

[2] Перевод Л.Бельского.



Лисп-машина серии Xerox 1100 и базирующаяся на Коммон Лиспе среда Common Loops.

культура. Из дерева изготовлялись различные предметы обихода, в том числе и вспомогательные средства, и инструменты для работы, с помощью которых в свою очередь снова делались новые предметы, машины и более совершенные рабочие инструменты. В трудные времена древесные отходы добавлялись и в хлеб. Без изготовленной из дерева бумаги не могла бы существовать и современная письменная культура, в том числе и эта книга. Так и в Лиспе символьные деревья являются как инструментом и средством программирования, так и обрабатываемым программами материалом, из которого строятся различные применения, создаются системы и более развитые инструменты.

Dot Parrett.

На дереве в его различных формах основывается также финская хозяйственная жизнь. Однако все более важным товаром становятся наши знания и компетентность, новые возможности применения и продажи которых в виде экспертных систем открывают методы искусственного интеллекта.

Говорят, что центр тяжести промышленности перемещается "от дерева к мысли". В этой книге представлены те основы, с помощью которых человеческие знания и компетентность могут быть перенесены из головы в (лисповское) дерево в этом вновь возникающем, дружественном окружающей среде "деревообрабатывающем производстве".

Благодарности

В процессе подготовки и написания книги нам много раз в различной форме оказывали помощь: Орри Эрлинг, Хейки Хонкио, Ниссе Хасберг, Еийя Ярвинен, Кари Каллио, Матти Карьялайнен, Фред Карлссон, Урпо Карьялайнен, Антеро Катайнен, Суви Кеттула, Эрки Куренниеми, Ора Лассила, Кари Маннерсало, Клаус Оэш, Пекка Пиринен, Отто Романовски, Тапани Саволайнен, Пертти Тапола, Пекка Толонен, Эва Вильянен, Матти Виртанен и Мартти Юлёстало. Мы хотим выразить признательность за помощь в работе Юхани Карвинену и Аннели Карппинен из издательства "Кирьяюхтюмя", а также Ханну Ромппанену из Гуммеруксена.

Подготовка и редактирование книги стали возможными благодаря Техническому университету и Финской Академии наук. Digital Equipment Corporation Oy, Knowledge Engineering Ky, Nokia Informaatiojärjestelmät, Rank Xerox, Texas Instruments и Министерство образования предоставили машинное время, технику и материалы. Благодарим Технический университет и Министерство образования за оказанную нам материальную помощь.

Отаниеми, 1987 Авторы

Прочел строку, прочел вторую, закипает кровь во мне.
Й. Рунеберг

1 ВВЕДЕНИЕ В МЕТОДЫ И СИСТЕМЫ ПРОГРАММИРОВАНИЯ

- **Основные типы знаний**
- **Методы представления знаний**
- **Процедурные и декларативные знания**
- **Способы решения проблем**
- **Лисп предлагает различные модели**
- **Методы и стиль программирования**
- **Парадигмы программирования**
- **Литература**

Функционирование интеллектуальных программ, с одной стороны, основано на содержащихся в системе данных и *знаниях* (knowledge), с другой – на использующем эти знания методе решения проблемы. В искусственном интеллекте в отличие от численного представления данных в научно-технических вычислениях для представления знаний обычно используются символьные структуры. Для решения задач из области искусственного интеллекта вместо традиционного алгоритмического подхода применяются различные эвристические методы поиска решения.

Различные уровни программирования задач искусственного интеллекта представлены на рис. 1.1. Первая часть книги в основном относилась к уровню программирования.

Вторая часть книги посвящена методам, механизмам и системам программирования, с помощью которых реализуются языки и средства, действительно относящиеся к работе со знаниями, такие как оболочки экспертных систем, анализаторы языка и другие прикладные программы. Поскольку методы программирования часто разрабатываются на основе моделей

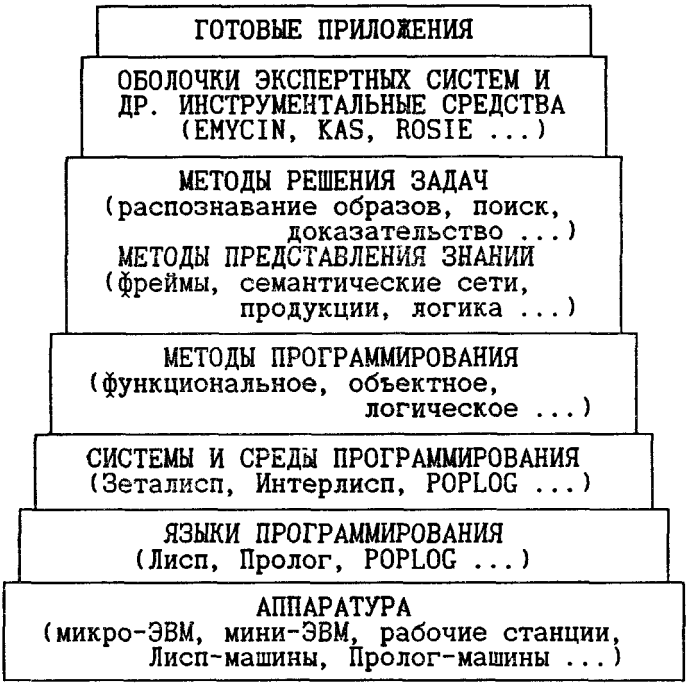


Рис. 1.1 Уровни программирования искусственного интеллекта и их трактовка в этой книге.

исходных данных и моделей мышления, то для многих методов можно найти явные аналоги на уровне представления знаний и методов решения проблем.

Основные типы знаний

Определение знания как понятия – трудная философская проблема. В области искусственного интеллекта наиболее важные типы знаний классифицируются следующим образом:

1. *Объекты* (object) и их *свойства* (property, attribute). Объекты – это существующие в прикладной области

универсальные понятия и их представители, например: живые существа (программист, кошка, Мауно Койвисто), предметы или материалы (компьютер, железо) или абстрактные понятия (легкость, счастье).

2. *События* (event). События описывают участие объектов в деятельности и ситуациях ("Обезьяна съела банан", "Король находится в замке"). События характеризуют, например, время, место и причинно-следственные отношения.

3. *Действия* (performance, function, process). Обычно интеллектуальная деятельность предполагает также способность совершать действия, т.е. процедурные знания о том, каким образом что-то делается, например каким образом из старых данных на основе правил выводятся новые.

4. *Метазнания* (meta-knowledge). Метазнания – это знания о знаниях и их использовании, например способность выбрать метод решения для проблем разных типов.

Методы представления знаний

Представление знаний (knowledge representation) – это основная область исследований по искусственному интеллекту. Любая работающая со знаниями программа должна каким-то образом отображать знания из своей области применения. Обычно для этого не достаточно примитивных структур данных, используемых в традиционной обработке данных, таких как числа, массивы, записи и др., и методов работы с ними. В искусственном интеллекте используются символьные языки представления знаний и формализмы, стоящие на более высоком понятийном уровне. Качество системы определяется в первую очередь тем, насколько широки и разнообразны эти знания.

Ниже приведены важнейшие из общих методов и формализмов, разработанных для представления и обработки знаний.

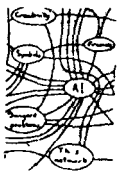
1. *Логика* (logic). В программах искусственного интеллекта особенно часто используются различные формы логики предикатов первого порядка. Например, язык Пролог непосредственно основывается на логике исчисления предикатов. Основное преимущество базирующихся на логике формализмов состоит в том, что обычно с их помощью проще обеспечить корректность структур и решений системы, чем с помощью других способов представления. Кроме Пролога в процессе исследований по искусственному интеллекту были разработаны и другие языки, базирующиеся на логике, такие как: PLANNER, CONNIVER, ALICE, QA2/3 и FOL.
2. *Продукционные системы* (production system, rule based system) – важнейший из используемых в экспертных системах способов представления знаний. Основная идея заключается в ассоциировании с соответствующими действиями набора условий в виде правил типа (ЕСЛИ–ТО):

   IF условия THEN действия

   На основе таких правил создано множество языков представления знаний (MRS, ROSIE, KAS и другие), некоторые из которых являются законченными программными продуктами (например, OPS5, M.1, S.1, ART и другие).
3. *Фреймы* (frame) и объекты (object). С помощью фреймов можно определять объекты с наследуемыми свойствами и конструкции, определяемые по умолчанию, напоминающие структуры, но более общего характера и с большим числом возможностей. Фреймы часто выстраиваются в иерархию понятий (conceptual hierarchy), в которой наследуют различные свойства своих "надпонятий" (ср. с объектно-ориентированным способом программирования). С операциями присваивания значений

фреймам и другими операциями можно сочетать сложные побочные действия и взаимовлияния (procedural attachment). В числе прочих наиболее известных языков представления знаний на основе фреймов являются KRL, OWL, UNITS, FRL, AIMDS и KL-ONE/TWO.

4. *Семантические сети* (semantic net) и классы (class, category). В семантической сети понятия и классы, а также отношения и связи между ними представлены в виде сети. Преимущества такого формализма заключаются в его наглядности и непосредственной связанности понятий через сеть, которая позволяет быстро находить связи понятий и на этой основе управлять принимаемыми решениями.



Используются и другие связанные с конкретным применением способы представления, но они менее распространены и, как правило, не годятся для использования в общем случае.

Не всегда можно однозначно сказать, какой формализм представления использован в системе. Например, некоторые формализмы семантической сети можно рассматривать как синтаксические варианты логики предикатов или фреймовых систем, продукционные системы – как логические и так далее.

В рамках одной и той же системы для представления различных видов знаний могут быть использованы различные формализмы. Иногда возможна запись одной и той же информации в нескольких формализмах для различных способов ее использования.

Процедурные и декларативные знания

В программировании знания могут выражаться как в виде пассивных декларативных структур данных, так и в виде процедурных знаний по их интерпретации и обработке. Принятое в традиционном программировании разграничение программы на "данные" и саму "программу" часто невозможно. Например, транслято-ры и интерпретаторы – это программы, которые рассмат-

ривают другие программы как данные. В зависимости от конечной цели одни и те же данные могут в каждом конкретном случае представляться или интерпретироваться различным образом либо как декларативные, либо как процедурные.

В традиционных вычислительных машинах фон неймановского типа представление данных и команд на машинном уровне одинаково. В языках высокого уровня (Фортран, Паскаль, Бейсик и др.), напротив, программа и данные обычно четко отделены друг от друга. Программа (процедура) – это свой оперирующий с пассивными данными замкнутый мир и зафиксированный на этапе трансляции образ действий. Поэтому воздействие на нее возможно только с помощью параметров.

В логическом программировании в представлении программы и данных наблюдается другая крайность. В логическом программировании нет необходимости отделять данные от программы, программу можно трактовать как совокупность логических правил.

В Лиспе в отношениях между программой и данными достигнут некоторый компромисс. Форма представления программы и данных едина, и программу можно свободно трактовать как данные так же, как данные можно интерпретировать и применять как программу. У языковых конструкций есть две интерпретации, зависящие от их использования. Использование и интерпретацию выражений определяет программист. Средства и механизмы, необходимые для вызова интерпретатора и применения объектов (EVAL, APPLY, FUNCALL, QUOTE, FUNCTION и так далее), так же как и средства, используемые для предотвращения вычислений, находятся в распоряжении программиста. Преобразования программы в данные и данных в программу определяются программистом.

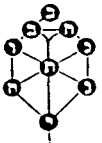
В некоторых Лисп-системах возможно объектно-ориентированное программирование (object oriented programming), подобно используемому в системе Смолтолк, в которой данные и процедуры отделены друг от друга, но тем не менее они составляют единый

самостоятельный объект (object). Управление вычислениями осуществляется в основном с помощью сообщений, которыми обмениваются объекты. Например, сложение выполняется посылкой числу сообщения "+" и слагаемого, умножение – посылкой сообщения "*" и множителя.

Способы решения проблем

В системах искусственного интеллекта используются разнообразные методы решения проблем, которые в той или иной форме сводятся в основном к более простым действиям, таким как *поиск* (search), *идентификация* (identification, recognition), *сравнение* (comparison), *логический вывод* (inference).

Проблема поиска формируется на языке *состояний* (state) и *операторов* (operator), с помощью которых можно перемещаться из одного состояния в другое. Все возможные состояния и переходы между ними образуют или описывают (сетеобразное) *пространство поиска* (search space) решаемой проблемы. Например, в программе игры в шахматы различные позиции можно рассматривать как состояния, а различные ходы – как операции. В программе медицинской диагностики состояниями могут быть частично установленные диагнозы, а операциями – жалобы пациента, результаты анализов, симптомы и другие связанные с состоянием пациента признаки и данные.



Решение (solution) проблемы в пространстве поиска означает нахождение такой последовательности операций, с помощью которой можно пройти из *начального состояния* (initial state) проблемы в некоторое *заключительное или целевое состояние* (goal), например в шахматах из начального положения к мату.

В экспертных системах обычно используются методы поиска и сравнения, в которых начальное состояние памяти преобразуется в конечное состояние с помощью цепочки решений, принимаемых на основе знаний, заключенных в правилах (операторах).

Лисп предлагает различные модели

В зависимости от области применения и от ситуации механизмы представления знаний и методы решения проблем программируются с помощью различных методов и механизмов. В области искусственного интеллекта не выработаны какие-либо единые методы представления знаний, решения проблем и техники программирования, поэтому исследователю необходимо ознакомиться с множеством различных методов и научиться объединять их в соответствии с необходимостью. Путем соединения различных методов достигается более естественное описание, программирование становится более простым, а решение – лучше.

Методы и стиль программирования

Кроме приведенного в первой части книги функционального программирования Лисп поддерживает также другие методы и стили программирования, важнейшие из которых мы рассмотрим в следующих главах. Лисп является одновременно и прекрасным рабочим инструментом, и формализмом, с помощью которого можно обучать, исследовать и развивать дальше различные методы и связанные с ними понятия.

Факторы, влияющие на выбор используемых средств и стиля, часто противоречивы. Например, ясность программы с точки зрения человека и эффективность с точки зрения вычислительной машины требуют различных способов изображения. Короткую программу легко написать, но не всегда легко понять. Память и требуемое для вычислений время обычно находятся на разных чашах весов. При подходящем выборе способа представления и метода решения можно и в такой противоречивой ситуации найти удачный компромисс.

Техника и стиль программирования зависят также от вкуса программиста. Иногда задачу можно решить красиво и эффективно многими различными способами. Разные программисты по-разному оценивают различные аспекты, и у каждого программиста со временем

складывается свой стиль и взгляды. Это во многом определяется самым первым языком программирования, с которым столкнулся программист, и освоенными при этом выразительными средствами.[1)]

Если уже освоен один из стилей, то овладение новым стилем и техникой программирования и мотивировка его использования не всегда просты. Часто начинающему легче научиться лисповскому функциональному мышлению или реляционному стилю Пролога, чем программисту с опытом работы на операторном языке. Привычка – вторая натура и в программировании.

Разнообразие и единство форм Лиспа и свобода их объединения, независимость типов и специальные механизмы абстракции дают программисту большую свободу в выборе различных стилей и методов программирования. Лисп предоставляет не только свободу выбора средств из числа имеющихся. Он позволяет пробовать и изобретать новые методы и вырабатывать индивидуальные стили.

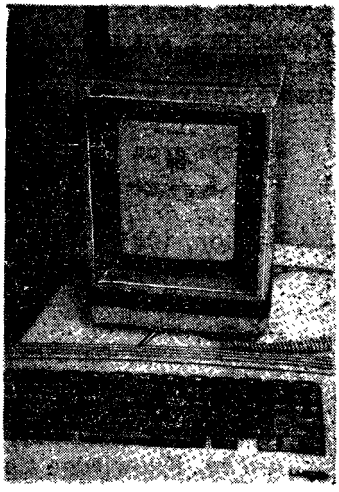
Для развития Лиспа характерно то, что лисповские формы впитывали совершенно различные подходы к программированию. Однако многие из них отталкивались от идей и методов, отработанных при решении проблем, возникавших в программировании задач искусственного интеллекта.

Парадигмы программирования

Часть идей ведет к созданию действительно новых способов мышления и программирования, даже не связаных с конкретным языком. В этом случае говорят

[1)] Действительно, в области программирования можно отметить свойство, аналогичное импринтингу в биологии, которое в программировании, как правило, сдерживает прогресс, играя консервативную роль. Причина этого очень проста. Она заключается в том, что биологическая среда обитания меняется медленно, тогда как идеи программирования быстро сменяют друг друга и усвоенные привычки программирования становятся тормозом. – *Прим. ред.*

о *моделях* (model) или *парадигмах* (paradigm) программирования, которые могут послужить основой самостоятельных языков программирования. Традиционны, например, процедурное и функциональное программирование. Более новые парадигмы – это продукционное программирование, логическое и объектно-ориентированное программирование.



Лисп-машина серии Lambda фирмы LMI.

Не все принятые в Лиспе парадигмы родом из самого Лиспа, несмотря на то что подобные направления использовались в Лиспе уже давно. Например, логическое программирование и Пролог разработаны в процессе исследования доказательства теорем, которое ранее осуществлялось на Лиспе. Идея объектно-ориентированного программирования возникла при обработке свойств символов в Лиспе задолго до того, как языки Симула и Смолтолк появились на свет. Однако в "чистые" парадигмы программирования (или в *метафоры* (metaphora), как это имеет место в языке Смолтолк) они вылились лишь в ходе развития соответствующих языков программирования. Все же возможности, присущие Лиспу, позволили освоить и включить в него результаты других разработок, и часто в более совершенной форме, чем в исходном языке. Например, объектно-ориентированная система Flavor в Зеталиспе во многих отношениях является более развитой, чем Смолтолк.

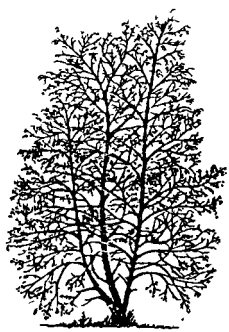
Использование различных парадигм программирования в Лиспе облегчается благодаря многим перенятым Лиспом удачным механизмам и способам записи. К ним относятся, например, развитые механизмы управления,

макросы, макросы чтения, потоки, замыкания и т.д. Кроме того, как мы увидим в дальнейшем, для различных парадигм разработано большое количество различных методов и приемов программирования.

Литература

1. Barr A., Cohen P., Feigenbaum E.(Eds.) *The Handbook of Artificial Intelligence,* Vol 1–3, Pitman, London, 1981–1982.
2. Hayes-Roth F., Waterman D., Lenat D. *Building Expert Systems.* Addison-Wesley, Reading Massachusetts, 1983. [Имеется перевод: Хейес-Рот Ф., Уотерман Д., Ленат Д. Построение экспертных систем. – М.: Мир, 1987.]

3. Hyvönen E. *Asiantutijajärjestelmien tietämystekniikka.* Knowledge Engineering Ky, Helsinki, 1986.
4. Nilsson N. *Principles ot Artifical Intelligence.* Tioga, Palo Alto, California, 1980. [Имеется перевод: Нильсон Н. Принципы искусственного интеллекта. – М.: Мир, 1985.]
5. Pearl J. *Heyristics. Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, London, 1984.



2 МЕТОДЫ ПРОГРАММИРОВАНИЯ

Джон фон Нейман.

Основные методы программирования и соответствующий образ мышления, представленные в первой части книги, отражают специфику классического функционального программирования, которое опирается в основном на лямбда-механизм и рекурсию. Однако Лисп – это один из редких языков программирования, который предлагает в рамках единого формализма возможности для большого разнообразия методов программирования, таких, например, как используемые в Бейсике и Фортране операторное программирование, фразовое программирования Паскаля, объектно-ориентированное программирование Смолтолка и, наконец, логическое программирование Пролога. Эта исключительная многосторонность объясняется хорошей теоретической основой языка, его независимостью от какой-либо конкретной архитектуры и его открытостью, гибкостью и расширяемостью. В этой главе мы рассмотрим наиболее важные модели и методы программирования, используемые в Лиспе.

Стиль – это отражение мышления.

А. Шопенгауэр

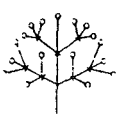
2.1 ОПЕРАТОРНОЕ ПРОГРАММИРОВАНИЕ

- **Функциональное программирование**
- **Операторное и процедурное программирование**
- **Рекурсия или итерация**
- **Рекурсивное операторное программирование**
- **Фразовое программирование**
- **Макропрограммирование**
- **Литература**

Функциональное программирование

Функциональная программа на Лиспе состоит из множества коротких, в общем довольно простых функций. Функции вызывают друг друга в единообразной форме, которая определяется использованием лямбда-механизма. В отображающем вычисления теле лямбда-выражения предусмотрена возможность динамического вызова функциями друг друга или рекурсивного вызова функцией самой себя (непосредственно либо опосредованно).

О передаче параметров и поддержании окружения вычислений заботятся лямбда-механизм и механизм блокировки вычислений. Последовательность вычисления вызовов определяется в основном в соответствии с иерархической структурой вызовов, лямбда-преобразованием и формами, управляющими вычислениями (QUOTE, COND и другие). Порядок следования определений функций в тексте программы не имеет никакого значения с точки зрения логики программы. Порядок вычислений в функциональном программировании непосредственно не выражен. Ход выполнения программы и необходимое для нее время и память проявятся лишь во время ее выполнения.



В главе, посвященной функциональному программированию[1], мы ограничивались в основном лишь использованием тех форм, которые не имеют побочного эффекта (лямбда-вызов, COND, базовые функции и т.д.). Мы, например, не использовали присваивания. Такой редуцированный, ортодоксный Лисп традиционно называют *чистым* Лиспом (pure Lisp). В качестве управляющих структур в чистом Лиспе достаточно иметь блокировку вычислений и условное предложение, с помощью которого осуществляется разветвление вычислений. Последовательность вычислений отображается вложенными вызовами, а повторение – рекурсией. В принципе другие управляющие механизмы, кроме этих, не нужны. Именование с помощью DEFUN не носит обязательного характера потому, что и его можно заменить динамической связью, как это делалось первоначально в теоретическом рассмотрении Лиспа.

Программист, привыкший к программированию с помощью операторов или предложений, будет себя чувствовать неуютно в ситуации, когда отсутствуют операторы присваивания, передачи управления, циклы и другие привычные средства. Эти выразительные средства теоретически не расширяют класс программируемых и решаемых задач, но в некоторых ситуациях они могут быть нужны или даже необходимы. В практическом программировании наряду с функциональным программированием используются и другие методы. Хорошему программисту нужно понимать и знать достоинства и недостатки различных стилей и уметь их использовать.

Операторное и процедурное программирование

Хотя функциональный стиль является стилем программирования на Лиспе, Лисп содержит необходимые средства и для обыкновенного *императивного* или *операторного программирования* (imperative program-

[1] См. том 1. – *Прим. ред.*

ming), которое используется в операторных языках, таких как языки ассемблера, Бейсик и Фортран. Операторное программирование обеспечивается в Лиспе операторами присваивания, условными операторами и операторами передачи управления (SETQ, IF, GO и другие), а также рядом предложений, поддерживающих циклы.

Обычно во всех операторных языках есть некоторый механизм, с помощью которого из последовательно совершаемых операторов можно составить бóльшие целостные единицы, или подпрограммы. Тогда говорят о процедурном программировании (procedural programming). Составные операторы и фрагменты в Лиспе можно строить, например, формами DO и PROG и объявлять их подпрограммами, используя предложение DEFUN. Содержащаяся во многих формах возможность *неявного* progn[1] также является отображением операторного стиля программирования. Для возврата управления программе внешнего уровня используются специальные механизмы, такие как условие окончания формы DO и оператор возврата управления RETURN.

Рекурсия или итерация

Поскольку операторные и функциональные программы по вычислительным возможностям одинаковы, каждое вычисление можно записать любым из этих двух способов. Однако программы, записанные различными способами, по своим свойствам заметно отличаются друг от друга. Поэтому нужно взвешивать, какой способ в какой ситуации лучше.

Факторами, влияющими на решение, могут быть эффективность вычислений в отношении времени или объема памяти, длина программы, ее прозрачность и понятность и т.д. На решение может повлиять, напри-

---

[1] Напомним, что неявный progn означает, что выражение почти всегда можно заменить последовательностью выражений, которые будут вычислятся слева направо, причем результатом вычисления последовательности будет служить значение ее последнего выражения. *–Прим. ред.*

мер, то, насколько используемая вычислительная система обеспечивает обработку различных видов рекурсии. Архитектура большинства современных машин рассчитана в первую очередь на операторное программирование, а преобразование рекурсивных функциональных вычислений в эффективные итерационные вычисления удается не всегда. Кроме того, на выбор могут повлиять привычки и привязанности программиста.

Из рекурсивного строения списков следует, что рекурсивное определение обрабатывающих их функций будет короче и нагляднее отразит суть вычислений. В свою очередь итеративная программа может быть с точки зрения вычислений более эффективной. Это является следствием того, что вычисление рекурсивной функции требует при каждом вызове некоторого учета уровней и значений параметров на различных уровнях. Эта работа пропадает впустую, если запоминаемые данные в вычислениях не участвуют. Однако хорошие трансляторы Лиспа, а часто даже и интерпретаторы способны анализировать код, преобразуя, например, концевую рекурсию в обыкновенную итерацию. В этом случае разницы в эффективности нет.

Рассмотрим в качестве примера преимуществ рекурсии копирование (верхнего уровня) списка, определение которого нам уже знакомо:

```
(defun copy-list1 (l)
   (if (null l) nil
      (cons (car l)
         (copy-list1 (cdr l)))))
```

Те же действия можно записать в операторном стиле, но мы получим более сложную программу на следующей странице.

Простая функция, запрограммированная с помощью операторов присваивания и передачи управления, теряет свою наглядность. Список сначала переворачивается и заносится в переменную V, проходя список до его конца, с тем чтобы затем копию U можно было

```
(defun copy-list2 (l)
 (prog (u v)
   (setq u l)
A  (if (null u) (go B) nil); список нужно
   (setq v (cons (car u) v)); сначала
   (setq u (cdr u))        ; перевернуть,
   (go A)
B  (if (null v) (return u)); чтобы его
   (setq u (cons (car v) u)); можно было
   (setq v (cdr v))        ; снова
   (go B)))                ; построить
```

строить от конца к началу. В этом случае итеративный вариант, кроме того, что он не нагляден, явно еще и менее эффективен, поскольку он строит копируемый список дважды.

COPY-LIST1 представляет особенно простую форму рекурсии – рекурсию по хвостовой части списка. Если бы рекурсия по своему типу была бы сложнее, то и ее реализация в операторном виде оказалась бы много сложнее. Если бы функция была одновременно рекурсивна как по головной части, так и по хвостовой, как функция COPY-TREE, то нам пришлось бы кроме цикла копирования запрограммировать стековый механизм для промежуточного сохранения значений на время копирования подсписков различных уровней.

Преобразование итерации в рекурсию обычно проще, чем наоборот. Это однако нужно редко. Обратное направление преобразования необходимо при трансляции программы в другой язык, не имеющий рекурсии, например при трансляции Лиспа в машинный язык. Если рекурсия сложная, то и задача становится сложной.

Для преобразования рекурсивных вычислений в итеративную форму разработаны специальные методы, на которых мы здесь однако останавливаться не будем.

Рекурсивное операторное программирование

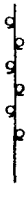
Лучше всего рекурсивные методы оправданы в задачах, в которых встречается формальная и прежде всего естественная рекурсия в структурах и процессах.

Рекурсивные процессы в практическом программировании обычно не являются чисто рекурсивными, а с ними часто связаны различные побочные эффекты. В таких случаях уместно использовать рекурсивное операторное программирование, другими словами компромиссное объединение рекурсивного и операторного программирования.

Например, параллельную рекурсию можно в отношении хвостовой части превратить в итерацию, но сохранить рекурсию в отношении головной части. Такое решение подчеркивает интерпретацию списка в виде упорядоченного множества, к каждому элементу которого рекурсивно применяется некоторое действие. Например, ранее представленное нами обращение списка, т.е. изменение последовательности элементов списка на всех уровнях, мы можем определить таким образом:

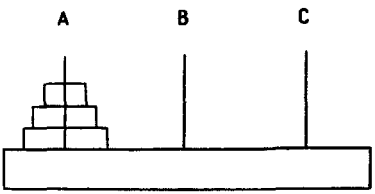
```
(defun ОБРАЩЕНИЕ (l)
   (do ((остатки l (cdr остатки))
        (результат nil
            (cons (ОБРАЩЕНИЕ
                     (car остатки))
                  результат)))
       ((null остатки) результат)))
```

Операторное программирование особенно полезно, если во время рекурсии нужно получить побочные эффекты, например, для записи или печати промежуточных результатов. Реализуем для примера классическую восточную игру Ханойские башни (Towers of Hanoi) в виде рекурсивной операторной программы.



Это древняя восточная игра, в которую играли буддийские монахи, и, возможно, что они ее и придумали. Говорят, что монахи верили, что с помощью этой игры можно вроде бы предсказать

наступление конца света. Конец света наступит, когда закончится игра с шестьюдесятью четырьмя дисками.



Игра состоит в следующем. Используется три вертикальных стержня A, B и C и совокупность N круглых дисков различной величины с отверстием. В начальном состоянии диски нанизаны по порядку в соответствии со своим размером на стержень A. Целью является перенесение всех дисков на стержень B. Однако диски переносятся не в произвольном порядке, при переносе нужно выполнять следующие правила:

1. За один раз можно перенести только один диск.

2. Больший по размеру диск нельзя положить на меньший.

Третий стержень можно использовать как вспомогательный (промежуточный). Если он свободен или там лежит больший диск, то на него можно переложить очередной диск на то время, пока переносится ниже лежащий диск. В этой идее и содержится решение игры. Идею нужно лишь обобщить. Перенесем сначала N–1 дисков на вспомогательный стержень C, чтобы можно было перенести нижний диск на стержень B. После этого мы можем N–1 дисков перенести на место. Мы таким образом разделили задачу на три подзадачи, из которых первая и последняя сводятся к первоначальной, но меньшей на один диск задаче и средняя – к простому переносу диска. В качестве результата мы получим следующие рекурсивные правила:

Алгоритм задачи Ханойские башни:

1. Перенести со стержня A N–1 дисков на вспомогательный стержень C (задача Ханойские башни для N–1).
2. Перенести нижний диск со стержня A на стержень B.
3. Перенести со стержня C N–1 дисков на стержень B (задача Ханойские башни для N–1).

При проведении редуцированных игр можно свободно использовать все стержни, поскольку больший диск не является препятствием для переноса других дисков. Нужно лишь помнить, с какого стержня на какой переносятся диски на текущем уровне, поскольку исходный, вспомогательный и целевой стержни меняются местами. Вследствие рекурсии первоначальная проблема так же, как и все подзадачи сводятся в конце концов к легко решаемой задаче с одним диском.

Хотя интуитивно алгоритм кажется ясным, тем не менее не очевидно, как реализуются переносы и смена ролей стержней на различных уровнях рекурсии. Это, однако, автоматически учитывается механизмом рекурсии. В этом мы можем убедиться, запрограммировав алгоритм так, что он при каждом переносе выводит диск и стержни:

```
_(defun ханойские-башни (высота)
   (progn (перенос 'a 'b 'c высота)
          'готово))
ХАНОЙСКИЕ-БАШНИ
_(defun перенос (из в вспомогательный n)
   (cond
      ((= n 1)                    ; ветвь 2
         (выведи из в)
      (t (перенос из              ; ветвь 1
            вспомогательный
            в
            (- n 1))
```

```
          (выведи из в)
          (перенос вспомогательный ;ветвь 3
             в
             из
             (- n 1)))))
ПЕРЕНОС
_(defun выведи (из в)
   (format t "~S -> ~S~%" из в))
ВЫВЕДИ
```

Задачи для башен, состоящие из одного или двух дисков, решаются просто. Задача для трех дисков требует уже больше переносов:

```
_(ханойские-башни 3)
A -> B
A -> C
B -> C
A -> B
C -> A
C -> B
A -> B
ГОТОВО
```

Легко убедиться в правильности переносов и в том, что решение годится для башни произвольной высоты.

Однако количество переносов для более высоких башен быстро возрастает. Заменив оператор печати счетчиком, мы можем убедиться что для игры десятого уровня нужно уже 1023 переноса. С помощью рекуррентных формул можно показать, что в общем случае игра уровня N требует 2 в степени N–1 переносов. Если мы предположим, что вычислительная машина затрачивает на один перенос одну микросекунду, то конец света наступит через 585 000 лет.

Фразовое программирование

В 70-х годах в литературе по программированию развернулась дискуссия об использовании оператора перехода и о его возможном отрицательном влиянии на привычки программистов и качество программирования (Dijkstra, 1975). В результате дискуссии был выработан новый более строгий стиль программирования, который стали называть *структурным программированием* (structured programming). На разработку метода повлияло и ранее разработанное так называемое *модульное* (modular) программирование, под которым понимали проектирование программ путем разбиения их на более или менее независимые подпрограммы, которыми легко можно воспользоваться из других программ, составляя библиотеки подпрограмм и модулей.

В структурном программировании выделяется значимость структур данных программы и задачи, методов и принципов абстрагирования, расчленения и конструирования операций, а также управляющих структур языка. На место проектирования программ "снизу вверх" (bottom up) пришел новый принцип проектирования "сверху вниз" (top down) и принцип пошагового уточнения (stepwise refinement). Программы нужно было подразделять на логические уровни (levels of abstract machine) и модули (module), между которыми нужно было четко определить интерфейсы. Подробности нужно было "спрятать" (information hiding) на разных уровнях реализации и внутри модулей так, чтобы они не влияли на использование модуля на более высоком уровне.

В соответствии с этими принципами была принята дисциплина использования предложений языка. Было показано, что использование оператора GOTO совершенно не нужно и не желательно. Необходимые действия можно было выполнить подобранными соответствующим образом другими управляющими структурами, например следующими, которым есть соответствие и в Лиспе:

Управляющая структура	Соответствие в Лиспе
Следование:	BEGIN ... END	PROGx, LET и др.
Разветвление:	IF ... THEN ... ELSE ...	IF, COND
Цикл:	WHILE ... DO ...	WHEN, DO, др.

Основные формы предложений для структурного программирования содержатся уже в языке Алгол-60, но они не смогли повлиять на предпочтения программистов потому, что большие производители машин поддерживали в первую очередь Фортран, а не Алгол. Структурное программирование получило распространение лишь в 70-х годах, когда разработанный в университетской среде Паскаль начал широко использоваться. В Паскале принципы и техника структурного программирования были развиты дальше посредством использования новых предложений, таких как REPEAT-UNTIL, CASE-OF и др.

В Лиспе первоначально не было таких более современных форм структурного программирования. Однако позже их включили в язык в качестве его расширения (DO, WHEN, UNLESS, CASE и другие). Фразовое программирование в стиле Алгола и Фортрана хорошо подходит и для Лиспа. И если некоторая форма отсутствует или если изобретена какая-нибудь новая форма, то ее можно легко добавить в язык в виде макроса.

Лисповские формы и функциональное программирование очень хорошо поддерживают модульное и структурное программирование так же, как и пошаговый способ разработки лисповских программ.

Функция в чистом виде является самостоятельным программным модулем. У нее нет побочного эффекта, и ее интерфейс точно определен посредством аргументов и возвращаемого значения.

В Лиспе наряду с традиционными статическими формами предложений используется структурный динамический механизм переходов CATCH-THROW, который позволяет передать управление из одного статического окружения в другое. Такая возможность

отсутствует в традиционных языках с блочной структурой.

Макропрограммирование

Макросистема Лиспа позволяет расширить язык путем использования новых форм и их настройки. Механизм годится и для структурного *макропрограммирования* (macro programming).

В Лиспе макропрограммирование можно применять разными способами, например так, как это принято в системном программировании. С помощью макросов можно определять близкие решаемой задаче языки более высокого уровня и *процессоры* (processor) для них, т.е. интерпретатор и транслятор или конвертор. Процессором является программа, реализующая типы данных и действия абстрактного языка в некотором другом языке. Реализованные с помощью макросов новый язык и его процессор называют *абстрактной моделью* (abstract model) языка.

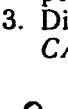
Макросы представляют собой важнейшее средство для достижения, например, машинной независимости и *переносимости* (portability) программ. С их помощью легко определить абстрактный промежуточный язык. С помощью макросов можно, например, определять абстрактные промежуточные языки, которые хорошо подходят для системного программирования и поддерживают подходящим образом различные архитектуры вычислительных машин, не являясь, однако, зависимыми от их деталей и различий.

С макропрограммированием связан важный принцип системного программирования, называемый *раскруткой* (bootstrapping). Буквально этот термин переводится как поднятие себя за голенища сапог или за шнурки ботинок. Однако речь идет не об известном подвиге барона Мюнхаузена. Под раскруткой понимается процедура, в соответствии с которой сначала вручную программируется для выбранной части языка процессор для некоторой вычислительной машины (или для

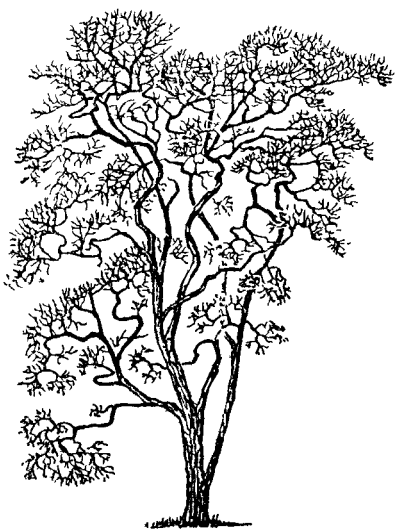
некоторого языка, уже существующего на машине). Полученное таким образом ядро языка затем используется для реализации оставшейся части языка или для его расширения таким же образом. Другими словами, язык используется при построении своего процессора. Раскрутку можно использовать в том числе при определении интерпретатора и транслятора с Лиспа на самом Лиспе или при переносе процессора и вместе с ним всего комплекса программ на другую машину или язык. Эту процедуру можно осуществлять с помощью большого количества этапов и в режиме интерпретации даже непрерывно. Так поступают в практическом программировании на Лиспе, которое можно обычно считать пошаговой раскруткой интерпретатора Лиспа до решающего данную проблему процессора высокого уровня.

Литература

1. Backus J. Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs. *CACM*, Vol 21, No 8, August, 1978, pp. 613–641.
2. Barron D.W. *Recursive Techniques in Programming*. MacDonald, London, 1968. [Имеется перевод: Баррон Д. Рекурсивные методы в программировании. –М.: Мир, 1974.]
3. Dijkstra E.W. Goto Statement Considered Harmful. *CACM*, Vol. 18, 1975.
4. Floyd R.W. The Paradigms of Programming. *CACM*, Vol. 22, 1979, pp. 455–460.
5. Henderson P. *Functional Programming, Application and Implementation*. Prentice-Hall, London, 1980. [Имеется перевод: Хендерсон П. Функциональное программирование. Применение и реализация. –М.: Мир, 1983.]
6. Ingerman P.Z. *A Syntax-Oriented Translator*. Academic Press, New York, 1966.
7. Kernighan B.W., Plauger P.J. Programming Style: Examples and Counter Examples. *ACM Computing Surveys*, Vol. 6, No. 4,1974, pp. 303–319.



8. Knuth D.E. Structured Programming with Goto Statement. *ACM Computing Surveys*, Vol. 6, No. 4, 1974.
9. Rohl J.S. *Recursion via Pascal.* Cambridge Computer Texts 19, Cambridge University Press, Cambridge, England, 1984.
10. Waite W.M. *Implementing Software for Non-Numeric Applications* Prentice-Hall, London, 1973.



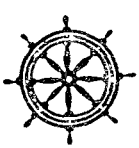
Лишь тот знает, кто действует.
Б. Шоу

2.2 ПРОГРАММИРОВАНИЕ, УПРАВЛЯЕМОЕ ДАННЫМИ

- **Принцип программирования, управляемого данными**
- **Универсальное программирование**
- **Дифференцирование выражений**
- **Язык представления электрических схем**
- **Другие методы программирования, управляемого данными**
- **Программирование, управляемое событиями**
- **Литература**

Принцип программирования, управляемого данными

Метод программирования, в котором внешние к программе данные используются с целью управления работой программы или сами интерпретируются в качестве программы, называется *программированием, управляемым данными* (data driven). В программировании, управляемом данными, программы хранятся вместе с данными или с отображениями их типов. Таким образом, необходимые в текущий момент функции можно определить или найти исходя из обрабатываемых данных.



Предпосылкой использования программирования, управляемого данными, является наличие в языке программных средств для интерпретации во время выполнения программы внешних по отношению к ней выражений данных в качестве программы. Кроме того, в общем случае необходимо уметь строить из данных программы до их выполнения. Программирование, управляемое данными, является как бы мостиком

между действиями внутри самой программы и внешним по отношению к ней миром данных. В традиционных транслируемых языках этих предпосылок нет, в них создание фрагмента программы в процессе ее исполнения и его запуск предполагают совместное транслирование фрагмента вместе с исполняемой программой.

В то же время в Лиспе программирование, управляемое данными, легко применимо в связи с единообразной формой представления данных и программы и возможностью с помощью вызовов функций EVAL, APPLY, FUNCALL и других вычислять выражения, являющиеся данными.

Универсальное программирование

С помощью программирования, управляемого данными, можно создать динамические программы, действия которых определяются в момент вычисления в зависимости от поступающих на обработку данных. При обыкновенном процедурном подходе действия программы определены заранее. На вычисления можно влиять посредством параметров, но в более ограниченном объеме, чем при динамическом создании и интерпретации новых функций.

С помощью динамических программ можно реализовать *универсальные* (generic) действия. Например, функциям, обрабатывающим разные типы данных или выполняющим аналогичные действия, можно давать единообразные универсальные имена и определять необходимые в каждом конкретном случае вычисления на основе обрабатываемых данных или их типов. Таким образом, действия, различающиеся лишь деталями, можно, с точки зрения пользователя, определить независимо от типов данных. В Коммон Лиспе с такими универсальными функциями мы уже знакомились в связи с типами данных и определенными для них обобщенными функциями.

Функции, связанные с различными типами, можно сохранять в качестве атрибута символа в его списке свойств или вместе с отображениями типов, но можно

использовать и другие решения. Программирование, управляемое данными, всегда основывается на некотором ассоциативном механизме, который активизируется и управляется в соответствии с обрабатываемыми данными. Преимуществом такого подхода является независимость программ и их расширяемость на новые типы и ситуации без внесения изменений в другие части программы. Достаточно определить новые функции для данного типа.

Универсальные функции входят в состав и некоторых других фразовых языков (Алгол 68, Фортран 77, Ада и другие). Однако в них типы должны быть известны, по крайней мере в момент трансляции программы. Полностью динамическая работа с типами в Лиспе более гибка. Типы объектов проверяются лишь на этапе вычислений во время применения функции независимо от того, является ли функция интерпретируемой или оттранслированной.

Рассмотрим управляемое данными программирование и технику универсального программирования на двух примерах.

Дифференцирование выражений

В качестве первого примера рассмотрим дифференцирование алгебраических выражений. Для наглядности ограничимся алгебраическими выражениями в следующей форме:

(+ x y) и (* x y)

Сложение и умножение можно свободно комбинировать. Например:

(* (+ a (* a b)) c)

Программируя непосредственно, мы получим следующее определение[1]:

---

[1] Строго говоря, допустимые выражения (д.в.) для приведенной ниже программы дифференцирования по x могут быть определены

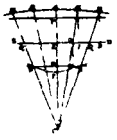
```
;;; производная выражения l по x

(defun дифференцируй (l x)
 (cond
   ((atom l)
      (if (eq l x) 1   ; l = x
          0))          ; l - константа
   ((eq (first l) '+)  ; l = (+ x y)
      (list '+
         (дифференцируй (second l) x)
         (дифференцируй (third l) x)))
   ((eq (first l) '*)  ; l = (* x y)
      (list '+
         (list '*
            (дифференцируй (second l) x)
            (third l))
         (list '*
            (дифференцируй (third l) x)
            (second l))))
   (t l)))
```

```
_(дифференцируй '(+ x (* 3 x)) 'x)
(+ 1 (+ (* 0 x) (* 1 3)))  ; = 1+0+3 = 4
```

Наша программа не умеет упрощать результирующее выражение. Она лишь совершенно механически строит выражение, являющееся производной. (Далее на примерах программирования (система Миксима) мы увидим, как можно сразу же упростить результат. В этом примере нас интересует лишь техника программирования, а не само решение.)

Приведенное выше решение слабо с точки зрения техники программирования, так как, например, его расширение на арифметические действия вычитание и деление или на другие действия требует изменения

рекурсивно следующим образом: x есть д.в., константа есть д.в., сумма и произведение двух д.в. (в префиксной записи) есть д.в. *–Прим.ред.*



определения функции, точнее добавления в предложение COND новых ветвей, соответствующих этим ситуациям. Таким образом, программа не является модульной. Опознание действий сравнением в различных ветвях по порядку друг за другом нам не кажется разумным, особенно в случае большого количества возможностей и в случае, когда используются сложные предикаты. Было бы лучше, если бы необходимое преобразование определялось непосредственно на основании формы дифференцируемого выражения и без сравнений.

Мы получим более подходящее решение, если для каждого действия определим свою дифференцирующую функцию и через имя, например ПРОИЗВОДНАЯ, свяжем в списке свойств определение с символом, обозначающим действие. Таким образом мы упростим и определение самой функции ДИФФЕРЕНЦИРУЙ. Необходимое преобразование мы можем извлечь непосредственно исходя из первого элемента (+ или *) дифференцируемой формы:

```
(defun дифференцируй1 (l x)
  (cond ((atom l)
         (if (eq l x) 1 0))
        (t (funcall
             (get (first l) 'производная)
             (cdr l) x))))
```

Функции дифференцирования, соответствующие сложению и умножению, становятся значением свойства ПРОИЗВОДНАЯ символов + и *:

```
(setf (get '+ 'производная)
      'дифференцируй+)
(setf (get '* 'производная)
      'дифференцируй*)
```

Функция дифференцирования для обрабатываемого в данный момент действия находится теперь в списке свойств символа, обозначающего действие. Основная функция лишь управляет выбором и применением определений, которые в свою очередь рекурсивно вызывают основную функцию. Так, мы получим следующие функции дифференцирования, ориентированные на определенные действия:

```
(defun дифференцируй+ (l x)
  (list '+ (дифференцируй1 (first l) x)
           (дифференцируй1 (second l) x)))

(defun дифференцируй* (l x)
  (list '+
    (list '* (дифференцируй1 (first l) x)
             (second l))
    (list '* (дифференцируй1 (second l) x)
             (first l))))
```

Благодаря модульности решения программу можно очень просто дополнить, например, вычитанием:

```
(setf (get '- 'производная)
      'дифференцируй-)

(defun дифференцируй- (l x)
   (list '- (дифференцируй1 (first l) x)
            (дифференцируй1 (second l) x)))
```

Таким образом, первоначальное управление вычислениями, связанное со структурой программы, мы преобразовали в динамическое управление, основанное на данных. Кроме того, мы добились расширяемости сферы применения функции, не изменяя ее определения. Другими словами, мы обобщили или абстрагировали ход вычислений.

Применяющие функционалы (FUNCALL, APPLY) и принцип управляемого данными программирования –

удобные средства определения интерпретаторов и компиляторов для различных языков программирования и различного представления данных.

Для примера рассмотрим комплекс методов программирования, в котором при решении проблемы дифференцирования используется макропрограммирование. Запрограммируем макрос определения DEFПРОИЗВОДНАЯ, с помощью которого определяются дифференцирующие функции для новых действий:

```
(defmacro defпроизводная
   (действие аргументы тело)

;; присвоить заданное в определении
;; выражение в качестве свойства символа
;; под именем ПРОИЗВОДНАЯ

   `(setf ,(get ',действие 'производная)
          '(lambda ,аргументы ,тело)))
```

Дифференцирующие функции различных действий можно теперь определить следующим образом:

```
(defпроизводная + (l x)
  (list '+ (дифференцируй1 (first l) x)
           (дифференцируй1 (second l) x))

(defпроизводная * (l x)
  (list '+
        (list '*
              (дифференцируй (first l)) x)
              (second l))
        (list '*
              (дифференцируй (second l) x)
              (first l))))
```

```
_(дифференцируй1 '(+ x (* 3 x)) 'x)
(+ 1 (+ (* 0 x) (* 1 3)))
```

Мы определили новую управляющую структуру и средства, используя которые можно задать соответствующие функции и связанное с ними множество абстрактных типов данных (дифференцируемые выражения). Фактически мы создали новый ориентированный на задачу *встроенный* (embedded) в Лисп язык программирования. Лисп очень хорошо подходит для разработки таких ориентированных на некоторое приложение новых языков и их интерпретаторов. Лисп – это язык для реализации языков высокого уровня.

Язык представления электрических схем

Во втором примере управляемого данными программирования составим программу, вычисляющую полное сопротивление электрической схемы переменному току, состоящей из сопротивлений, индуктивностей и конденсаторов. Сначала сконструируем универсальный язык представления электрических схем, проводников и отдельных элементов. Затем запрограммируем интерпретатор этого языка, вычисляющий полное сопротивление сети, представленный записью на этом языке.

Проектирование программы на Лиспе можно разделить на две части. С одной стороны, создается некоторая модель строения решаемой задачи и язык представления данных для нее. С другой стороны, проектируется интерпретатор или транслятор, который способен решать задачи, записываемые на этом языке, или преобразовывать этот промежуточный язык в интерпретируемые формы. Если язык представления содержит сведения о структуре задачи, то представление многих практических задач и связанных с ними данных и знаний идет проще и с точки зрения осуществления вычислений более эффективно. Исходя из единообразия данных и программ, можно представить данные в виде, подходящем для их интерпретации в качестве функций, и использовать данные в управлении вычислением. Таким образом, решение задачи и программирование часто можно значительно упростить.

Для представления электрических схем нам нужны формы записи как самих элементов, так и их последовательного и параллельного соединения.

Отдельные сопротивления, индуктивности и конденсаторы можно естественным образом представить в виде списка, первый элемент которого обозначает тип элемента (соответственно r, l или c), а второй – полное сопротивление (в омах, генри или фарадах). Последовательное соединение двух элементов изображается в виде списка, состоящего из символа ПОСЛЕДОВАТЕЛЬНО, за которым следуют элементы или схемы, образующие последовательное соединение. Аналогично параллельное соединение изображается в виде списка, начинающегося с символа ПАРАЛЛЕЛЬНО.

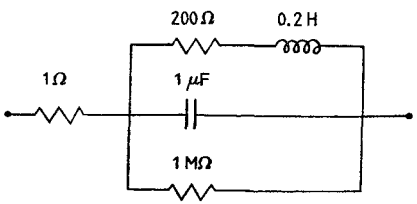


Рис. 2.2.1 Электрическая схема.

Из элементов и их параллельного и последовательного соединений можно составить произвольную, соответствующую этим правилам схему. Например, схему, изображенную на рис. 2.2.1, можно представить так:

```
(setq схема-1
 '(последовательно (r 1)
    (параллельно
      (последовательно (r 100) (l 0.2))
      (параллельно (c 1.0e-6) (r 10e6)))))
```

Исходя из этих предположений, мы определили язык описания задачи. Предложения языка однозначно соответствуют электрическим схемам. Благодаря разумно выбранному строению (синтаксису) языка для

него можно легко построить семантический интерпретатор, который определяет смысл элементов различных типов и их соединений в различных частях схемы. В данном приложении в качестве "смысла" структур рассматривается величина полного сопротивления схемы. (Для этого же языка представления можно определить и другую интерпретацию. Например, графическая интерпретация могла бы состоять в вычерчивании схемы на экране дисплея.)

Если схема состоит из одного элемента, то ее полное сопротивление можно легко вычислить, определив в качестве функций формы, представляющие элементы:

```
(defun r (res)
   (list res 0))

(defun l (ind)
   (list 0 (* *omega* ind)))

(defun c (cap)
   (list 0 (/ -1 (* *omega* cap))))
```

Значением функции будет комплексное число (пара чисел). Первое число пары отображает действительную часть полного сопротивления и второе – мнимую часть. (Мы могли бы использовать и комплексную арифметику, имеющуюся в Коммон Лиспе.) Угловая частота переменного тока изображается с помощью свободной динамической переменной *OMEGA*:

```
(defvar *omega*)  ; динамическая *omega*
```

Обработку последовательного соединения можно представить в виде:

```
;;;   z = z1 + z2 + ...+ zN

(defun последовательно (&rest импедансы)
   (apply 'кплюс импедансы))
```

Здесь КПЛЮС – действие сложения для переменного количества комплексных чисел:

```
(defun кплюс (кчисла)
   (вычисли-сумму 0 0
      (first кчисла) (rest кчисла)))

(defun вычисли-сумму
      (r-часть i-часть число остаток)
   (if (null число)
      (list r-часть i-часть)
      (вычисли-сумму
            (+ r-часть (first число))
            (+ i-часть (second число))
            (first остаток)
            (rest остаток))))
```

Аналогичным образом запрограммируем обработку параллельного соединения:

$$z = 1/(1/z1 + 1/z2 + \ldots + 1/zN)$$

Для этого кроме функции КПЛЮС нам потребуется функция ОЧИСЛО, которая вычисляет обратную величину комплексного числа:

$$1/(a + ib) = (a - ib)/(a^2 + b^2)$$
$$= a/(a^2 + b^2) - ib/(a^2 + b^2)$$

На основе этих функций можно определить функцию:

```
(defun очисло (z)
 (let* ((a (first z))    ; LET не подходит
        (b (second z))
        (сумма (+ (expt a 2) (expt b 2))))
      (list (/ a сумма)
         (* b (/ -1 сумма)))))
```

Обработка параллельного соединения теперь представляется в виде

```
(defun параллельно (&rest импедансы)
  (очисло (apply 'кплюс
                 (mapcar 'очисло импедансы))))
```

После таких определений наш язык описания компонент и цепей можно непосредственно интерпретировать с помощью интерпретатора Лиспа, который в качестве значения выдаст полное сопротивление электрической схемы. В данном случае нет необходимости разрабатывать собственный интерпретатор:

```
(defun импеданс (схема *omega*)
   (eval схема))
```

С помощью интерпретатора мы можем, экспериментируя с частотой, поискать собственную частоту цепи (т.е. частоту, при которой полное сопротивление состоит лишь из действительной части):

```
_(импеданс схема-1 100) ; значительно
(101.39016 19.033777)   ; отличается
_(импеданс схема-1 2179.45); близко к соб-
(2000.601074 -0.003724) ; ственной частоте
```

Составить соответствующую программу на каком-нибудь операторном языке было бы не так просто. Для представления данных, работы с ними и их тестирования, а также для управления памятью потребовалось бы множество вспомогательных функций. Интерпретатор Лиспа и сам язык уже содержат эти возможности, поэтому проектирование программы можно начать непосредственно с уровня символьного языка, близкого решаемой задаче.

Другие методы программирования, управляемого данными

Метод программирования, управляемого данными, особенно удобен в практических областях, где существует большое многообразие разнородных но аналогичных типов данных и иерархий типов. В качестве примера можно упомянуть обработку естественного

языка или изображений, а также специальные задачи моделирования реального мира.

На программировании, управляемом данными, основаны некоторые специальные методы и формализмы, например используемые в разборе предложений *сети перехода состояний* (transition network), или так называемые языки ATN (augmented transition network grammar) и *классификационные сети* (discrimination net).

Сеть ATN – это формализм, в котором алгоритм анализа управляется специальным описанием грамматики в виде сети. Использование правил грамматики, переход между состояниями и применяемые при этом функции определяются на основании анализируемого предложения.

В свою очередь, классификационные сети – это метод классификации и выбора, в котором данные в соответствии с описанием сети подвергаются последовательным тестам, выявляющим свойства и другие характеристики данных и управляющим их обработкой. Классификационные сети используются, например, при формировании предложений, особенно при выборе слов на основе грамматических и семантических ограничений, а также в специальных приложениях, связанных с классификацией и обучением.

Программирование, управляемое событиями

Программированию, управляемому данными, в некотором смысле противоположным будет *программирование, управляемое событиями* (event/action driven).

Под вычислениями, управляемыми событиями, понимают такой стиль вычислений, в котором функции или другие действия не вызываются непосредственно или на основе интерпретации данных, а активируются в определенных ситуациях. Ситуацией может, например, быть изменение значения переменной или поля структуры, вызов функции и т.п., или это может быть внешнее по отношению к программе событие, о котором узнает программа.

Функцию или действия, запускаемые в результате возникновения ситуации, называют *демоном* (daemon). Название происходит от того, что демон как бы ждет в засаде, когда произойдет нечто, после чего он начинает осуществлять свои намерения.

Литература

1. Charniak E., Riesbeck C., McDermott D. *Artificial Intelligence Programming*. L. Erlbaum, Hillsdale, New Jersey, 1980.
2. Stoyan H., Goerz G. *Lisp – Eine Einführung in die Programmierung*. Springer-Verlag, Berlin, 1984.
3. Woods W.A. *Transition Network Grammars for Natural Language Analysis*. *CACM*, Vol. 13, No. 10, 1970.

Лоф[1] сам выбирает себе крышку.

Финская пословица

2.3 СОПОСТАВЛЕНИЕ С ОБРАЗЦОМ

- **Сопоставление с образцом и распознавание образов**
- **Распознавание списочных образов**
- **Условия сопоставимости**
- **Использование переменных в образце**
- **Сопоставление с переменной образца**
- **Предикатный образец ограничивает сопоставимость**
- **Компьютерный психиатр ELIZA**
- **Распознавание структур**
- **Литература**

Сопоставление с образцом и распознавание образов

Под *сопоставлением с образцом* (pattern matching) понимается процедура, при которой с известной символьной структурой, или *образцом* (pattern, template), сопоставляется некоторая другая структура, или *образ* (pattern), с целью выявления единообразия или подобия структур или для выявления условий этого. Этот метод называют также *распознаванием образов* (pattern recognition), когда хотят подчеркнуть в процессе сторону идентификации образа, а не сопоставления с образцом. Обратите внимание, что в английском языке слово "pattern" обозначает как "образец", так и "образ".

Огонь.

В этой главе мы рассмотрим символьное и структурное сопоставление с образцом сначала с точки зрения уяснения принципов и используемой нотации,

---

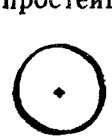
[1] Лоф – древний деревянный сосуд, служивший также и мерой сыпучих тел. – *Прим. перев.*

а затем с точки зрения примеров программирования. Построим лисповские функции, отвечающие различным условиям сопоставимости, для распознавания одноуровнего образа и реализуем с их помощью общее распознавание списочных структур. Наконец построим классический пример программы ELIZA, поддерживающей диалог на естественном языке, в основе которой лежит распознавание образов.

Сопоставление с образцом создает основу и для рассматриваемых далее *продукционного программирования* (rule based programming) и *логического программирования* (logic programming). Продукционное и логическое программирование являются важными методами, используемыми в экспертных системах, обработке естественных языков и других применениях программ искусственного интеллекта.

Распознавание списочных образов

Предикаты сравнения Лиспа, например EQ, EQL, EQUAL, MEMBER и др., уже фактически являются простейшими функциями сопоставления с образцом. Условия совпадения для этих предикатов все же очень жесткие, хотя они и отличаются друг от друга. Если мы хотим использовать более свободные условия сопоставления, то их нужно определять самому. Как мы увидим, это сделать довольно просто. Для начала ограничимся сопоставлением со списочным одноуровневым образцом. Сопоставление со списком можно с помощью рекурсии обобщить для многоуровневых списочных структур.



Воздух.

Условия сопоставимости

К сопоставлению элементов образца и образа можно предъявлять различные требования. Во-первых, можно потребовать, чтобы определенные элементы были идентичны. Для более гибких требований мы будем использовать символы-заменители, для которых определены условия сопоставимости. Можно различить следующие основные случаи:

ЗАМЕНИТЕЛЬ

? произвольный символ образа
\+ непустая последовательность символов образа, или сегмент
* сегмент или пустая последовательность

Мы хотим определить функцию СОПОСТАВЬ, которая проверяет, подходит ли данный образ к заданному с использованием заменителей образцу:

(СОПОСТАВЬ *образец образ*)

```
_(сопоставь '(? ? к ?) '(ф а к т))
T
_(сопоставь '(+ к т) '(о б ъ е к т))
T
_(сопоставь '(* т р * а к т)
            '(а н т р а к т))
T
_(сопоставь '(? с *) '((a b) c d ((e))))
T
```

Образцы сопоставляются только на самом внешнем уровне списков.

Идея следующего определения функции СОПОСТАВЬ состоит в том, что образец и образ рассматриваются символ за символом и проверяется выполнение условий. Когда встречается заменитель, то используя методы управляемого данными программирования применяется соответствующая функция сопоставления:

```
(defun сопоставь (m h)
  (cond ((null m)     ; конец образца
         (null h))
        ((null h) nil) ; конец образа
        ((equal (car m)(car h))
```

```
        ;; совпадающий элемент
        (сопоставь (cdr m) (cdr h)))
      ((and (atom (car m)) ; сопоставление
            (get (car m) 'сопоставитель))
        (funcall           ; с образцом
          (get (car m) 'сопоставитель)
          m h))
      (t nil)))            ; неудача
```

Для определения соответствующих заменителям функций сопоставления, или сопоставителей, мы сначала, так же как и ранее в примере дифференцирования, определим макрос определения:

```
;;; Макрос определения сопоставителей

(defmacro defсопоставитель
          (символ парам тело)
  `(setf (get ',символ 'сопоставитель)
     '(lambda ,парам ,тело)))
```

Сопоставителями будут:

```
(defсопоставитель ? (m h)
  (сопоставь (cdr m) (cdr h)))

(defсопоставитель + (m h)
  (or (сопоставь (cdr m) (cdr h))
      (сопоставь m (cdr h))))

(defсопоставитель * (m h)
  (or (сопоставь (cdr m) (cdr h))
      (сопоставь (cdr m) h))
      (сопоставь m (cdr h))))
```

При необходимости выбор условий сопоставления можно расширить. Например заменитель "–", которому соответствует в образце один символ или пустая последовательность, можно было бы реализовать следующим образом:

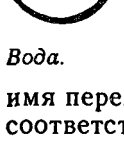
```
(defсопоставитель - (m h)
   (or (сопоставь (cdr m) (cdr h))
      (сопоставь (cdr m) h)))
```

```
(сопоставь '(? ? - к - -) '(т а к т)
T
```

Механизм распознавания можно уточнять, используя комбинацию условий. Можно, например, потребовать, что лишь определенные символы могут быть добавлены или допустить их отсутствие и т.д.

Использование переменных в образце

Сопоставление можно сделать значительно более разносторонним, используя в образце переменные различных типов. Переменная образца получает в качестве значения некоторую часть образа в том случае, когда сопоставление соответствующего типа удалось. Переменные образца будем изображать в виде двухэлементного списка, первый элемент которого отображает тип сопоставления, а второй – имя переменной. Мы будем использовать переменные, соответствующие типам ?, + и *.



Вода.

ПЕРЕМЕННАЯ ОБРАЗЦА	СОПОСТАВЛЯЕТСЯ С
(?> *переменная*)	произвольным символом
(+> *переменная*)	непустым сегментом символов
(*> *переменная*)	пустым или непустым сегментом символов

Теперь напишем новый вариант функции СОПОСТАВЬ1. При успешном сопоставлении она в качестве результата будет возвращать список пар, отображающий значения, связанные с переменными. Если в образце нет переменных и сопоставление прошло успешно, то возвращается Т. Например:

```
_(сопоставь1 '((?> x) или (+> y))
             '(до или после полудня))
((X . ДО) (Y ПОСЛЕ ПОЛУДНЯ))

_(сопоставь1 '(? или *)
             '(до или после полудня))
T
```

Для функции сопоставления СОПОСТАВЬ1 нужен дополнительный параметр (ПАРЫ), который содержит соответствия между переменными и частями образа в виде списка пар:

```
(defun сопоставь1
       (m h &optional (пары nil))
 (cond
   ((null m)
      (if (null h) (if пары пары t) nil))
   ((null h) nil)
   ((equal (car m) (car h))
      (сопоставь1 (cdr m) (cdr h) пары))
   ((atom (car m))
      (if (get (car m) 'сопоставитель))
         (funcall
            (get (car m) 'сопоставитель)
            m h пары)
         nil))
   (t (funcall
         ;; сопоставление с переменной
         (get (first (car m))
                  'сопоставитель)
         m                    ; образец
         h                    ; образ
         (second (car m))     ; переменная
         пары))))             ; соответствия
```

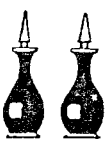
Сопоставителями, как и ранее, будут

```
(defсопоставитель ? (m h пары)
   (сопоставь1 (cdr m) (cdr h) пары))

(defсопоставитель + (m h пары)
   (or (сопоставь1 (cdr m) (cdr h) пары)
      (сопоставь1 m (cdr h) пары)))

(defсопоставитель * (m h пары)
   (or (сопоставь1 (cdr m) (cdr h) пары)
      (сопоставь1 (cdr m) h) пары)
      (сопоставь1 m (cdr h) пары)))
```

Аналогично можно определить типы сопоставления переменных ?>, +> и *>:

```
(defсопоставитель ?> (m h v пары)
   (сопоставь1 (cdr m) (cdr h)
         (acons v (car h) пары)))

(defсопоставитель +> (m h v пары)
   (or (сопоставь1 (cdr m) (cdr h)
               (добавь v (car h) пары))
      (сопоставь1 m (cdr h)
               (добавь v (car h) пары))))

(defсопоставитель *> (m h v пары)
   (or (сопоставь1 (cdr m) (cdr h)
               (добавь v (car h) пары))
      (сопоставь1 m (cdr h)
               (добавь v (car h) пары))
      (сопоставь1 (cdr m) h пары)))

;;; ДОБАВЬ присваивает переменной или
;;; обновляет старую связь

(defun добавь (имя значение пары)
 (cond
   ((null пары)
      (acons имя значение nil))
```

```
((eql имя (caar пары)
   (if (atom (cdar пары))
      (acons имя
         (list (cdar пары) значение)
         (cdr пары))
      (acons имя
         (append (cdar пары)
                 (list значение))
         (cdr пары))))
(t (cons (car пары)
      (добавь имя значение
                 (cdr пары))))))
```



Соответствия элементов образца собираются в параметре ПАРЫ функции ДОБАВЬ, который в случае успешного сопоставления возвращается в качестве результата функции СОПОСТАВЬ1.

Сопоставление с переменной образца

Значение, связанное с переменной, можно в дальнейшем использовать в процессе сопоставления. Например, было бы естественным потребовать, чтобы все встречающиеся в образце переменные с одинаковым именем сопоставлялись бы с одним и тем же значением. Чтобы иметь возможность сослаться на значение ранее сопоставленной переменной, мы будем использовать еще одну форму

Земля.

(< *переменная*)

которая сопоставляется лишь с той частью образа, с которой эта переменная уже сопоставлена. Например:

```
_(сопоставь1 '(? (?> x) (*> y) (< x))
        '(символ ТОТ-ЖЕ должен быть ТОТ-ЖЕ))
((X . ТОТ-ЖЕ) (У ДОЛЖЕН БЫТЬ))
```

Направленный справа налево уголок скобки < напоминает о том, что переменная уже ранее сопоставлена.

Форма < обрабатывается так, что значение переменной берется из списка пар и заменяет в образце выражение с <, и с полученным таким образом образцом вновь пытаются выполнить сопоставление:

```
(defсопоставитель < (m h v пары)
   (сопоставь1 (cons (значение v пары)
                     (cdr m)) h пары))

(defun значение (имя пары)
   (cdr (assoc имя пары)))
```

Предикатный образец ограничивает сопоставимость

Сопоставители можно развивать дальше, используя форму, дающую возможность наряду с заменителями и переменными применять предикатный образец. Предикатное условие можно задать в образце, например, в следующем виде:

(p? *предикат*)

Здесь предикат – функция с одним аргументом, которая получает в качестве аргумента текущий элемент образа. Если предикат истинен, то считается, что сопоставление выполнено. Этим методом можно пользоваться для ограничения сопоставимых с образцом элементов так, чтобы они принадлежали классу, задаваемому предикатом.

Сопоставитель, соответствующий такому предикатному образцу, определяется следующей формой:

```
(defсопоставитель p? (m h пред пары)
   (if (funcall пред (car h))
      (сопоставь1 (cdr m) (cdr h) пары)
      nil))
```

```
_(сопоставь1 '(? ? (p? numberp) +)
             '(a b 3 c))
T
```

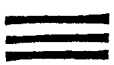
Сопоставление с предикатом можно разнообразить, сочетая его с использованием переменных или с сегментным сопоставлением. С помощью предиката можно было бы учесть правый и левый *контекст* (context) сопоставляемого элемента. Правый контекст образца был бы непосредственно доступен через параметр H. Для исследования левого контекста можно было бы завести вспомогательную переменную сопоставления, которая менялась бы в зависимости от продвижения сопоставления. Взаимозависимости играют важную роль, например, в работе с естественным языком. Их можно учесть в образце с помощью специальных предикатов.

Компьютерный психиатр ELIZA

Используя рассмотренный ранее метод распознавания образов, мы можем запрограммировать простой автомат, поддерживающий беседу на естественном языке, который на самом деле "не понимает" язык, но тем не менее пытается вести весьма содержательную беседу. Программа основана на множестве правил вида:

(*образец вопрос*)

Система пытается сопоставить введенное предложение с образцом какого-нибудь правила, и если это удается, задает вопрос из этого правила. Получив от пользователя новый ответ, к нему снова пытаются применить правило и т.д. Когда нельзя применить ни одно правило, система задает случайный вспомогательный вопрос.



Спокойное состояние.

Возвышенное состояние.

Наречем нашу собеседницу ELIZA в честь ее известного предка[1]. ELIZA беседует на английском языке и моделирует работу психиатра. Применение метода к финскому языку потребовало бы распознавания и продуцирования форм склонения и спряжения слов.

```
(defun eliza (&aux bindings)
 (print '(i am eliza.
          what is your trouble?))
 (terpri)
 (do*
   ((clause (read) (read)))
   ((null clause)
       '(bye bye. wellcome again))
   (do*
      ((rules *rules* (cdr rules))
       (rule (car rules) (car rules)))
      ((cond
         ((null rules)
            (print (help-question))
            (terpri) t)
         ((setq bindings
            (сопоставьl (car rule)
                        clause))
            (print (answer (second rule)
                           bindings))
            (terpri) t))))))

;;; Вспомогательный вопрос задается, когда
;;; ни одно правило не подошло

(defvar *help-questions*
 '((what do you mean?)
   (and so?)
   (could you express yourself more
   clearly?)
```

[1] Имеется в виду программа Вайзенбаума ELIZA. *–Прим. ред.*

```
    ...
    (have we already talked about your
    dreams?)))

;;; HELP-QUESTION возвращает случайный
;;; вспомогательный вопрос

(defun help-question ()
   (nth (random (length *help-question*))
        *help-questions*))

;;; Правила

(defvar *rules*
 '(((* i cannot (+> x))     ; образец
      ;; ответный вопрос
      (why cannot you (< x) ?))
   ((no)
      (please, don't be so short))
   ((yes, +)
      (i understand, but why are you so
      sure?))
   ((* i have a (?> x) *)
      (tell me more about your (< x)))
   ((* you are (+> x))
      (what makes you think that i am (<
      x)))
   ...
   ((+ mother +)
      (have you had problems with your
      family))))

;;; Формирование ответа

(defun answer (model pairs)
 (cond
   ((null model) nil)
   ((atom (car model))
      (cons (car model)
            (answer (cdr model) pairs)))
```

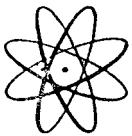
```
    ((eq (caar model) '<)
       (cons (значение (cadar model) pairs)
             (answer (cdr model) pairs)))
    (t (cons (car model)
             (answer (cdr model) pairs)))))

_(eliza)
(I AM ELIZA. WHAT IS YOUR TROUBLE?)
(i cannot enjoy Fortran programming any
more)
(WHY CANNOT YOU ENJOY FORTRAN PROGRAMMING
ANY MORE)
(because of this book)
(WHAT DO YOU MEAN?)
(guess yourself!!!)
(HAVE WE ALREADY TALKED ABOUT YOUR
DREAMS?)
(you are crazy)
(WHAT MAKES YOU THINK THAT I AM CRAZY ?)
...
```

Возбужденное состояние.

Классическими примерами использования метода сопоставления с образцом могут служить: программа PARRY (Colby 1976), которая моделирует пациента больного паранойей и подходит в качестве собеседника для психиатра ELIZA, предложенного Вайзенбаумом; программа STUDENT, написанная Боброу (1962), которая решает алгебраические задачи, задаваемые словесным описанием; а также система информационного поиска SIR, разработанная Рафаэлом (1967).

Распознавание структур

Ранее мы использовали сопоставление с образцом лишь для распознавания одноуровневых списков. Однако этот метод можно легко обобщить для распознавания структур с произвольным уровнем вложенности, применяя распознавание рекурсивно над элементами списка.



Для распознавания более сложных классов образов (языка) можно использовать методы анализа *структурного распознавания образов* (structural pattern recognition). С их помощью можно кроме линейных последовательных языков распознавать более общие символьные структуры *древовидных* и *сетевых языков* (tree, graph language).

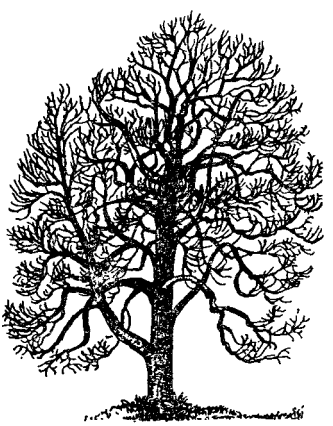
В анализаторе распознаваемый язык описывается и распознается с помощью правил *грамматики* (grammar), которая часто задается в виде образцов. С помощью правил можно гибко определить довольно богатые языки, например естественный язык. В распознавании и обработке структур таких языков обычно не достаточно одних лишь методов сопоставления с образцом, поскольку существует слишком много различных языковых структур и возможностей их комбинирования. В результате анализа получается, как правило, структурное отображение – обычно *дерево разбора* (parse tree) – распознаваемого образа.

Литература

1. Bobrow D.G. A Question Answerer for Algebra Word Problems. Memo No. 45, AI Project, MIT, Cambridge, Massachusetts, 1962.

2. Bundy A. *Catalogue of Artificial Intelligence Tools.* Springer-Verlag, Berlin, 1984.
3. Colby K. *Artificial paranoia. A Computer Simulation of Paranoid Processes.* Elmsford, Pergamon, New York, 1976.
4. Emanuelsson P. *Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation.* Dissertation, Linköping,1980.
5. Raphael B. *SIR, a Computer Program for Semantic Information Retrieval.* in: Minsky M. (ed.): *Semantic Information Processing.* MIT Press, Cambridge, Massachusetts, 1967.
6. Raulefs P. et al. A Short Survey of the State of Art in Pattern Matching and Unification Problems. *SIGSAM Bulletin,* Vol. 13, 1978.

7. Stoyan H., Görz G. *Lisp, Eine Einfåhrung in die Programmierung*. Springer-Verlag, berlin, 1984.
8. Weizenbaum J. ELIZA – A Computer Program for the Study of Natural Language Communication between Man and Machine. *CACM*, Vol. 9, No. 1, 1965.



Нет правил без исключений.

Финская пословица

Исключение – это правило, относящееся к единственному случаю.

Автор неизвестен

2.4 ПРОДУКЦИОННОЕ ПРОГРАММИРОВАНИЕ

- **Продукция = условие + следствие**
- **Интерпретатор продукций применяет продукции**
- **Полный перебор**
- **Аннулирование выбора**
- **Направление поиска**
- **Порядок перебора альтернатив**
- **Программирование методов поиска**
- **Поиск в глубину, в ширину и по наилучшему варианту**
- **Применения продукционного программирования**
- **Литература**

Основой используемых в построении экспертных систем методов представления данных и решения проблем является обычно продукционное программирование. Этот метод программирования часто используется совместно с методами распознавания образов. Далее мы рассмотрим основные понятия продукционного программирования, а также используемые в нем методы поиска.

Продукция = условие + следствие

Под *продукцией* (rule), или правилом, понимается объект, состоящий из двух частей: *условия* (antecedent, condition) и *следствия* (consequent, conclusion):

ЕСЛИ условие ТО следствие

Условие продукции часто представляется с помощью образца, применяемого к данным. Когда данные сопоставимы с образцом, выполняется следствие продукции. Следствие часто представляет собой *действие* (action), изображаемое процедурой.

В грамматике правила используются для продуцирования предложений или, наоборот, для их распознавания или анализа (parse). В таком случае говорят о правилах *трансформации* (преобразования) (transformation) и о *порождающих правилах* (production rule), в которых левая часть есть условие продукции, а правая – следствие продукции.

Интерпретатор продукций применяет продукции

Продукции *применяются* (apply) всегда в отношении каких-нибудь данных. Изучаемые и изменяемые посредством правил данные часто называют *базой данных* (database), содержимым рабочей памяти или другим подобным образом. Условием применимости продукции является то, что *состояние* (situation, state) базы данных содержит в момент применения образ, который сопоставим с образцом, находящимся в условии некоторой продукции.

Применение продукций к базе данных осуществляет *интерпретатор продукций* (rule interpreter), который часто в связи с экспертными системами называют *машиной вывода* (inference engine) и в связи с обработкой языка – *распознавателем* (recognizer) или *анализатором* (parser). Если интерпретатор находит применимую продукцию, то он выполняет действия, записанные в следствии продукции.

Интерпретатор, руководствуясь методами программирования, управляемого данными, выполняет следствия из продукций, опираясь на данные из базы и образцы.

Вообще-то в результате работы продукции состояние базы данных изменяется, т.е. в нее добавляются или из нее удаляются данные. Переход базы данных в новое состояние меняет множество применимых

продукций. Между продукциями (в общем виде) нет прямых зависимостей, как, например в случае подпрограмм, а взаимодействие между продукциями осуществляется через общую для них базу данных.

Продукции переводят базу данных из одного состояния в другое. Состояния, достижимые с помощью продукций из некоторого начального состояния, и переходы между ними образуют пространство поиска, *узлами* (node) которого являются различные состояния базы данных, а соединяющими их направленными *дугами* (arc, edge) – соответствующие продукциям переходы. Таким образом, продукционное программирование и решение проблемы сводится к поиску, в котором через применение продукций ищется путь в пространстве поиска между начальным и конечным состоянием. Найденный путь называют *решением* (solution) проблемы.

Полный перебор

В области искусственного интеллекта уже с 50-х годов интенсивно исследуются различные методы поиска.

Основным типом поиска является *полный перебор* (exhaustive/blind search), в котором решение проблемы (последовательность операций) ищется систематическим опробованием всех вариантов. Если решение существует, то с помощью полного перебора его всегда можно найти.

Существует много способов полного перебора. Важными свойствами, характеризующими поиск и во многом не зависимыми друг от друга свойствами являются следующие:

- способ аннулирования выбора,
- направление поиска,
- порядок исследования альтернатив.

Аннулирование выбора

Поиск основывается на методе проб и ошибок. Может оказаться, что решение проблемы ищется на пути, который в начале выглядел многообещающим, но позже выявилась его бесплодность. При попадании в ошибочную ситуацию, перед возвратом к предшествующим состояниям и пробой новых вариантов решения, аннулируются ранее сделанные решения и заключения. Проблема решается *динамически* (dynamic) на основе собранных и (гипотетически) выявленных за время предыдущей работы данных и с помощью общих стратегий, правил и знаний. Таким образом, программа справляется с новыми проблемами и ситуациями, которые программист при создании системы учитывал лишь на уровне принципов.

Проще всего принятые решения можно исправлять и изменять посредством *хронологического механизма возвратов* (chronological backtracking). В этом случае при попадании в тупик всегда происходит возврат к последней по времени ситуации выбора и осуществляется новый выбор. Недостаток метода заключается в том, что при возврате и отмене неверного решения отменяются и правильные решения, и их приходится вновь принимать позже. Тупик не обязательно возникает в результате операции, непосредственно предшествовавшей тупику.

Более развитым является метод, в котором поддерживаются данные о применяемых в процессе поиска продукциях, о сделанных в связи с ними предположениях и выводах (dependency record) и в которой в случае необходимости по цепочке зависимостей отменяются лишь ошибочные данные. Такой способ отмены данных называют *механизмом возвратов по зависимостям* (dependency directed/relevant backtracking).

Направление поиска

Задачу можно решать, начиная из начального состояния, из конечного состояния (гипотезы) или исходя из них обоих одновременно:

Прямой поиск (forward chaining, data driven search). В процессе прямого поиска из начального состояния генерируются все новые состояния до тех пор, пока не будет достигнуто конечное состояние.

Обратный поиск (backward chaining, goal driven search). При обратном поиске исходят из конечного состояния (или из предполагаемого решения задачи) и пытаются обосновать сделанные гипотезы подбором начального состояния, используя операторы, направленные в обратную сторону.

Двунаправленный поиск (bi-directional search). Он объединяет движение в прямом направлении от исходного состояния и в обратном направлении от конечного состояния и пытается достичь некоторого общего для них промежуточного состояния.

Выбор направления поиска зависит от характера пространства поиска решаемой проблемы. Если начальных состояний мало, а конечных много, то часто используется прямой поиск. Если же конечных состояний мало, а начальных много, то использование обратного поиска обычно более естественно и эффективно. С помощью двунаправленного поиска можно иногда объединить достоинства прямого и обратного поисков. Некоторые экспериментальные системы сами умеют на основе метазнаний оценить и выбрать подходящую для данной ситуации стратегию.

Порядок перебора альтернатив

Способ поиска характеризуется и тем, в каком порядке операторами генерируются новые альтернативные состояния и модели решения (conflict resolution).

Основными альтернативами являются *поиск в глубину* (depth-first) и *поиск в ширину* (breadth-first). Оба способа при полном переборе приводят к логически одинаковым результатам, но предполагаемое время

поиска и объем используемой памяти отличаются для различных способов в зависимости от характера пространства поиска. Факторами, влияющими на выбор, являются предполагаемое расстояние в пространстве поиска между начальными и конечными состояниями и разветвленность пространства поиска.

Часто разумность различных альтернатив можно оценить заранее. Это приводит к поиску специального типа – "по наилучшему варианту" (best-first). Критерии оценки различных решений меняются в зависимости от типа проблемы и области применения, и поэтому мы не будем разбирать их подробнее.

Более основательно поиск рассматривается, например, в работах (Nilsson 1980) и (Pearl 1984).

Программирование методов поиска

Рассмотрим поиск решения проблемы на основе продукций на простом примере. Рис. 2.4.1 изображает множество городов и имеющихся между ними морских коммуникаций. Наша цель состоит в написании программы, которая ищет маршруты между двумя произвольными городами.

Задача приводит к следующему пространству поиска:

- Состояниями пространства являются города.
- Операторами являются продукции, отображающие коммуникации. Таким образом, операторы отображают переход из одного состояния в другое.
- Пространством поиска является сеть, состоящая из различных городов и коммуникаций между ними.
- Состояния пространства поиска можно представить просто в виде названия города. Продукции можно изобразить в виде пары состояний:

  (*начальный-город конечный-город*)

  Или в словесном виде:

"Если находишься в начальном городе X,
то можешь переместиться в конечный город Y".

Рис. 2.4.1 Города и коммуникации между ними.

Продукции, соответствующие пространству поиска, изображенному на рисунке, выглядели бы так:

```
(setq *продукции*
   '((хельсинки стокгольм)
   (хельсинки таллинн)
   (хельсинки ленинград)
   (стокгольм хельсинки)
   (таллинн хельсинки)
   (ленинград выборг)
   (выборг хельсинки)))
```

Функция МАРШРУТ ищет путь между двумя городами:

```
(defun маршрут (начало конец порядок)
   (поиск конец
      (применимы начало)
      (list начало)
      порядок))
```

Функция ПРИМЕНИМЫ возвращает продукции, применимые в данной ситуации:

```
(defun применимы (состояние)
   (mapcan #'(lambda (правило)
      (if (eq состояние (car правило))
         (list правило) nil))
      *продукции*))
```

Рассматриваемая далее функция ПОИСК в параметре ПЛАН содержит потенциально применимые в текущий момент продукции. Искомое конечное состояние сохраняется в параметре КОНЕЦ, а маршрут строится в параметре РЕШЕНИЕ. Если конечный город первого правила из списка ПЛАН совпадает с конечным состоянием, то поиск удался и шаг за шагом сформированный в параметре РЕШЕНИЕ маршрут из состояний можно выдать в качестве решения. Однако список нужно сначала перевернуть, поскольку он строился в обратном порядке. Функция РЕШЕНИЕ состоит из исследованных функцией ПОИСК городов в порядке их прохождения и необязательно отражает маршруты между городами.

```
(defun поиск (конец план решение порядок)
 (cond
   ((null план) nil)
   ((eq конец (second (car план)))
      (reverse (cons конец решение)))
   ((member (second (car план)) решение)
      (поиск конец (cdr план)
         решение порядок))
   ((поиск конец (funcall порядок план)
      (cons (second (car план)) решение)
      порядок))
   (t (поиск конец (cdr план)
         решение порядок))))
```

Поиск в глубину, в ширину и по наилучшему варианту

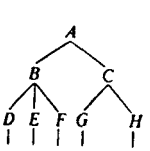
Мы уже попытались определить функцию ПОИСК в гибком и, используя функциональную абстракцию, в весьма общем виде. Наиболее важные способы поиска, т.е. поиск в глубину (depth-first), поиск в ширину (breadth-first) и поиск по наилучшему варианту (best-first) отличаются друг от друга лишь тем, в каком порядке изучаются состояния пространства поиска. Это свойство абстрагировано в функциональный аргумент ПОРЯДОК. Значением этой функции в каждой ситуации будет список анализируемых продукций (ПЛАН) в порядке, соответствующем способу поиска.

Порядок состояний при поиске в глубину определяется следующим образом:

```
(defun в-глубину (план)
   (append (применимы (second (car план)))
           (cdr план)))
```

В процессе поиска сначала изучаются состояния, непосредственно следующие за первым состоянием. Таким образом, поиск осуществляется в пространстве поиска сначала в глубину, и возврат происходит после исследования наиболее далеких состояний.

```
_(маршрут 'стокгольм 'выборг 'в-глубину)
(СТОКГОЛЬМ ХЕЛЬСИНКИ ТАЛЛИНН ЛЕНИНГРАД
ВЫБОРГ)
```



При поиске в ширину сначала изучаются все состояния каждого уровня и лишь после этого происходит переход на бо́льшую глубину. В функции упорядочивания это означает, что в каждом состоянии достижимые из первого состояния возможности помещаются в конец плана:

```
(defun в-ширину (план)
   (append (cdr план)
           (применимы (second (car план)))))

_(маршрут 'стокгольм 'выборг 'в-ширину)
(СТОКГОЛЬМ ХЕЛЬСИНКИ ТАЛЛИНН ЛЕНИНГРАД
ВЫБОРГ)
```

При поиске по наилучшему варианту вся последовательность состояний вновь упорядочивается по некоторому критерию так, что сначала в каждом состоянии всегда изучаются наилучшие варианты продолжения:

```
(defun наилучший (план)
   (sort (append (применимы (car план))
                 (cdr план))
         #'критерий))
```

SORT – это общая функция упорядочивания Коммон Лиспа и КРИТЕРИЙ – некоторый двуместный предикат, определяющий порядок двух состояний[1]).

Предположим, например, что мы предпочитаем сначала более северные маршруты и что географическая широта городов хранится в виде их свойства ШИРОТА. В этом случае КРИТЕРИЙ можно определить следующим образом:

```
(defun критерий (x y)
  (>= (get (second x) 'широта)
      (get (second y) 'широта)))

(setf (get 'хельсинки 'широта) 4)
(setf (get 'стокгольм 'широта) 1)
(setf (get 'таллинн 'широта) 2)
(setf (get 'ленинград 'широта) 3)
(setf (get 'выборг 'широта) 5)
```

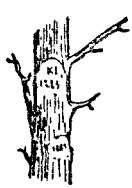
---

[1]) Хотя порядок определяется на состояниях, однако сортируются операторы. *–Прим. перев.*

```
_(маршрут 'стокгольм 'выборг 'наилучший)
(СТОКГОЛЬМ ХЕЛЬСИНКИ ЛЕНИНГРАД ВЫБОРГ)
```

Использование этого способа поиска дает результат, отличный от результата, получаемого при использовании поиска в глубину и поиска в ширину. При попытке сначала выбрать более северные варианты, будет найден второй возможный вариант решения, который совсем не исследует Таллинн.

С помощью представленного выше алгоритма поиска можно осуществить и другие способы, заменяя лишь параметр ПОРЯДОК. Более сложные функции могли бы, например, делать предварительную оценку различных направлений поиска и для повышения эффективности даже исключать некоторые альтернативы. Опасность таких *сокращающих* (pruning) эвристик заключается в том, что можно отвергнуть и такую альтернативу, которая позже может оказаться верной. В самом худшем случае могут оказаться выброшенными вообще все решения задачи.



В принципе в функции ПОИСК способ поиска можно было бы менять динамически в зависимости от ситуации, заново определяя параметр ПОРЯДОК перед каждой его передачей. Такие данные и решения, принимаемые о способе их обработки, называют *метазнаниями* (meta knowledge).

Применения продукционного программирования

С помощью продукционного программирования можно решить различные задачи связанные с принятием решений, управлением, упрощением, анализом, доказательством и другими моментами. Преимущество метода в том, что продукции в виде набора шаблонов, грамматических правил, множества аксиом или других подобных форм, близких решаемой задаче, можно записывать независимо от программы. Продукционная модель наглядна для пользователя, и ее легко модифицировать и расширять, поскольку изменение продук-

ции не обязательно влияет на другие продукции или интерпретатор продукций. Для увеличения эффективности интерпретатор продукций и связанное с ним сопоставление с образцом можно реализовать и в виде транслятора, который преобразует продукции и связи между ними непосредственно в лисповские функции.

Для продукционного программирования создано множество языков и средств разработки систем, основанных на обработке знаний (Planner, Conniver, OPS, Rosie, EMYCIN, KAS, Expert и др.). Большинство из них реализованы как встроенные в Лисп расширения языка. На продукционном программировании частично основано и излагаемое в следующей главе логическое программирование и язык Пролог.

Обычно продукции лучше всего применимы для представления простых ассоциаций вида наблюдение-действие. Механизм человеческой компетентности и опыта часто можно удовлетворительно смоделировать на поверхностном уровне без содержательного отображения глубинной сути, значения и причинных связей между явлениями. Продукционное программирование используется как метод представления знаний и программирования в разработке экспертных систем. Для компенсации ограничений, присущих продукционному программированию, его можно комбинировать с другими методами программирования и представления данных.

Литература

1. Barr A., Cohen P., Feigenbaum E. *The Handbook of Artificial Intelligence,* Vol. 1–3, Pitman, London, 1981–1982.
2. Buchanan B., Duda R. Principles of Rule-Based Expert Systems. Stanford University, Heuristic Programming Project, Report HPP–82–14, Stanford, 1982.
3. Davis R. Expert Systems Where are We? And Where Do We Go from Here? Massachusetts Institute of Technology, AI Laboratory, Memo 665, 1982.

4. Hayes-Roth F. Knowledge Based Expert Systems. *Computer,* October, 1984, pp. 263–273.
5. Hayes-Roth F., Waterman D., Lenat D. *Building Expert Systems.* Addison-Wesley, Reading, Massachusetts, 1983. [Имеется перевод: Хейес-Рот Ф., Уотерман Д., Ленат Д. Построение экспертных систем. –М.: Мир, 1987.]

6. Hyvönen E. *Asiantuntijajärjestelmien tietämystekniikka.* Knowledge Engineering Ky, Helsinki, 1986.
7. Gevarter W., An Overview of Expert Systems. National Bureau of Standards, Report NBSIR 82–2505, October 1982.
8. McDermott J.A. A Rule Based Configurer of Computer Systems. *AI Journal,* Vol. 19, 1982, pp. 39–88.
9. Nilsson N. *Principles of Artificial Intelligence.* Tioga, Palo Alto, California, 1980. [Имеется перевод: Нильсон Н. Искусственный интеллект. Методы поиска решений. – М.:Мир, 1973.]
10. Pearl J. *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley, London, 1984.
11. Shortliffe E.H. *Computer Based Medical Consultations: MYCIN.* American Elsevier, New York, 1976.
12. Sussman G.J., McDermott D.V. Why Conniving is Better than Planning. Memo No. 225A, AI Laboratory, MIT, Cambridge, Massachusetts, 1972.
13. Waterman D.A., Hayes-Roth P. (eds.) *Pattern-Directed Inference Systems.* Academic Press, New York, 1978.
14. Weiss S., Kulikowski C. *A Practical Guide to Designing Expert Systems.* Chapman and Hall, London, 1984.

Я мыслю, следовательно, я существую.

Декарт

Я не мыслю, следовательно, я не существую.

Г. Лихтенберг

2.5 ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

- **Декларативная программа не содержит алгоритма**
- **Процедурная семантика**
- **Отношение является обобщением функции**
- **Унификация структур**
- **Алгоритм унификации**
- **Логика хорновских предложений**
- **Логическая интерпретация хорновских предложений**
- **Логическое определение отношений**
- **Множество предложений трактуется как программа**
- **Метод резолюций**
- **Алгоритм доказательства**
- **Реализация интерпретатора**
- **Пролог использует поиск в глубину**
- **Развитие логического программирования**
- **Литература**

Функция – это отображение из множества определения в множество значений. В Лиспе с вычислением функции неразрывно связано *направление*: функция получает аргументы и возвращает значение. То же самое изображение функции не позволяет в "перевернутом направлении" вычислять значения аргументов исходя из значения функции. Эта несимметричность проистекает из того, что лисповские функции, как и написанные на традиционных языках программы, обрабатывают данные в порядке, задаваемом описанием алгоритма, несмотря на то что эту последователь-

ность можно в Лиспе отобразить весьма гибкими и общими формами. Такие языки назовем *процедурными языками* (procedural language). В процедурных языках и формализмах программирование и решение проблем сводится в основном к разработке алгоритма, выполняющего действия.

Декларативная программа не содержит алгоритма

В логических языках, известным представителем которых является Пролог, алгоритмы в таком смысле не используются. Если функциональные и операторные языки описывают, "каким образом" решается некоторая задача, то в логических языках для ее решения достаточно точного логического описания. Языки, в которых решение задачи получают из описания структуры и условий задачи, называют *декларативными языками* (declarative language). Декларативные формализмы отвечают на вопрос "что".

Поскольку в декларативной программе последовательность и способ выполнения программы не фиксируются, как при описании алгоритма, программы могут в принципе работать в обоих направлениях. Например, чисто логическая программа, написанная на Прологе, может на основе исходных данных вычислить результат, но и с тем же успехом без дополнительного программирования на основе результата – исходные данные.

Процедурная семантика

Чтобы в декларативном языке можно было выполнять разумные вычисления, для него наряду с декларативным смыслом определяется интерпретация в виде действий, или *процедурная семантика*. Однако, для того чтобы писать программы, знать о ней необязательно. Эту ситуацию можно, пожалуй, сравнить с естественным языком, который мы понимаем и используем, хотя мы не знаем о действии физических процессов в нашем мозгу и не понимаем логико-лингвистических процессов языка! Но, с другой стороны, понимание

механизмов, лежащих как за естественным, так и за формальным языком, полезно для использования языка.

В этой главе мы рассмотрим, каким образом в Лиспе осуществляется программирование, похожее на описанное выше декларативное программирование. Идея декларативного программирования получила наибольшее развитие в так называемом *логическом программировании* (logic programming), где декларативный смысл структур выражается в логических предложениях, и их процедурная интерпретация основана на специальных алгоритмах доказательства теорем. В Лиспе возможностей логического программирования можно добиться, запрограммировав алгоритм доказательств. В дальнейшем мы для примера реализуем небольшой интерпретатор Пролога. Но сначала рассмотрим понятие отношения, унификацию (unification) выражений и основы логики.

Prolog

Отношение является обобщением функции

Отношение (relation) определяется как подмножество прямого произведения (×) множеств.

Например, сложение двух целых положительных чисел можно представить в виде отношения между множествами целых чисел (N1, N2 и N3)

$$R \in N1 \times N2 \times N3$$

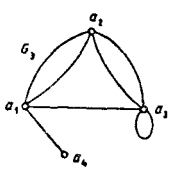
где n1+n2=n3 и ni принадлежит множеству Ni. Отношение можно и явно представить как множество троек:

$$R = \{(1\ 1\ 2),\ (1\ 2\ 3),\ (1\ 3\ 4),\ \ldots\}$$

Это отношение имеет три численных *аргумента*, т.е. оно является 3-местным. 0-местное отношение реализует константу, 1-местное – свойство, 2-х и более местное отношение можно графически представить в виде сети.

Отношение – это обобщение функции. Любую функцию можно представить в виде отношения с помощью следующего преобразования:

$$(fn\ x1\ x2\ \ldots\ xN) = y \Leftrightarrow (fn\ x1\ x2\ \ldots\ xN\ y)$$



Функция – это отношение, не содержащее двух элементов

$$(x1\ x2\ \ldots\ xN\ y1)$$

$$(x1\ x2\ \ldots\ xN\ y2)$$

таких, что y1≠y2, поскольку значение функции определяется однозначно.

Заметьте, что у функции, представленной в форме отношения, аргументы и результат находятся в равнозначных позициях аргументов отношения. Отношение, подобно настоящей функции, не содержит элементов асимметрии.

В процедурных методах и языках программирования понятие отношения используется для обозначения произвольной абстракции данных, например в языке запросов реляционной базы данных. Благодаря общности, симметричности и "глобальности" на их основе можно реализовать совершенно новые декларативные языки. И если в Лиспе за основу программирования приняты функции и лямбда-исчисление, то Пролог основан на более общем понятии отношения и так называемой хорновской логике.

Унификация структур

В декларативных языках программирования и интерпретации в симметричных формализмах используется

метод, с помощью которого можно сравнивать выражения и распознавать соответствия их частей между собой. Этот метод называют *унификацией* (unification) выражений. Унификация – это обобщение сопоставления с образцом, в котором два образца с переменными сравниваются друг с другом с целью их сопоставления в отличие от рассмотренного нами ранее сопоставления образца с константным образом. В результате унификации получается одна

или несколько *подстановок* (substitution), или соответствий, т.е. множество связей переменных, с помощью которых сравниваемые структуры можно сделать идентичными наиболее простым образом. Такая подстановка называется *наиболее общим унификатором* (most general unifier) образцов (выражений). Унифицируя, например, образец

```
(1 2 (? x))          ; "Какова сумма чисел 1 и 2?"
```

с тройками приведенного отношения сложения можно найти, что x=3. В этом случае унификация образцов соответствует обыкновенному вызову функции сложения. Но вычисления с помощью отношений можно осуществить также и в другом направлении, задавая в образце сумму константой, а аргументы с помощью переменных:

```
((? x) (? y) 3)      ; "Сумма каких чисел равна 3?"
```

Унифицируя с тройками отношения, можно найти ответы x=1 и y=2 или x=2 и y=1.

В этих примерах было достаточно рассмотренного ранее сопоставления с образцом. В следующем случае этого уже недостаточно, поскольку как в образце, так и в образе содержатся переменные. В этом случае унификация достигается подстановкой x=D и y=B:

```
((? x) b (? y) d)    ; образец 1
(d (? y) b (? x))    ; образец 2
```

Обратите внимание, что при унификации не имеет смысла использовать различающиеся типы связывания и подстановки переменных (типы > и <), как мы это делали при сопоставлении образцов, здесь последовательность связывания переменных не определяется непосредственно по сопоставляемому образцу, а динамически зависит от распознаваемого образа. Унификация дает одинаковый результат независимо от того, какой образец является образом и какой образцом.

Алгоритм унификации

Изложим далее простой алгоритм унификации. В дальнейшем будем называть унифицируемые образцы *термами*. Ограничимся рассмотрением термов, которые являются либо

- константными символами,
- переменными или
- списками, рекурсивно состоящими из термов.

Переменная сопоставима с произвольным термом, и она изображается в виде:

(? *переменная*)

На унификацию двух термов можно наложить следующие условия:

1) переменная унифицируется с произвольным термом;
2) два символа унифицируются лишь в случае, когда они идентичны;
3) списочный терм унифицируется, если унифицируются все его элементы;

Унификацию можно осуществить с помощью следующей функции УНИФИЦИРУЙ, возвращающей связи переменных, приводящие к унификации, в качестве значения:

```
(defun унифицируй (x y связи)
;; Пытаемся унифицировать термы x и y:
 (let ((x (значение x связи))
       (y (значение y связи)))
   (cond
;; переменные унифицируются с чем угодно:
      ((переменная-p x)
         (cons (list x y) связи))
```

```
      ((переменная-р у)
         (cons (list у х) связи))
;; идентичные константы унифицируются:
      ((or (atom х) (atom у))
         (and (eql х у) связи))
;; унифицируются головы и хвосты
;; структуры:
      (t (let ((новые-связи
                  (унифицируй (first х)
                              (first у)
                              связи)))
           (and новые-связи
              (унифицируй
                 (rest х) (rest у)
                 новые-связи)))))))
```

Связанные с переменными значения сохраняются в параметре СВЯЗИ, и их можно получить с помощью функции ЗНАЧЕНИЕ (обратите внимание на рекурсивность определения значения):

```
(defun значение (х связи)
;; Возвращаем значение переменной х из
;; окружения
  (if (переменная-р х)
      (let ((связь (assoc х связи
                          :test 'equal)))
        (if (null связь) х
            (значение (second связь)
                      связи)))
      х))
```

Встроенная в систему Коммон Лиспа функция ASSOC сравнивает головные элементы пар в ассоциативном списке функцией EQL, которая неприменима для сравнения двух списков (на логическом уровне). Однако предикат сравнения можно изменить в месте вызова с помощью ключевого параметра :TEST, как это уже делалось ранее. Благодаря этому стано-

вится возможной работа со списком пар, в котором головные элементы пар представляют собой списки.

Нам еще понадобится предикат ПЕРЕМЕННАЯ-Р, который распознает переменные:

```
(defun переменная-р (х)
   (and (listp x) (eq (first x) '?)))
```

Приведем примеры унификации:

```
_(унифицируй '((? x) c (? y))
             '((? y) c a))
(((? Y) A) ((? X) (? Y)))
_(унифицируй '(a (b (? x)) (c d))
             '((? y) (b (c d)) (? x)))
(((? Y) A) ((? X) (C D)))
```

Обратите внимание, что алгоритм может унифицировать структуры с произвольным уровнем вложенности.

Приведенная выше синтаксическая унификация не очень сложна. Ситуация осложнилась бы, если бы мы захотели учесть возможные в выражениях формальные свойства, такие как ассоциативность и дистрибутивность операций и др. В таком случае нужно было бы признать унифицируемыми и следующие формы:

((x + y) + z) и
(x + (y + z))

Мы не будем касаться таких случаев.

Логика хорновских предложений

Для записи отношений можно использовать различные формализмы. Особенно полезный, общеприменимый и эффективно реализуемый на вычислительной машине формализм предлагает *логика хорновских предложений* (Horn clause logic), которая является ограниченной формой *исчисления предикатов первого порядка* (first order predicate calculus).

В логике хорновских предложений используются только так называемые *предложения Хорна* (Horn clause). Их называют еще правилами (rule). В лисповской нотации хорновское предложение можно представить в следующем виде:

$(p\ q1\ q2\ \ldots\ qn)\ ;\ n \geq 0$

Правило состоит из *заключения* (head, consequence) p и *тела* или *предусловий* (body, precondition) q1 q2 ... qn. Заключение и элементы тела называют *предикатами*.

Как хорновские предложения, так и предикаты можно представить с помощью определенных ранее термов. Несмотря на внешнее сходство, предикаты нельзя путать с хорновскими предложениями. Предикат – это список, первый элемент которого – имя предиката, и далее следуют аргументы:

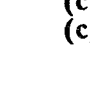
$(pred\ a1\ a2\ \ldots\ an)$

Аргументы ai являются термами, т.е. константами, переменными или составленными из них списками.

Логическая интерпретация хорновских предложений

Под декларативной интерпретацией предиката (pred a1 a2 ... an) в качестве отношения подразумевается, что между аргументами a1...an имеется отношение *pred*, или что (a1 a2 ... an) принадлежит отношению *pred*. Например:

(сумма 1 2 3) ; истина, поскольку (сумма 1 2) = 3
(супруги Филип Элизабет)



Хорновское же предложение (p q1 ... qN) интерпретируется как логическая импликация: "Если предикаты *q1,...,qn* истинны, то и предикат *p* истинен". Хорновское предложение соответствует в продукционном программировании продукции:

ЕСЛИ q1 и q2 и ... qn ТО p

Предполагается, что все переменные в хорновском предложении охвачены квантором всеобщности.

Специальным видам хорновских предложений, в которых заключение или тело ложны, придается следующая интерпретация:

Если тело (условие) пусто, то правило всегда истинно, является *фактом* (fact) и обозначается:

(*p*)

Например:

((сумма 1 2 3))

Если заключение пусто или ложно (nil), то мы имеем всегда ложный предикат:

(nil (сумма 1 2 4)) **; сумма 1 и 2 не равна 4**

Логическое определение отношений

С помощью хорновских предложений можно определять отношения в гибкой и интуитивно понятной форме правил. Например, отношение ДЕДУШКА можно было бы определить с помощью правила "у является дедушкой x, если z является одним из родителей x, и у есть отец z" следующим образом:

```
((дедушка (?x) (?y))      ; голова
 (родитель (? x) (? z)) ; тело
 (отец (? z) (? y)))
```

Отношение РОДИТЕЛЬ можно определить с помощью следующих предложений:

```
((родитель (? x) (?y))   ; голова
 (отец (? x) (? y)))     ; тело
```

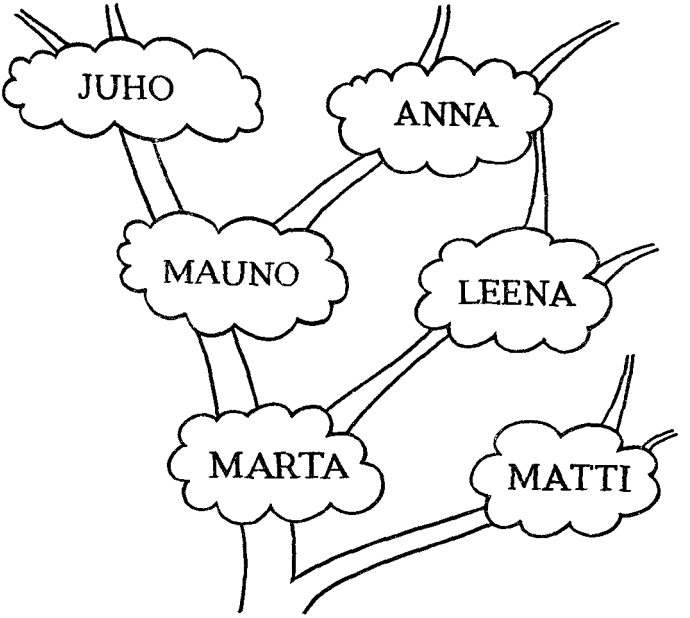


Рис. 2.5.1 Дерево родства.

```
((родитель (? x) (? y)) ; голова
 (мать (? x) (? y)))    ; тело
```

Область действия переменных – всегда лишь одно предложение.

Отношения ОТЕЦ и МАТЬ могут быть фактами, которые задаются в виде перечисления отношений. Например:

```
((отец marta mauno))
((отец mauno juho)) ...
```

```
((мать marta leena))
((мать mauno anna)) ...
```

Множество предложений трактуется как программа

Эти предложения образуют множество правил, в котором все использованные отношения перечислены или определены через другие отношения или рекурсивно через самих себя. Для такого множества хорновских предложений можно определить процедурную интерпретацию.

Программа, написанная на Прологе, образуется просто из множества хорновских предложений:

```
(setq программа
    '(((дедушка (? x) (?y))     ; правила
       (родитель (? x) (? z))
       (отец (? z) (? y)))
      ((родитель (? x) (? y))
       (отец (? x) (? y)))
      ((родитель (? x) (? y))
       (мать (? x) (? y)))
      ((отец marta mauno))      ; факты
      ((отец mauno juho))
      ((мать marta leena))
      ((мать mauno anna))))
```

Программа запускается путем передачи ей в качестве вопроса и для принятия решения некоторого предиката, например:

(мать marta leena) (1)
(родитель marta (?x)) (2)

Ответом будет либо логическое значение (1), либо связи переменных заданного вопроса, при которых предикат истинен (2):

⇒ **X = MAUNO** *или* **X = LEENA**

Программа работает в обоих направлениях:

(родитель (? ребенок) mauno)
⇒ РЕБЕНОК = MARTA

В общем программу, состоящую из логических предложений, можно считать реляционной базой данных, в которой данные и программы представлены в виде термов. Программе (базе данных) можно задавать вопросы, касающиеся записанных в ней отношений. И вопросы – предложения той же формы. Одинаковая форма представления данных и программы дает возможность обрабатывать прологовскую программу на Прологе таким же образом, как и в Лиспе. Определяя системное множество полезных и необходимых предикатов с побочным эффектом, таких как предикаты ввода и вывода, арифметические операции и другие, можно логику хорновских предложений превратить в самостоятельный язык программирования.

Метод резолюций

Остановимся далее более подробно на принципе работы интерпретатора Пролога.

Доказательство в случае хорновских предложений основывается на *методе резолюций* (resolution) (Robinson 1965). Для доказательства методом резолюций множество предложений сначала приводится к так называемой конъюнктивной нормальной форме, в которой из предложений удалены кванторы существования, импликации и эквивалентности:

c1 и c2 и c3 и ... cN

Здесь предложения ci могут содержать лишь связки "или" и отрицания. Предполагается, что переменные связаны квантором всеобщности.

В методе резолюций используется доказательство от противного. Предложение доказывается добавлением его отрицания с

помощью связки "и" к известному множеству предложений

(не c) и c1 и c2 и ... cN

и приведением такого множества предложений путем применения *правила резолюции* (resolution rule) к логическому противоречию. Правилом резолюции является форма:

((не p) или q1) и (p или q2)
⇔
(q1 или q2)

В результате применения правила получается новое сокращенное множество предложений, называемое *резольвентой* (resolvent). Правило применяется повторно до тех пор, пока множество предложений не будет редуцированно до логического противоречия:

(не p) и p

$$\frac{A,\ A \Rightarrow B}{B}$$

Для использования метода резолюций, чтобы исключить импликацию между заключением и телом, хорновские предложения предварительно преобразовываются в конъюнктивную нормальную форму по следующему логическому правилу:

p ⇐ q1 и q2 и ...qn
⇔
p и (не q1) и (не q2) и ...(не qn)

Затем отрицание (не p) доказываемого предиката p добавляется в множество хорновских предложений, формируемых из базы данных. Предикат x может быть противоречивым, только если в программе существует хорновское предложение (факт или правило), заключение которого имеет ту же форму, что и x:

nil и (не x) ; *отрицание доказываемого предиката*
...
p1 ; *факт (1)*
...
pn и (не q1) и ... (не qk) и ... (не qm) ; *правило (2)*
...

Противоречивость показана, когда найденное предложение (x≜p1) является фактом (1). Если с p связано тело (2) (x≜pn), то противоречивость можно доказать, показав противоречивость каждого отрицания (не qk), k=1...m из тела. Это можно осуществить рекурсивно таким же образом, как и для первоначального предложения (не x). Теперь изучаются лишь предложения с заключениями qk:

qk и (не w1) и ... (не wn) ; *n = 0, 1, 2, ...*

Рекурсия заканчивается на фактах.

Поскольку в обрабатываемом предикате и хорновском предложении могут быть переменные, то заключения предложений сравниваются с предикатом с помощью попытки их унификации. Так на каждом шаге получаются соответствия между переменными и термами, которые действуют и в дальнейшем. В процессе осуществления доказательства множество соответствий для переменных разных предложений все возрастает и часто становится опосредованным. Можно, например, требовать, чтобы значение переменной x одного предложения было бы таким же, как значение y некоторого другого предложения, а значение y в свою очередь определяется на основе некоторого третьего предложения. Механизм соответствует передаче параметров традиционных языков.

Алгоритм доказательства

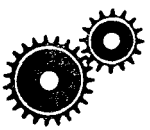
Описанный ранее простой способ доказательства позволяет интерпретировать в виде программы описание, образуемое из логических предложений. Представим его далее в немного более уточненной форме. Доказываемый в

настоящий момент предикат-теорему назовем *целью* (goal). Цель можно доказать следующим алгоритмом:

1. Найти для доказательства цели первое предложение, заключение которого унифицируется (unify) с целью.
2. Если предложение с таким заключением не найдено, то доказательство не удалось. Если предложение найдено и его тело пусто, то цель доказана. Если у предложения не пустое тело, то надо доказать его предикаты в качестве новых целей рекурсивно слева направо.
3. Если алгоритм зашел в тупик, т.е. доказательство не удалось, то надо вернуться (backtrack) в предыдущее место, где можно выбрать другую цель (если такая имеется) и продолжить доказательство[1].

Подбор связей переменных основан на методе проб и ошибок и на поиске. Доказываемый в настоящий момент предикат пытаются унифицировать первым возможным способом. Если решение окажется неверным, т.е. некоторый предикат позже не удастся унифицировать, то происходит возврат к предыдущей унификации, выбирается следующая из еще имеющихся возможностей, и делается попытка унифицировать оставшиеся предикаты другим способом.

Реализация интерпретатора



Описанная ранее процедура доказательства предложений образует простой интерпретатор Пролога, который мы далее запрограммируем на Лиспе.

Основной цикл интерпретатора Пролога можно реализовать так, как это показано на следующей странице.

Интерпретатор выводит знак вопроса и ожидает ввода пользователем доказываемых предикатов.

---

[1] Имеется в виду другое предложение, заключение которого унифицируемо с целью. *–Прим. перев.*

```
(defun prolog (программа)
 (progn
   (princ "?")
   (do ((предикаты (read) (read)))
       ((eq предикаты 'конец) 'конец)
     (докажи
      (list (поменяй-имена предикаты '(0)))
      '((основа)) программа 1)
     (princ "?"))))
```

Встречающиеся в них переменные переименовываются в уникальные имена функцией ПОМЕНЯЙ-ИМЕНА

```
(defun поменяй-имена (терм уровень)
;; На каждом уровне имена уникальные
 (cond
   ((переменная-р терм)
      (append терм уровень))
   ((atom терм) терм)
   (t (cons
        (поменяй-имена
          (first терм) уровень)
        (поменяй-имена
          (rest терм) уровень)))))
```

Предикат ПЕРЕМЕННАЯ-Р уже нами определен.

В дальнейшем с помощью численного параметра УРОВЕНЬ можно будет отличить связи одноименных переменных из различных правил:

```
_(поменяй-имена
   '((родитель marta (? x)))
   '(0))
((РОДИТЕЛЬ MARTA (? X 0)))
```

Введенные пользователем предикаты доказываются функцией ДОКАЖИ на следующей странице.

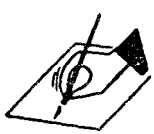
В начале окружение связей переменных СВЯЗИ пусто. Через параметр ПРОГРАММА передается используемая программа на Прологе. Когда с помощью функции ДОКАЖИ-КАЖДЫЙ предикаты доказаны,

```
(defun докажи
      (предикаты связи программа уровень)
;; Докажи предикат в текущем окружении
 (cond
   ((null предикаты)
      (выведи-связи связи связи)
      (princ "Еще? (д/н)")
      (eq (read 'н)
   (t (докажи-каждый программа программа
         (rest предикаты)
         (first предикаты)
            и уровень))))
```

функции ВЫВЕДИ-СВЯЗИ в конце результата выводятся значения связей переменных предиката с его уровня и пользователь задается вопрос, желает ли он с механизмом возвратов найти дополнительные решения. Из функции ДОКАЖИ-КАЖДЫЙ передаются используемые в процессе доказательства связи программа. Пролог, кроме дополнительные предикаты, предикат, указываемый в настоящий момент, связи переменных и уровень.

```
(defun докажи-каждый
      (остаток-прогр программа
       остаток-пред пред связи уровень)
;; Докажи предикаты с возвратами
 (if (null остаток-прогр) nil
   (let*
      ((теорема
         (изменяй-имена
            (first остаток-прогр)
            (list уровень)))
       (новые-связи
         (унифицируй пред
            (first теорема) связи)))
```

```
(cond
   ((null новые-связи)
      (докажи-каждый
         (rest остаток-прогр)
         программа остаток-пред
         пред связи уровень))
   ((докажи
      (append (rest теорема)
              остаток-пред)
      новые-связи программа
      (+ уровень 1)))
   (t (докажи-каждый
         (rest остаток-прогр)
         программа остаток-пред
         пред связи уровень))))))
```



Идея функции состоит в том, что еще не унифицированные предикаты содержатся в параметре ОСТАТОК-ПРЕД и связи переменных уже унифицированных предикатов – в параметре СВЯЗИ. Если предикат можно непосредственно унифицировать с некоторым фактом программы или с таким правилом, заключение которого унифицируется с доказываемым предикатом ПРЕД, то вызывается ранее описанная функция ДОКАЖИ. Используемую функцию унификации УНИФИЦИРУЙ мы уже описали ранее. В момент вызова уже унифицированный предикат удаляется из списка ОСТАТОК-ПРЕД и в качестве окружения берется полученное в результате унификации новое окружение НОВЫЕ-СВЯЗИ. Кроме этого, предикаты из тела правила добавляются (вызов APPEND) к списку подлежащих доказательству предикатов ОСТАТОК-ПРЕД.

Пролог использует поиск в глубину

В нашей программе предикаты из тела добавляются (вызов APPEND) в начало доказываемых предикатов. Это означает, что в доказательстве предполагается использование поиска в глубину, т.е. всегда сначала

исследуются предикаты из правил, находящихся в цепочке или дереве доказательства на более глубоком уровне.

В принципе новые предикаты, подлежащие доказательству, можно было бы добавить и в конец списка. Это привело бы к поиску в ширину, при котором всегда сначала доказываются предикаты более низкого уровня. Последовательность доказательства имеет значение для используемых в программе на Прологе предикатов с побочным эффектом (вывод, ввод и др.). Даже эффективность выполнения программы обычно зависит от того, описана ли проблема в форме, эффективной с точки зрения удобства доказательства. В программировании на Прологе можно выделить два этапа: сначала пытаются решить проблему логически на декларативном уровне; после этого рассматривается, как и в какой последовательности разработанная программа правильно решает задачу, и делается попытка оптимизировать определение.

Поиск основан на методе проб и ошибок. Программа все же в состоянии справиться со сделанными ранее ошибочными выборами. Две последние ветви предложения COND в функции ДОКАЖИ-КАЖДЫЙ реализуют необходимый для возврата механизм. Если предикаты из тела правила, заключение которого унифицировано, не удастся унифицировать рекурсивным вызовом ДОКАЖИ, то рекурсивный вызов из последней ветви дает новую возможность, пока все предложения программы не будут перепробованы для унификации с доказываемым предикатом. Лишь после этого в предложении IF предикат (NULL ОСТАТОК-ПРОГР) станет истинной.

Развитие логического программирования

Метод резолюций вместе с унификацией, поиском и связанным с ними механизмом возврата образуют основу используемой в Прологе техники программирования, *логического программирования* (logic programming).

Логическое программирование возникло в начале 70-х годов как продолжение исследований по машинному доказательству теорем, осуществляемых в области искусственного интеллекта. Исследования основывались на классической математической логике и достигли наибольшего успеха в результате разработки принципа резолюции, оказавшегося особенно удачным для программирования на вычислительной машине.

В начале 70-х годов Ковальски разработал теорию использования логики в качестве языка программирования (Kowalski 1974, 1979). Одновременно Колмероэ из университета Экс-Марсель реализовал первый основанный на логике язык программирования (Colmerauer et al. 1973, Roussel 1975), который был назван Прологом (Programmation en logique, Programming in logic).

$p \wedge q$

$p \vee q$

$\sim p$

$p \supset q$

Однако практическое применение Пролог получил лишь, когда Уоррен (1977) создал для него эффективную реализацию для вычислительной машины DEC-10 в Эдинбургском университете. В систему входили как интерпретатор, так и транслятор, каждый из которых в основном был написан на Прологе. После этого Пролог-системы начали быстро создаваться для многих машин.

Возможное в Прологе реляционное и логическое программирование завоевывало все большее количество горячих сторонников. Многие Лисп-системы содержат встроенный интерпретатор Пролога или другого подобного языка. Основанное на логике программирование с помощью правил в будущем может таким же образом войти в Лисп, как, например, вошло объектно-ориентированное программирование Смолтолка.

Логическое программирование хорошо подходит для решения проблем, для работы с формальными и естественными языками, для баз данных, запросных и экспертных систем и для других дискретных невычислительных задач. Пользователя привлекает ясность, содержательность программ и их нетехнический характер. В программе не нужно описывать, каким

образом решается задача. Достаточно описания самой задачи и того, что желательно узнать.

Однако логическое программирование с использованием лишь хорновских предложений было бы слишком узконаправленным. Поэтому, кроме этого, используются и другие методы программирования, например управляющие структуры процедурного программирования. Некоторые задачи по своему характеру процедурные, и программировать их чисто декларативными средствами непрактично. Нужны и более развитые типы данных, такие как массивы, структуры и объекты. Пролог и логическое программирование непрерывно расширяются, схватывая все новые методы программирования и формы изображения именно в направлении процедурного и объектно-ориентированного программирования, а также в направлении параллельных вычислений.

Литература

1. Bramer M., Bramer D. *The Fifth Generation – An Annotated Bibliography*. Addison-Wesley, New York, 1984.
2. Campbell J. A. *Implementations of Prolog*. Ellis Horwood, London, 1984.
3. Carlsson M., Kahn K.: *LM-Prolog User Manual*. Technical Report No. 24, UPMAIL, Uppsala University, Uppsala, 1983.
4. Clocksin W., Mellish C. *Programming in Prolog*. Springer-Verlag, New York, 1981. [Имеется перевод: Клоксин У., Меллиш К. Программирование на Прологе. –М.: Мир, 1987.]
5. Colmerauer A., Kanoui H., Roussel P., Pasero R. *Un Systeme de Communication Homme-Machine en Francais*. Universite d'Aix-Marseille, 1973.
6. Jaakkola H. *Viidennen sukupolven tietokone-projekti. Tilanne 12/84*. Käsikirjoitus, Tampereen teknillinen korkeakoulu, 1985.
7. Kahn K. Pure Prolog in Pure Lisp. *Logic Programming Newsletter*, Winter 1983/1984, Universidade Nova de Lisboa, 1983.

8. Kay M. Unification in Grammar. *Natural Language Understanding and Logic Programming,* Conference Proceedings, Institut National de Recherche en Informatique et en Automatique (INRIA), Rennes, 1984.
9. Kowalski R. Predicate Logic as a Programming Language. *IFIP 74,* Stockholm, 1974, pp. 569–574.
10. Kowalski R. *Logic for Problem Solving.* North-Holland, Amsterdam,1979.
11. Michie D. Expert Systems. *Computer Journal,* No 23, 1980, pp. 369–376.
12. Poe M., Nasr R., Potter J., Slinn J. A KWIC Bibliography on Prolog and Logic Programming. *Journal of Logic Programming,* No. 1, 1984, pp. 81–142.
13. Robinson J. A Machine-Oriented Logic Based on Resolution Principle. JACM, Vol. 12, Jan. 1965, pp. 23–41. [Имеется перевод: Робинсон Дж.А. Машинно-ориентированная логика, основанная на принципе резолюции. – В кн. Кибернетический сборник, вып. 7. –М.: Мир, 1970.]
14. Robinson J., Sibert E. LOGLISP: An Alternative to Prolog. *Machine Intelligence 10,* Edinburgh University Press, Edinburgh, 1982.
15. Roussel P. *PROLOG Manuel de Reference et D'utilisation.* 1-Sep-75, Technical Report University of Marseilles, Marseilles, 1975.
16. O'Shea T., Eisenstadt M. *Artifical Intelligence, Tools, Techniques, and Applications.* Harper et Row, New York, 1984.
17. Sato M., Sakurai T. QUTE: A Prolog/Lisp Type Language for Logic Programming. *Proceedings of IJCAI-83,* Karlsruhe, William Kaufmann, Palo Alto, 1983.
18. Shieber S., Karttunen L., Pereira F. (eds.) Notes from the Unification Underground. A Compilation of Papers on Unification-Based Grammar Formalisms. Techical Note 327, SRI-International, Menlo Park, 1984.
19. Warren D., Pereira L., Pereira F. Prolog – the Language and its Implementation Compared to Lisp. *SIGPLAN Notices,* Aug., 1977.

Определение понятий – это начало ссылок.

Сократ

Мысли не сохраняются, их надо во что-то воплотить.

А. Уайтхед

2.6 ОБЪЕКТНОЕ ПРОГРАММИРОВАНИЕ

- **Объектное мышление и объектное программирование**
- **Объект, класс объектов и метакласс**
- **Объект содержит данные и действия**
- **Свойства и состояние объекта**
- **Действия или методы объекта**
- **Сообщения управляют вычислением**
- **Подкласс и надкласс**
- **Естественный класс и качественный класс**
- **Иерархия классов и механизм наследования**
- **Порядок наследования в иерархии классов**
- **Композиция методов в вычислениях**
- **Базовые классы и метаклассы системы**
- **Пример системы – Flavors**
- **DEFFLAVOR определяет класс**
- **MAKE-INSTANCE создает новый объект**
- **DEFMETHOD определяет метод**
- **SEND посылает сообщение**
- **Объекты моделируют мир проблемы**
- **Применимость объектного программирования**
- **Развитие объектного мышления и программирования**
- **Литература**

Наиболее абстрактную и многообразную из рассмотренных до настоящего момента моделей программирования и вычисления предлагает *объектное программи-*

рование (object programming). В объектном программировании обобщаются свойства функционального и операторного подхода, а также программирования, управляемого данными и событиями. Для обозначения объектного программирования используется и более длинный термин *объектно-ориентированное программирование.*

В этой главе мы изложим принципы и совокупность основных понятий объектного мышления на примере системы Смолток из-за ясности ее построения и ее универсальных свойств. В качестве примера мы рассмотрим систему Flavor, включенную в Зеталисп, объекты которой более разнообразны, чем в Смолтолке, и которая, кроме того, представляет интересующий нас лисповский мир.

Объектное мышление и объектное программирование

В объектном программировании не предполагается, что программа состоит из набора подпрограмм и данных, ее основными составляющими единицами являются объекты, содержащие как процедуры, так и данные. Идея объединения данных и программы является основной идеей объектной модели и основой объектного программирования.

Объекты подразделяются на различные типы или *классы* (class) в зависимости от присущих им свойств и действий, которыми они обладают. Свойства и действия объектов одного типа одинаковы. Набор свойств объектов разных типов различен, хотя некоторые свойства и действия могут при этом совпадать. Интерфейсы объектов и взаимодействие между объектами одного или разных типов определяются на основе присущих типу действий и соответствующих им сообщений.

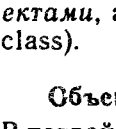
Объект, класс объектов и метакласс

У объектов так же, как у функций, замыканий и структур можно различить *определение* (object definition) и *вызов* (call), с помощью которого из определения создаются фактические объекты или *экземпляры* (instance) объектов. Определение объекта – это абстрактное описание типа объекта, которое представляет все объекты данного типа. Тип объекта часто называют *классом объектов* (object class) или просто *классом* (class). Класс – это множество всех потенциальных объектов, которые могут возникнуть из его определения.

Способ определения класса в различных Лисп-системах разный. В качестве класса можно рассматривать и само определение, которое тоже, в известном смысле, является объектом, а именно языковым объектом, изображающим класс объектов. Поэтому определения классов называют (в Смолтолке) *метаобъектами*, а образуемый ими класс – *метаклассом* (metaclass).

Объект содержит данные и действия

В первой части книги мы определяли абстрактный тип данных через объекты данных и их свойства с помощью применимых к ним действий. Таким образом, внутреннюю структуру и форму представления объектов можно было отделить от внешнего мира и единообразно определить их интерфейс. Пользователь поэтому не должен был знать деталей реализации типа. Достаточно того, что тип сам об этом знает.



В объектах используется тот же принцип. Объект содержит как данные наподобие структуры, так и действия наподобие функций и подпрограмм. Но набор действий здесь обобщен и расширен. Программист может определить для объекта произвольные действия.

Свойства и состояние объекта

Содержащиеся в объекте данные находятся в переменных, которые называют *переменными экземпляра* (instance variables) или *свойствами* (attribute). Переменные экземпляра являются местными переменными каждого объекта, у которых есть имя и значение. У объектов одного типа переменные экземпляра имеют одинаковые имена, но их значения индивидуальны.

Значения переменных экземпляра определяют *состояние* (state) объекта, которое индивидуально и меняется во времени для каждого объекта. Состояние объекта можно исследовать и делать в нем изменения с помощью действий объекта.

Действия или методы объекта

Предусмотренные в объекте действия называют *методами* (method). Методы – это функциональные свойства, определяющие использование объекта в виде функции (behaviour, functionality) или процедуры с побочным эффектом.

Объекты индивидуальны, и у каждого из них есть свое текущее состояние и свой набор соответствующих типу методов. Вычисления формируются из событий и запусков, возникающих в тот момент, когда объекты посылают другим объектам сообщения, интерпретируют их и реагируют на них, активизируя подходящие методы, которые еще посылают сообщения и т.д. Вызов действий обобщен и его единообразность обеспечивается специальным механизмом, основанным на программировании, управляемом данными, который называется *обменом сообщениями* (message passing).

Метод определяется в Зеталиспе так же, как и функция или подпрограмма, но собственной формой, в которой перечисляются класс объектов метода и активизирующее метод сообщение:

(DEFMETHOD (*класс объектов сообщение***)** *лямбда-список форма***)**

Текстуально описания записываются вне определения класса, но система автоматически привязывает их к выбору действий упомянутого класса. Выбор методов не фиксирован, и при надобности его можно дополнять и изменять.

У объектов различных типов могут быть одноименные, но работающие по-разному методы. Способ, которым объект реагирует на посланное ему сообщение, может зависеть от типа объекта. Это свойство тоже являлось одним из принципов абстракции данных и универсального программирования.

У объектов различных типов могут быть и общие методы. К этому вопросу мы вернемся позже.

Сообщения управляют вычислением

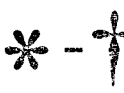
Метод объекта можно активизировать, послав соответствующее этому методу *сообщение* (message). Сообщение состоит из названия метода и возможных аргументов.

Посылка сообщения объекту осуществляется вызовом универсальной функции, в которой объекту передается имя вызываемого метода и используемые им аргументы. Принимающий сообщение объект содержит механизм, который опознает сообщение, выбирает соответствующий ему метод, активизирует его и передает ему содержащиеся в сообщении требуемые методом параметры. Метод вычисляется так же, как обыкновенная функция, и значение возвращается в качестве ответа на сообщение. Вычисление метода может в результате побочного эффекта изменить состояние объекта и быть причиной посылки новых сообщений другим объектам.

Если у объекта нет метода, соответствующего принятому сообщению, то сообщение теряется. Обычно причиной этого является ошибка или по крайней мере исключительная ситуация, к обработке которой нужно быть готовым. Набор методов не фиксирован, его можно свободно расширять, определяя новые методы.

После обработки сообщения и после того, как объект перестанет быть активным, он и его состояние

сохраняются и объект будет ожидать новых сообщений. Переменные экземпляра и их значения не исчезают, как в случае обыкновенных функций, а их значения сохраняются и могут использоваться, когда объект вновь получает сообщение. В этом смысле объект обладает свойством замыкания, которое тоже запоминает связи (свободных) переменных предыдущего запуска.



Посылка сообщения соответствует вызову функционала FUNCALL. Если представить, что определение функции МЕТОД сохранено в списке свойств объекта ОБЪЕКТ под именем МЕТОД, то его можно было бы вызвать так:

(FUNCALL (GET объект метод) аргументы)

Так осуществляется явной связь между объектным мышлением и списком свойств. Список свойств символа – это в некотором смысле тоже объект. Его методами являются, например, системные функции SETF, GET и REMPROP.

Объекты – это одновременно и близкие "родственники" структур. Определение структур DEFSTRUCT автоматически генерирует для работы со структурой соответствующие ее типу функции создания, чтения и присваивания. Их можно считать методами структуры.

В объектной системе возможности определения методов более многосторонни. Они открыты для пользователя, и форма вызова более простая. Кроме того, определение типов сделано более удобным путем использования специального механизма наследования, к которому мы в дальнейшем вернемся.

Объекты, методы и обмен сообщениями позволяют использовать новый принцип разбиения и вычисления программы, более общий и гибкий, чем традиционная идея подпрограммы. Ход событий не управляется извне, а осуществляется в зависимости от объектов.

Объекты являются более независимыми единицами, чем традиционные структуры и подпрограммы. Они

могут содержать сведения о собственном строении и функционировании, участвовать в управлении вычислением и взаимодействовать с другими объектами, посылая и принимая сообщения.

Подкласс и надкласс

Рассмотрим два класса, скажем A и B. Множества их свойств P(A) и P(B) могут находиться в различных взаимоотношениях:

1. P(A) и P(B) *не пересекаются*, т.е. у объектов классов нет общих свойств.
2. P(A) и P(B) *пересекаются*, следовательно, у них есть как общие, так и различные свойства.
3. P(A) *содержит* множество P(B) или наоборот.

Если P(A) содержит множество P(B), будем говорить, что класс A является *подклассом* (subclass) класса B и соответственно класс B является *надклассом* (superclass) класса A. Свойства могут и совпадать, но тогда мы имеем дело с тем же классом.

Таким образом, у подкласса всегда больше свойств, чем у содержащего его надкласса. Его объекты являются в отношении некоторых свойств более *конкретными* или *специфичными* (specific) по сравнению с объектами надкласса, которые в свою очередь являются более *универсальными* (generic).

Естественный класс и качественный класс

В Зеталиспе классу соответствует понятие *аромат* (flavor). Однако аромат более общее понятие, чем класс Смолтолка. Аромат может представлять и некоторые абстрактные свойства, которые как таковые не могут быть реализованы объектом. Например, понятие "круглый" нельзя реализовать конкретным самостоятельным объектом. Таким же образом "финское национальное самосознание" представляет собой абстрактную совокупность свойств или характеристик, которое можно реализовать лишь через конкретные объекты,

как, например, личности, вещи, действия и пр. Назовем такие классы *характеристическими классами* (mixing flavor) или *классами свойств* в отличие от классов объектов, которые назовем *естественными*.

Характеристический класс можно использовать в определении класса объектов для того, чтобы он как бы "ароматизировал" объекты некоторого *базового* (base flavor) класса объектов. Аналогично добавлению в пищу разных приправ объекты можно снабдить разными оттенками и ароматами.

Если в процессе трансляции у классов объектов можно найти общие свойства, то можно их выделить и определить в виде характеристического класса. Так можно значительно сократить определения классов. Таким образом, характеристический класс является еще одним механизмом абстрагирования данных и действий.

Иерархия классов и механизм наследования

Если определения классов образуют иерархическую структуру, то необходим специальный механизм, с помощью которого организуется множество свойств. Этой цели служит *механизм наследования* (inheritance mechanism), содержащийся в той или иной форме во всех иерархических объектных системах.

Посредством механизма наследования новый определяемый класс может автоматически унаследовать свойства (переменные экземпляра и методы) класса, определенного как его надкласс. С помощью наследования достигается та же цель, что и в иерархии типов Коммон Лиспа: более общие свойства, связанные с типом более высокого уровня, нет необходимости специально определять в связи с типами нижнего уровня. Достаточно указать, из какого надкласса они наследуют свойства. Таким образом, можно упростить определения и сэкономить усилия на их поддержку. Изменения можно локализовать (localize) лишь в одном изменяемом месте программы.

Наиболее важное преимущество объектного программирования, пожалуй, и состоит в том, что оно предлагает механизм абстракции для разбиения задачи и ее решения на части. С другой стороны, оно предполагает, что объекты проблемной области, а также их свойства и зависимости должны быть тщательно проанализированы.

Чтобы различить классы можно использовать естественное разбиение объектов на разные классы. Например, живые существа разделяются на млекопитающих, птиц, пресмыкающихся, рыб, насекомых и т.д. Далее они подразделяются на подклассы в соответствии с дополнительными свойствами каждого вида.

Кроме того, можно выделить абстрактные совокупности свойств в характеристические классы и использовать их в определениях. Характеристический класс может содержать свойства, противоположные или аналогичные свойствам базовых классов в их естественной классификации. Например, классы "солнце" и "яблоко" могли бы унаследовать из некоторого класса геометрических свойств свойство "круглый".

Порядок наследования в иерархии классов

В разных системах приняты различные соглашения о том, какова может быть топология структур наследования. Например, в Смолтолке у класса может быть только один надкласс, но несколько подклассов. В этом случае возможны лишь структуры наследования в виде дерева. Зеталисп и Loops допускают принадлежность класса одновременно нескольким надклассам. Структура наследования здесь может быть подобна ациклической сети, похожей на решетку.

На самом деле система Flavor допускает и циклическое определение, но исключает рекурсивное наследование. Структура определений может быть направленной сетью общего вида, но система распознает циклические определения и обрывает в этом случае наследование. Каждое свойство учитывается лишь один раз. Даже такие многообразные способы наследования могут оказаться ограниченными. Например, в

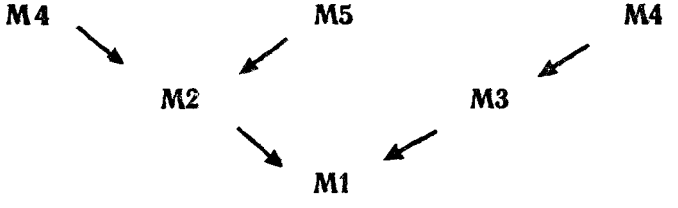
математике и в естественном языке используются рекурсивные типы.

Однако для большинства практических применений достаточно ациклических структур. В этом случае класс наследует свойства не только классов, непосредственно стоящих над ним, но и свойства их надклассов, т. е. свойства всех классов, расположенных выше.

Если класс наследует свойства из многих надклассов, то встает вопрос о порядке наследования, когда одноименные свойства наследуются из надклассов, находящихся в различных ветвях, или если один и тот же класс встречается в структуре несколько раз. В каком порядке находятся классы и в каком порядке учитываются свойства? При этом говорят о *комбинации* (combination) методов.

В принципе возможными последовательностями поиска являются, например, поиск в глубину (depth first) и по уровням (level first). В каждой ветви возможными порядками прохождения являются прямой (preorder) и обратный (pastorder). Под прямым порядком понимают порядок, в котором узел дерева (или сети) проходится раньше прохождения в том же порядке его поддеревьев (сетей). Обратный порядок можно определить аналогично.

В системе Flavor в прохождении иерархии ароматов применяется поиск в глубину и прямой порядок обхода. Свойства наследуются в том порядке, в котором надклассы указаны в описании класса. Класс с тем же именем повторно не проходится. Предположим, что иерархия классов выглядит так:



В этом случае порядок прохода получился бы следующий:

M1 → M2 → M4 → M5 → M3

Обратите внимание, что класс M4 проходится лишь раз.

Поскольку в разных классах могут быть одноименные, но различные по определениям методы, то для избежания конфликтов нужно договориться о том, какой из них останется в силе. Здесь разумно ограничиться двумя возможностями: в силе останется либо первый метод, либо последний.

В системе Flavor в силе остается первое встретившееся свойство. Если, например, в классе M5 был бы некоторый метод с тем же именем, что и в классе M4, то метод из M5 не учитывался бы. Система, однако, дает возможность изменить заложенный по умолчанию порядок следования. Если, например, определяется, что в силе остается последний метод, то говорят, что он *перекрывает* (override) предыдущие вхождения.

Переменные экземпляра подкласса определяются как множество своих переменных экземпляра и переменных экземпляра надклассов. Если у надкласса некоторого класса есть переменная экземпляра с тем же именем, то она является *общей* (shared instance variable) для класса. Общие переменные можно использовать для связи между классами. Одну и ту же переменную подкласса можно, например, присваивать и читать методами, определенными в разных надклассах.

Композиция методов в вычислениях

В объектном программировании вычислениями управляют в основном с помощью методов, вызывающих друг друга, подобно вызову подпрограмм в традиционном программировании. Кроме того, в управлении вычислениями могут использоваться и другие средства, например управление на основе событий. На последовательность применения методов могут повлиять связанные с ними *методы-демоны* (daemon method).

Методы-демоны, или короче демоны, – это вспомогательные методы, которые объявляются вместе с определением основного метода. Если основной метод получает сообщение, то без особого упоминания вычисляется и демон.

Демон-дерево.

Демоны бывают двух сортов: *пред-* и *постдемоны* (before- и after-daemon). Преддемон – это метод, который выполняется всегда перед вычислением основного метода. Постдемон выполняется соответственно после вычисления основного метода. С помощью пред- и постдемонов можно, например, проследить, когда некоторое сообщение посылается или возвращается объектом, изменить другие объекты на основе получаемых или посылаемых сообщений, разбить задачу данного метода на части и т.д.

Композицию методов можно осуществлять не только по времени через пред- и постдемоны, но и другими средствами. Способ композиции методов можно выражать, например, формой, на основе которой определяется вычисление методов, связанных с известным сообщением или ситуацией. Например, композиция методов с помощью формы AND вычисляется до тех пор, пока не встретится первый метод, значение которого NIL. Пользователь может и сам определять функции, объединяющие методы.

Базовые классы и метаклассы системы

Кроме определенных пользователем объектов и классов объектная система обычно содержит общие системные классы, свойства которых доступны всем. Такие классы мы назовем *базовыми классами*. Например, в Зеталиспе есть базовый *ванилиновый* аромат (vanilla flavor). Его свойства и методы наследуются автоматически всеми объектами. В системе Loops соответствующий базовый класс называют классом объектов (object class).

Базовый класс может содержать различные системные свойства и действия, такие как ссылка на самого себя (self), на имя (или определение) класса, методы, вызывающие объект и описывающие его свойства, и другие свойства по умолчанию (default method).

В Loops известны также понятия класс классов (class class) и метакласс (metaclass). Класс классов определяет свойства по умолчанию для всех определений классов, если определения интерпретируются как описывающие объекты. Метакласс – это класс, определяющий свойства по умолчанию для классов, которые создают классы. Это единственный объект, который сам является метаклассом.

Пример системы – Flavors

В качестве объектной системы, основанной на Лиспе, рассмотрим немного подробнее систему Flavors, являющуюся расширением Коммон Лиспа, входящего в систему Зеталисп на Лисп-машинах.

DEFFLAVOR определяет класс

В системе Flavors форма определения класса или аромата напоминает определение функции, макроса и структуры:

(DEFFLAVOR *класс* (*свойства*)
(*надклассы*)
(*режимы*))

В некоторых применениях можно было бы, например, КОРАБЛЬ определить следующим ароматом:

```
(defflavor корабль                    ; класс
   ;; переменные экземпляра
   (имя x y x-скорость y-скорость)
   (движущееся-тело)                  ; надклассы
   :gettable-instance-variables  ; режимы
   :settable-instance-variables)
```

У объектов КОРАБЛЬ в виде свойств есть ИМЯ, координаты местонахождения X и Y, а также компоненты скорости X-СКОРОСТЬ и Y-СКОРОСТЬ. Кроме этого, объекты наследуют другие свойственные объектам класса ДВИЖУЩЕЕСЯ-ТЕЛО свойства и действия. К режимам мы далее вернемся подробнее.

MAKE-INSTANCE создает новый объект

На основе определения класса мы можем создавать новые экземпляры (instance), принадлежащие данному классу. Это осуществляется вызовом MAKE-INSTANCE:

(MAKE-INSTANCE *класс*)

Например, с помощью класса КОРАБЛЬ можно создать судно:

```
(setq корабль1 (make-instance 'корабль))
```

Поскольку позже мы хотим сослаться на созданный объект, то присваиваем его значение переменной КОРАБЛЬ1.

Переменным экземпляра можно во время вызова дать значение. По нашему определению одно из свойств корабля – это переменная экземпляра ИМЯ. Ей можно присвоить значение в момент вызова:

```
(setq корабль78 (make-instance 'корабль
':имя 'titanic))
```

Переменная КОРАБЛЬ78 – это внешнее имя для ссылки, которое, например, используется служащим порта или программистом в модели порта, в то же время TITANIC – это свойство, связанное с самим судном, как прикрепленная к нему табличка. Каждое из них можно изменить, но только различными способами.

DEFFLAVOR реализован в виде сложного макроса, похожего на форму DEFSTRUCT Коммон Лиспа. Побочным эффектом его вызова будет автоматическая генерация необходимых функций доступа, которые можно использовать в объектно-ориентированном чтении и присваивании значений переменным. Побочным эффектом, возникающим в связи с генерацией объекта, можно управлять посредством следующих, необязательно задаваемых в определении, ключевых параметров (режимов).

1. Режим :GETTABLE-INSTANCE-VARIABLES приводит к генерации системой методов, читающих значения переменных экземпляра (в примере: ИМЯ, :X, :Y, :X-СКОРОСТЬ и :Y-СКОРОСТЬ).
2. Режим :SETTABLE-INSTANCE-VARIABLES приводит к генерации методов для присваивания новых значений переменным экземпляра. Методы присваивания значения имеют форму :SET-*x*, где *x* –имя переменной экземпляра. Например, имя метода, присваиваемого переменной ИМЯ, будет :SET-ИМЯ.

DEFMETHOD определяет метод

Определение класса метода пользователя производится формой DEFMETHOD:

(DEFMETHOD (*класс тип-метода метод***)**
*лямбда-список форма***)**

DEFMETHOD связывает определение класса с методом определенного типа, который вызывается с аргументами, задаваемыми лямбда-списком. Телом метода является произвольное выражение.

В определении обыкновенного метода параметр *тип-метода* не используется. Он предназначен для определения пред- и постдемонов. Тип демона выражается ключевыми словами :BEFORE и :AFTER (пред- и постдемоны). Для других способов композиции существуют свои ключевые слова.

Например, для объекта класса КОРАБЛЬ можно определить метод :СКОРОСТЬ, который вычисляет скорость судна:

```
(defmethod (корабль :скорость) ()
   (sqrt (+ (* x-скорость x-скорость)
            (* y-скорость y-скорость))))
```

SEND посылает сообщение

Кроме перечисленных ранее методов чтения и присваивания переменным экземпляра теперь можно объектам типа КОРАБЛЬ посылать и сообщение :СКОРОСТЬ,

которое возвращает в качестве значения скорость объекта (в этом случае у переменных экземпляра объекта X-СКОРОСТЬ и Y-СКОРОСТЬ, отображающих компоненты скорости, должны быть значения).

Сообщение посылается формой SEND:

(SEND *объект метод* **&REST** *аргументы***)**

Объект – это объект, которому посылается сообщение *метод* с данными аргументами. Получающий сообщение объект активизирует соответствующий ему метод и возвращает значение тому, кто послал сообщение. Пример на следующей странице.

Механизму передачи сообщений соответствует универсальный вызов применяющего функционала FUNCALL, передающего вызываемый метод вместе со своими параметрами объекту, интерпретируемому как

```
_(send корабль78 ':имя) ; имя дано объекту
TITANIC                 ; при создании
_(send корабль78 ':set-имя 'BORE)
BORE                    ; дается новое имя
_(send корабль78 ':имя) ; спрашивается имя
BORE
_(send корабль78 ':set-x-скорость 3.0)
3.0             ; присваивается x-скорость
_(send корабль78 ':set-y-скорость 4.0)
4.0             ; и y-скорость
_(send корабль78 ':скорость)
5.0     ; спрашивается скорость
```

функциональный объект:

(FUNCALL *объект метод* **&REST** *аргументы*)

Если мы хотим кроме кораблей образовывать и другие движущиеся тела и вычислять их скорость, то можно использовать более общий тип объекта, например ДВИЖУЩЕЕСЯ-ТЕЛО. Определенные для объекта КОРАБЛЬ переменные экземпляра X, Y, X-СКОРОСТЬ, Y-СКОРОСТЬ и метод :СКОРОСТЬ можно теперь перенести в определение класса объектов ДВИЖУЩЕЕСЯ-ТЕЛО, в этом случае их не надо вновь перечислять в определениях подклассов. Достаточно сослаться на их принадлежность надклассу ДВИЖУЩЕЕСЯ-ТЕЛО, например:

```
(defflavor нло (наблюдение) ; добавочная переменная
               (движущееся-тело)    ; надкласс
               ...)
```

Свойства и действия надкласса распространяются автоматически на объекты подклассов. Например, в определении НЛО нужно перечислить лишь те свойства (переменная экземпляра НАБЛЮДЕНИЕ), которые отличают его от других движущихся тел.

Соответственно можно определить другие классы, например: ВРАЩАЮЩЕЕСЯ-ТЕЛО, КРУТЯЩЕЕСЯ-ТЕЛО, КАТЯЩЕЕСЯ-ТЕЛО, СТАЛКИВАЮЩЕЕСЯ-ТЕЛО

– и для каждого из них характерные переменные экземпляра и действия. Так можно построить семантическую модель механики твердых тел.

Объекты моделируют мир проблемы

В объектном программировании строение объекта и его действия можно описать как единое целое. Свойства, зависимости и поведение реальных объектов так же, как их состояния и закономерности, теперь, как правило, можно запрограммировать во многом более адекватно решаемой задаче. Моделирование реальных объектов с помощью активных объектов нередко более естественно, чем с помощью обособленных и пассивных чисел и других данных, обработка которых осуществляется со стороны.

Благодаря внешнему воздействию свойства объектов и их состояние могут изменяться в соответствии с действительностью и объекты могут сами реагировать и воздействовать на окружающую среду. Они могут испытывать внешнее влияние и раздражения, самостоятельно интерпретировать получаемые сообщения и влиять на другие объекты. Их можно наблюдать, и на основе их состояния и их деятельности делать выводы.

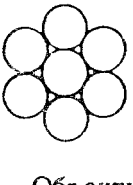
Применимость объектного программирования

Характерно то, что объектное программирование становится тем более необходимым, чем больше различных типов данных используется в программе и чем длиннее становятся программы. Выгоднее объединять действия с объектами данных, чем хранить их отдельно, поскольку обычно способ обращения с данными зависит от их представления и строения.

Например, действие сложения выполняется поразному для чисел различных типов, матриц и некоторых других математических объектов. В то же время способ записи действия и абстрактное его значение совпадают. Поэтому естественно выделить для каждого типа специфические свойства, требуемые в вычислениях от операндов различных типов, и определить общие

свойства сложения с помощью общего для них метода более высокого уровня.

Предпосылкой объектного программирования является прояснение и анализ основных понятий проблемного мира. Однако объекты, зависящие от целевой системы, и отношения между ними могут быть сложными и трудно различимыми. В качестве примера большой системы можно упомянуть естественный язык и семантику его понятий. Однако анализ понятий является частью любой задачи независимо от ее области.



Объектное мышление приводит к естественной модульности программ, при которой между различными частями системы существуют ясные и естественные интерфейсы.

В объектном мышлении вопрос состоит не в чисто теоретических или программно-технических тонкостях. Это прежде всего практический образ мышления, который применим к решению различных проблем и проектированию разных систем. Особенно хорошо понятие объектов подходит для системного программирования и моделирования, в которых часто встречаются параллельные процессы. Фактически объектное программирование как раз и является методологией системного анализа и проектирования, которая кроме идей моделирования и проектирования содержит необходимые для практического использования язык представления данных и средства программирования.

Развитие объектного мышления и программирования

Объектное мышление было выработано в результате плодотворного взаимного влияния моделирования, представления данных и системного программирования. Первый раз понятия объектов и классов были использованы в языке Симула (Dahl et al. 1971). Класс (class) Симулы – это обобщение блока (block) Алгола. Как и блок, он содержал локальные переменные, определения процедур, а также основную программу. После этого путем ссылки на класс можно было

создавать соответствующие определению конкретные объекты (object instance). У каждого класса был характерный для него набор свойств (attribute), переменных и процедур. У каждого объекта были соответствующие классу свойства, но индивидуальное состояние (state) и управление (control state).

Объекты Симулы можно считать записями (record), которые кроме данных содержат и программу, и ее состояние. С другой стороны, его можно считать активизированным в процессе вычислений блоком языка с блочной структурой, который не исчезает после того, как его покинет управление, ожидая возможного возврата управления.

У объектов и классов Симулы, однако, были свои ограничения. Новые типы объектов (классы) нельзя было определять и нельзя было изменять набор свойств уже существующих объектов во время счета, т.е. после трансляции программы. Взаимодействие между объектами ограничивалось передачей управления и параметрами.

В Смолтолке (Goldberg и Kay 1976, Ingalls 1978, Goldberg и Robson 1982) состав понятий объектного подхода и методы программирования были развиты гораздо дальше. Язык и его реализация целиком построены на объектах и классах. Однако и в Смолтолке были свои ограничения и проблемы. Не совсем удачным был синтаксис языка, и не вполне эффективными были его реализации.

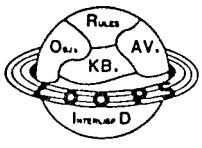
Серьезное содержательное ограничение Смолтолка состояло в том, что наследование свойств основывалось на древовидной иерархии, в которой объект мог принадлежать лишь одному надклассу. На практике объекты могут принадлежать нескольким различным классам в зависимости от того, с какой стороны их рассматривать. Например, морковь может быть растением, пищей или продукцией, корабль может быть средством передвижения или игрушкой и т.д. Вещи можно рассматривать с различных *точек зрения* (view) и с позиций их различных *ролей* (role).

В принципе в объектно-ориентированном программировании наряду с *объектной моделью* (object model) Смолтолка различают *модель акторов* (actors) Хьюита. Модели немного отличаются друг от друга порядком управления. Если в объектной модели объект обычно возвращает ответ *отправителю* (sender) сообщения, то в модели с исполнителями объект передает его следующему объекту, который продолжает вычисления.

В Лиспе объекты можно считать обобщением списка свойств. Списки свойств с самого начала использовали для сохранения как значений, так и определений функций. Первыми настоящими расширениями в направлении объектного программирования были, пожалуй, фреймовые формализмы Минского (Minsky 1975), многооконная система Интерлиспа (Teitelman 1977) и пакет EXTEND Маклиспа (Kerns 1980). Даже язык Ада в некоторой мере опирается на объекты (package).

Ада Байрон.

Особенно удачно объекты были реализованы в системах Flavors Зеталиспа (Weinreb и Moon 1981) и Loops Интерлиспа (Bobrow и Stefik 1983). В этих сделанных для Лисп-машин системах удалось решить и снять с повестки дня многие из проблем, стоявших перед более ранними попытками. Loops – это реализованный в Интерлиспе рабочий язык высокого уровня с многими парадигмами, объекты которого параллельно содержат единые средства процедурного, событийного и продукционного программирования.



В Зеталиспе объекты сначала использовались в оконной системе и в реализации поддержки одновременных сеансов. Окна были удачно определены в виде объектов, каждый из которых должен обладать формой, размером, позицией, именем, содержимым, способом изображения и другими данными о состоянии. Исследование состояния окон и работа с

ними производятся с помощью одних и тех же методов. Действия с окнами осуществлялись с помощью управляющих сообщений, передаваемых им с клавиатуры, от мыши или из программы, и приводящих к желаемым результатам. Однако вскоре оказалось, что ароматы универсальны – и их включили в язык.

Объектный подход в Лиспе быстро завоевал популярность как в системном, так и в прикладном программировании. В связи с объектами речь идет не о новом и многостороннем типе данных, а о другом способе организации программы и данных, в котором как пассивные данные, так и определенные для них действия объединяются в единое целое.

В Лиспе объекты можно реализовать с помощью списка объектов и применяющих функционалов (APPLY, FUNCALL). В более новых Лисп-системах средства объектного программирования уже содержатся в системе (DEFLAVOR, DEFMETHOD, SEND и другие). В определение Коммон Лиспа (Steele 1984) объекты не входят, но в системе они имеются.

Литература

1. Bobrow G., Stefik M. *The LOOPS Manual.* Xerox PARC, Palo Alto, California, 1983.
2. Cannon H.I. Flavors: A Nonhierarchical Approach to Object-Oriented Programming. Unpublished paper, AI Laboratory, MIT, Massachusetts, 1982.
3. Dahl O.-J., Myhrhaug B., Nygaard K. Simula 67 Common Base Language. Norwegian Computing Centre, S–22, Oslo, 1971.
4. Goldberg A. Introducing the Smalltalk-80 System. *Byte 6:8,* August,1981.
5. Goldberg A., Kay A. *Smalltalk-72 Instruction Manual.* Xerox PARC, Palo Alto, California, 1976.
6. Goldberg A., Robson D. *Smalltalk-80 – The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts,1982.

7. Hewitt C., Bishop P., Steiger R. A Universal Modular Actor Formalism for Artificial Intelligence. *In Advance Papers of the Third IJCAI–73,* Los Altos, William Kaufmann Inc., Palo Alto, 1973, pp. 235–245.
8. Ingalls D.H. The Smalltalk-76 Programming System: Design and Implementation. *Conference Record of the 6th ACM Symposium on Principles of Programming Languages,* Tucson, Arizona, 1978, pp. 9–16.
9. Kerns R.W. EXTEND Package for Maclisp. AI Laboratory, MIT, Cambridge, Massachusetts, 1980.
10. MacLennan B.J. Values and Objects in Programming Languages. *SIGPLAN Notices,* Vol. 17, No. 12, pp. 70–79.
11. Minsky M. A Framework for Representing Knowledge. In Winston P. (ed.): *The Psychology of Computer Vision.* McGraw-Hill, New York, 1975.
12. di Primio M., Christaller M. A Poor Man's Flavor System. Working Paper No. 47, Issco, Geneve, 1983.
13. Teitelman W. A Display Oriented Programmer's Assistant. *Proceedings of the 5th IJCAI,* Cambridge, Massachusetts, 1977, pp. 985–915.
14. Weinreb D., Moon D.A. *Lisp Machine Manual.* Symbolics Inc., Cambridge, Massachusetts, 1981.
15. Weinreb D., Moon A.M. Introduction to Using the Window System. AI Laboratory, MIT, Cambridge, Massachusetts, 1981.

Смысл вычислений заключен не в числах, а в подходе.

Р. Хэмминг

Смысл программирования заключен не в коде, а в понимании.

Авторы

2.7 ДОСТОИНСТВА И КАЧЕСТВО ПРОГРАММИРОВАНИЯ

- **Факторы качества и подходы к программированию**
- **Разделяй и именуй объекты естественным образом**
- **Используй хорошо определенные соединения**
- **Переносимость и стандартизация**
- **Другие советы**
- **Литература**

★ ★ ★ ★

Методы и стили программирования для различных задач применяются по-разному. Однако независимо от используемых стиля и метода существуют общие достоинства и принципы, о которых всегда полезно помнить.

В программировании содержатся также эстетические и практические ценности. Хорошую программу можно читать как стихи, и она может привлекать как песня, музыка или прекрасная картина. Каждый программист на своем опыте знает, что он ощущает каждый раз, когда программа начинает работать. Еще большее интеллектуальное удовлетворение можно получить, найдя хорошее решение трудной задачи. Наилучшие образцы программирования представляют собой творческую науку и искусство.

В этой главе мы рассмотрим общие подходы, влияющие на качество программ, а также перечислим считающиеся обычно полезными принципы и практиче-

ские приемы программирования. Эти правила и ограничения позволяют избежать плохого стиля программирования, но, с другой стороны, они не должны стать препятствием для творческого поиска новых решений, исследования и экспериментирования.

Факторы качества и подходы к программированию

Про факторы качества и цели программирования и программ много писалось, а также делалось много попыток их выделения и анализа. В общем виде под качеством программ понимается то, насколько хорошо они соответствуют установленным для них *требованиям* (specification) и сколь высоко установлена планка этих требований. Факторы требований и их вес в свою очередь могут меняться в зависимости от случая. В общем случае они даже иногда противоречат друг другу.

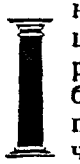
A-1

Увеличение размеров систем и усложнение программирования предъявляют все большие требования к качеству программ. Автоматизация программирования и автоматический синтез программ по описаниям также предъявляют требования к используемым языку и методам программирования.

В связи с этим нет возможности и необходимости детально различать вопросы качества и методов. Удовлетворимся перечислением наиболее важных качеств и связанных с ними подходов:

1. *Корректность* (correctness). В первую очередь программа должна правильно работать. Ошибочную программу невозможно использовать, даже если у нее будут другие полезные свойства. Доказательство и обеспечение корректности программ обычно тем более сложно, чем больше и сложнее сами программы. (Здесь надо отметить, что наибольшая работа в области автоматического *доказательства корректности* (proving) программ проделана именно в рамках Лиспа.)
2. *Надежность* (reliability). Программы искусственного интеллекта не всегда дают верные решения из-за наших недостаточных или ошибочных

знаний об области применения или из-за используемых в решениях эвристик, хотя технически программы и не содержат ошибок. Тесно связанным с корректностью свойством является надежность. Программирование приобретает все большую общественную значимость, и поэтому работа программиста может быть связана с большой ответственностью. Стоимость ошибки, проявившейся в программе обычно тем больше, чем позднее обнаружена ошибка и чем шире используется программа.



3. *Полезность* (validity). Верно работающая программа не найдет своего применения, если она не решает задачу пользователя и не соответствует его ожиданиям. Разработка программного обеспечения и его использование могут, например, оказаться экономически убыточными.
4. *Эффективность* (efficiency). Хорошо, когда программа функционально эффективна по памяти и времени выполнения. Вычислительная машина традиционно являлась критическим ресурсом. Однако эффективность программирования становится все более важным фактором. Как правило, сначала пытаются создать программу как можно быстрее и проще и лишь после этого сосредоточиваются на вопросах, связанных с эффективностью, если это вообще потребуется.

5. *Удобочитаемость* (readability) и понятность программ. Помимо автора и другие должны быть при необходимости в состоянии понять смысл программы и ознакомиться с текстом работающих функций (и прочей документацией). Обычно программист сам вскоре забывает принятые решения, и ему нужно вновь открывать их для себя. С увеличением среднего размера программ все большее их количество приходится создавать коллективными усилиями, чем еще больше подчеркивается значимость удобочитаемости.

6. *Тестируемость* (testability) и корректируемость. Тестирование программ и проверка их корректности – часто более объемная работа, чем их написание. Плохую программу нередко легче сделать заново, чем выполнить тестирование и коррекцию.
7. *Отлаживаемость* (debuggability). Почти во всех уже оттестированных программах позже находятся ошибки или непременно возникают какие-нибудь потребности внести исправления. Программы нужно создавать так, чтобы позднее было просто локализовать и исправить ошибки.
8. *Сопровождаемость* (maintainability). Программы часто нужно изменять и расширять, например в связи с изменениями во внешнем мире. У практических программ часто долгая история разработки и количество усилий, необходимое для сопровождения, возрастает по мере увеличения числа и размера используемых программ.
9. *Переносимость* (portability). Машины и технические средства развиваются и дешевеют быстрее, чем программы. Поэтому программное обеспечение должно быть легко переносимым на новые и более дешевые машины и из одной среды в другую.

10. *Адаптируемость* (adaptability) и модифицируемость. Изменение со временем требований к программам требует их легкой модифицируемости, учитывая различные цели использования и управляемость модификациями. Программные системы и их части полезно проектировать независимыми друг от друга и определять их интерфейсы четко и локально. Так программы получаются более общими, и их можно использовать как строительный материал в различных областях применения.

Обычно у программиста по опыту написания программ уже есть некоторое чувство действенности программ и их других качественных факторов. Однако это чувство интуитивно и его ощущение и надежность во многом зависят от предлагаемых языком средств, стиля

программирования, а также от основных используемых решений и методов.

Рассмотрим далее немного подробнее, какими средствами и принципами может достигаться качество. Приведенные ниже инструкции не подчиняются некоему единому стилю и не перечисляются в порядке их значимости. Это только приблизительный набор общих одобряемых всеми принципов и универсальных правил, хорошо зарекомендовавших себя на практике.

Разделяй и именуй объекты естественным образом

Достижение различных факторов качества программы во многом зависит от удачного выбора решения задачи и общей структуры программы. Это тем важнее, чем больше программа, о которой идет речь. Формирование программы и ее отдельных частей, понятность и управляемость можно улучшить, разбивая задачу на абстрактные единицы и естественные части и используя при этом для составляющих наглядные имена:

1. Пытайся разбить задачу на части естественным образом, объединяя при этом данные с той частью, которой они принадлежат. Пытайся отделить данные от кода программы и включить их в отображение проблемной области. Пытайся использовать общеприменимые методы и абстрагировать данные в соответствии с ними.

2. Не пиши функции длиннее, чем их можно вообразить и ощутить как единое целое (на листе А4 или в окне на экране). Лучше всего, если функции будут короткими и подходящим образом сгруппированными как с точки зрения логической последовательности, так и с точки зрения иерархии понятий. Используй уровни абстракции и образуй по мере необходимости подходящие комплексы из понятий.

3. Используй меткие и характерные имена, которые не взяты наугад, не слишком специфичны или общи, а которые характеризуют смысл или назначение. Читатель, изучая логику программы, не должен

запоминать специально, какое имя, смысл или значение имеет переменная. Предпочтительны имена и термины на естественном языке и близкие задаче, например МЕСТО-ВОЗВРАТА, ДОКАЗЫВАЕМОЕ-ПРЕДЛОЖЕНИЕ, ПРИЛАГАТЕЛЬНОЕ и т.д.

4. Придерживайся какой-нибудь системы в использовании имен. Например, динамически вычислимые переменные можно помечать признаком, скажем, звездочкой: *ГЛАСНЫЕ*, *БАЗА-ДАННЫХ* и т.д.
5. Не используй для различных объектов одинаковые или схожие имена, а для одного и того же объекта различные имена, даже если это технически возможно. Используй для аналогичных целей аналогичные имена.

Используй хорошо определенные соединения

Работа различных частей программы должна быть понятна вне зависимости от ее других частей, и взаимодействие должно происходить через хорошо определенные *интерфейсы* (interface) и *протоколы* (protocol).

ОК При рассмотрении маленькой части кода программы сразу должна быть ясна ее роль и значение с точки зрения всей программы как целого, т.е. программы должны быть *прозрачными* (transparent). Работу функции или других объектов желательно понимать и иметь возможность управлять ею без знаний о работе или состоянии других объектов и зависеть лишь от эффекта определяющей формы. Еще некоторые советы:

1. Пытайся разбить задачу так, чтобы взаимодействия ее частей минимизировались, и при разбиении факторы, влияющие на состояние и поведение отдельных частей, предпочтительнее оставлять внутри их, чем снаружи.
2. Не передавай данные присваиванием их переменным вне функций и других объектов, когда это не является естественным, а используй параметры и передавай данные через предоставляемые определяющими формами средства и механизм вызова.

3. Избегай решений, которые приводят к значительным побочным эффектам.

Переносимость и стандартизация

Пытаясь достичь переносимости, нужно избегать использования в программировании функций и механизмов конкретной системы. Между некоторыми системами существуют преобразователи (в Зеталиспе, например, Interlisp Compatibility Package). Однако преобразования не покрывают весь язык и не всегда правильно действуют.

Коммон Лисп – это важный шаг в сторону стандартизации Лиспа, совместимости систем и переносимости программного обеспечения, но на практике используются и будут в дальнейшем использоваться различные версии и расширения языка. Коммон Лисп – это не последнее слово, хотя в его определении и предусмотрены расширения.

Другие советы

Далее приведем еще некоторые универсальные правила, касающиеся конкретных форм, и советы для практического программирования:

1. При записи условий используй формы, которые лучше всего подходят для выражения смысла. Например, предикат NOT больше подходит для логической проверки, чем NULL, хотя результат будет одинаков. В свою очередь NULL больше подходит для проверки на пустой список. Проверка на непустой список естественнее осуществлять предикатом (NOT (NULL x)), чем предикатом (CONSP x) или x. (Хороший транслятор будет транслировать эти формы в одинаково эффективный машинный язык.)

2. Избегай использования псевдофункций SETQ внутри определений функций. Лучше используй формы LET и DO, с которыми связаны механизмы определения, доступности и изменения значений

переменных. Переменные получают значения всегда в одном и том же месте и после использования связи автоматически обрываются. Если используешь SETQ, то прокомментируй его побочный эффект.

3. Избегай использования разрушающих функций (RPLACA, NCONC и другие), если ситуация не является критической в отношении времени или памяти.
4. Не используй предикаты AND и OR вместо условных выражений.
5. Предпочитай формы LET и DO формам LET* и DO*, в связи с которыми нужно учитывать побочный эффект.
6. Не используй предложение PROG, если можно использовать более наглядную форму, например DO.
7. Не используй переменные DO в качестве константы, а задавай их внешней формой LET.

Следующие обстоятельства полезно запомнить для облегчения поиска и исправления ошибок, а также для облегчения внесения изменений, возникающих в процессе сопровождения:

1. Используй короткие функции и создавай определения, не зависящие от окружения момента вызова. Таким образом, их можно до включения в систему отдельно проверять вручную и на машине, а также без усилий использовать их в составе различных систем.
2. В определениях обращай внимание и на "невозможные" случаи и поясняй ошибки в комментариях. Например, специально проверяй в предложении COND все возможные ситуации и в качестве условия последней ветви используй предикат T, а в качестве ее результата – сообщение об ошибке.

3. Аргументы функции задавай в логической последовательности. Если у нескольких функций один и

тот же аргумент, то пусть он будет первым или последним аргументом. Если одинаковых аргументов больше, то задавай их всегда в одном и том же порядке.

4. Если у действия функции или у ее аргументов предполагается наличие некоторых свойств, то опиши их явно в комментарии.
5. Комментируй программу разумно и аккуратно. Используй комментарии как внутри определений, так и снаружи так, чтобы с их помощью отразить всю структуру программного обеспечения. Специально фиксируй внимание на исключениях и сложных местах.

Стилю программирования лучше всего учиться программируя.

Литература

1. Brooks R. How to Hack Lisp Real Good. Lecture Notes, Stanford University, Computer Science Department, 1984.
2. Ершов А.П. О человеческом и эстетическом факторах в программировании. *Кибернетика*. No. 5, 1972.
3. Knuth D. *The Art of Computer Programming. Vol. 1., Fundamental Algorithms*. Addison-Wesley, Reading, Massachusetts, 1968. [Имеется перевод: Кнут Д. Искусство программирования. Т.1. Основные алгоритмы. – М.: Мир, 1976.]

3. СРЕДСТВА И СРЕДА ПРОГРАММИРОВАНИЯ

3.1 ПЕРВИЧНАЯ СРЕДА КОММОН ЛИСПА
3.2 СРЕДА ИНТЕРЛИСПА
3.3 СРЕДА ЗЕТАЛИСПА

В этой главе мы познакомимся с лисповской средой программирования. Коммон Лисп определяет для среды программирования лишь граничные условия, давая возможность принимать в реализации среды различные решения. Более детальные вопросы было решено оставить открытыми, поскольку среды программирования вместе с их характеристиками и средствами находятся еще в стадии интенсивного исследования.

Далее мы изложим определенные в Коммон Лиспе для среды граничные условия и основные компоненты. После этого в качестве примера мы рассмотрим лисповские среды программирования систем Интерлисп и Зеталисп. Интерлисп представляет собой классический пример интегрированной среды, которая первоначально разрабатывалась для системы разделения времени вычислительных машин DEC-10/20 и которая после этого в документированном виде используется на Лисп-машинах фирмы Xerox. Зеталисп происходит из системы Маклисп, разработанной в MIT для той же машины DEC-10/20. Система получила существенное развитие, и она используется на Лисп-машинах подобных Маклиспу (LMI, Symbolics, TI, Sperry, нынешний Unisys и другие системы).

Обстоятельства меняются,
принципы нет.

О. Бальзак

3.1 ПЕРВИЧНАЯ СРЕДА КОММОН ЛИСПА

- **Аппаратная среда реализаций языка**
- **Составные части среды программирования на Лиспе**
- **Интегрированность и прозрачность**
- **Редактирование программ: ED**
- **Тестирование программ: TRACE и STEP**
- **Прерывание вычислений: BREAK и ERROR**
- **Трансляция программ: COMPILE**
- **Система документирования и справочная система**
- **Комментарии**
- **Средства определения количественных характеристик вычислений**

Аппаратная среда реализаций языка

Лисп-системы доступны для достаточно различных аппаратных средств, начиная с 8 битовых микро-ЭВМ и кончая суперЭВМ и специализированными Лисп-машинами. Раньше разработку программ на Лиспе осуществляли с помощью систем разделения времени на больших машинах, особенно на машинах серии DEC-10/20. В настоящее время акцент смещается на персональные рабочие станции и Лисп-машины, которые не без основания можно считать специализированными вычислительными машинами, учитывая свойства языка Лисп и требования, предъявляемые исследованиями в области программирования задач искусственного интеллекта.

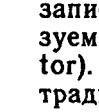
В Коммон Лиспе пытаются отделить влияние аппаратного окружения от самого языка Лисп. Исходя из этого окружения определяются лишь стандартные имена устройств, логические механизмы работы с

файлами и внешними устройствами. Таким образом, предпринята попытка обеспечить как возможность развития аппаратных средств, так и переносимость программ, написанных на Коммон Лиспе из одной среды в другую.

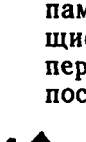
Составные части среды программирования на Лиспе

Кроме внешних устройств с Лисп-системой связан набор системных программ, таких как операционная система и специальные вспомогательные средства для программирования, которые в большей или в меньшей степени использованы в подсистемах языка. Рассмотрим далее наиболее важные из них в порядке их использования на этапах разработки программ:

1. Этап определения. Программы (определения DEFUN, присваивания SETQ и прочие) сначала записываются в файл обычно с помощью используемого на вычислительной машине *редактора* (editor). Это происходит таким же образом, как и в традиционных языках. Вообще-то редактор можно вызвать непосредственно из интерпретатора Лиспа, в этом случае из интерпретатора можно перейти в редактор и обратно без промежуточного использования операционной системы.



С точки зрения работы редактора Лисп-системы бывают двух основных типов: (1) работающие в памяти или *резидентные* (resident) и (2) работающие с файлом или *загружаемые* (loadable). В первом случае функции можно тестировать сразу после их определения. Во втором случае функцию (или функции) нужно сначала загрузить в память с помощью специальной директивы (LOAD). Загрузка может производиться и автоматически, когда из редактора переходят в интерпретатор (Маклисп).



2. Этап тестирования. Если определения были синтаксически верны и сообщения об ошибке не последовало, то можно начать тестирование их работы.

Поиск ошибок вычисления можно проводить с помощью *трассировщика* (trace), который выводит имя и значения аргументов отслеживаемой функции каждый раз, когда она вызывается. Другой возможностью является использование *пошагового исполнения* (stepping), когда вычисления выполняются шаг за шагом под наблюдением пользователя. Найденные таким образом ошибки можно затем исправить редакторомлибо непосредственно в памяти или в файле в зависимости от типа системы.

Некоторые Лисп-системы позволяют исправлять ошибки во время выполнения программы. Когда программа прерывается на ошибке или пользователь сам прерывает ее, управление передается *функциям прерывания* (break). В состоянии прерывания можно изучать и изменять значения переменных или вызывать редактор для исправления (debug) определений функций, после чего вычисления можно продолжить, не производя их заново. Таким образом экономится машинное время, и уже сделанные вычисления не пропадут.

3. Трансляция. Если программа написана и оттестирована, то ее можно отдать на трансляцию *транслятору* (compiler) Лиспа. Транслятор создает вариант функций или программы на машинном языке, который обычно требует меньше памяти и работает значительно быстрее, чем интерпретируемая версия. Разница, как правило, составляет один порядок, но зависит, естественно, от транслируемой программы и от качества транслятора. Эффективно транслируются макросы, а также некоторые специальные виды рекурсии, например *концевая рекурсия* (tail recursion). В Коммон Лиспе трансляция более важна, чем это имеет место обычно, поскольку есть возможность оптимизировать сильно абстрагированные формы.

В добавок к основным действиям в расширенных системах используются различные более развитые

вспомогательные средства и инструменты, облегчающие программирование. Например:

1. Активно помогающий программировать и дающий справки *ассистент* (programmer's assistant) или *подмастерье* (apprentice).
2. Автоматический *корректор ошибок* (Do What I Mean – not what I type), который в состояние исправлять простые опечатки и синтаксические ошибки[1].
3. Протокол диалога, или так называемая *история* (history list), посредством которой можно сослаться на более ранние реплики как пользователя, так и системы, отменить или вновь выполнить предшествующие действия и т.п.

4. *Инспектор* (inspector), который позволяет исследовать сложные объекты в наглядном виде.
5. *Анализаторы* программы и файлов, которые могут исследовать взаимоотношения определений функций и их вызовов и генерировать документацию.
6. Интерактивные *справочные системы* (on-line documentation).

Интегрированность и прозрачность

Среда программирования образуется из набора различных вспомогательных средств программирования и управления данными. Различные средства часто объединены в *интегрированную* (integrated) рабочую среду, которую трудно отличить от операционной системы. Под интегрированностью среды понимается то, что вспомогательные средства системы можно вызывать друг из друга, не обращаясь в промежутке к операционной системе. Операционная система обычно пользователю не видна и подсистемы

[1] Подсистема "Делай то, что я имею в виду" или DWIM Интерлиспа представляет пользователю весьма тонкую помощь, угадывая верное положение скобок в формах и многое другое, а не только исправление опечаток. *–Прим. ред.*

друг для друга *прозрачны* (transparent). Если, например, выполнение программы прервалось на ошибке, то интерпретатор Лиспа автоматически переходит на обработку ошибки и функции прерывания. Отсюда можно снова вызвать интерпретатор Лиспа, например, для вычисления значений или определения переменных и функций в окружении, где возникла ошибка, после чего можно запустить редактор для исправления ошибки и продолжить выполнение исправленной программы с места ее прерывания.

Часто работу среды невозможно отличить от самого языка Лисп. С точки зрения пользователя, услуги, предлагаемые языком и системой, так же как и определенные им самим функции, являются частями единой среды. Например, в некоторых Лисп-машинах практически все окружение Лиспа реализовано на нем самом, и система в общей сложности использует тысячи различных функций и действий.

Хотя Коммон Лисп и не определяет свою программную среду, однако он содержит некоторые входящие в состав среды функции и формы вызова подсистем. Далее мы познакомимся с наиболее важными из них.

Редактирование программ: ED

Для *редактора* (editor) в Коммон Лиспе определена лишь форма его вызова:

(ED &OPTIONAL *объект***)**

Объект в вызове можно задать с помощью необязательного параметра, и он может быть, например, переменной, функцией, структурой или файлом. Вызов ED без параметра возвращает в состояние, в котором была прекращена предыдущая редакция.

В Лисп-системах используются разнообразные редакторы. Лисповские функции и программы естественно можно в любой момент редактировать с помощью обычного текстового редактора, который имеется в ЭВМ. Однако из-за наличия у объектов внутреннего строения предпочтительнее

использовать так называемые структурные редакторы (structure editor), которые знают синтаксис языка. Структурные редакторы также различны. Редактор может позволять свободное передвижение по экрану или это передвижение может быть связано с логической структурой списка. В обоих случаях редактор может автоматически позаботиться о соответствии скобок и о том, чтобы отступ логически соответствовал синтаксису языка. Интеллигентные структурные редакторы могут содержать и некоторые другие удобные автоматические действия.

Тестирование программ: TRACE и STEP

Для тестирования программ в систему обычно входит функция TRACE (или подобная ей). С помощью трассировки (trace) можно отследить вычисление тестируемой функции (или функций) с целью локализации и исправления (debug) ошибок. Если некоторая функция помечена как трассируемая, то система информирует об имени функции и значениях аргументов каждый раз при входе в функцию и соответственно выводит значение функции, как только оно вычислено. Трассировка включается с помощью следующего вызова:

(TRACE &REST *функции***)**

Например:

```
_(trace +)          ; аргументы не вычисляются
(+)                 ; трассируемые функции
```

После этого интерпретатор трассирует вычисление функции +:

```
_(+ (+ 1 2) 3)
+:                  ; имя функции
arg1 = (+ 1 2)      ; аргументы
arg2 = 3
```

```
    +:                ; вложенный
    arg1 = 1          ; вызов
    arg2 = 2
    + = 3     ; значение вложенного вызова
+ = 6         ; значение внешнего вызова
6
```

Трассировку можно отменить директивой UNTRACE:

```
_(untrace member)
(MEMBER)
```

Вычисление формы можно проследить и шаг за шагом (single step) интерактивно с помощью директивы STEP:

(STEP *форма***)**

Форма теперь будет вычисляться по одному подвыражению за один раз. При вычислении каждой подформы интерпретатор напечатает вычисляемое выражение и остановится. В этом состоянии программист может в диалоге исследовать и изменить состояние вычислений, перейти из одного состояния в другое, запустить интерпретатор Лиспа, редактор и другие средства, интегрированные (включенные) в систему.

Прерывание вычислений: BREAK и ERROR

Наряду с формами TRACE и STEP важным элементом прослеживания и идентификации ошибочных ситуаций является BREAK·

(BREAK &OPTIONAL *сообщение***)**

BREAK – это форма, которую в процессе поиска ошибки можно поместить в критическую точку программы.

Когда вычисление доходит до места вызова, выводится сообщение и вычисления под управлением подсистемы исправления ошибок приостанавливаются. После этого прерванное состояние можно изучить, сделать необхо-

димые исправления и в зависимости от ситуации продолжить вычисления, в случае чего уже сделанные вычисления не теряются.

Однако для выдачи информации о более серьезных ошибочных ситуациях (error signaling) и для вызова ошибки из самой программы вместо функции BREAK используются функции ERROR и CERROR. Форма вызова ERROR выглядит так:

(ERROR *образец* **&REST** *аргументы***)**

С точки зрения вывода функция работает аналогично функции FORMAT, однако вычисления прерываются. Например:

```
_(defun reverse1 (список)
   ;; переворачивание списка
  (do
    ((остатки список (rest остатки))
     (результат (list (car список))
       (cons (first остатки) результат)))
   ;; условие окончания
    ((null остатки) результат)
    (if (atom остатки)
      (error "Неверная списочная форма ~S
        в функции REVERSE1." список))))
REVERSE1
_(reverse1 '(a b c))
(C B A)
_(reverse1 '(a b . c))
Неверная списочная форма (A B . C) в функ-
ции REVERSE1.
break>          ; приглашение в состоянии
                ; прерывания
```

Вызванное функцией ERROR ошибочное состояние относится к серьезным ошибкам (fatal error), и из него нельзя после исправления продолжить вычисления. В Коммон Лиспе определена и функция CERROR (continuable error). Используя ее, в прерванном состоянии

можно исправить ошибку и продолжить вычисления, как это делается функцией BREAK.

Трансляция программ: COMPILE

Обычно программы можно разрабатывать в интерпретирующем режиме. В этом случае функции используются в виде своих символьных списочных вариантов, трассировка и исправление которых делается легче. Когда программы уже не содержат ошибок, их можно оттранслировать.

Транслятор – это лисповская функция, которую можно вызвать следующими формами:

```
(COMPILE fn)          ; трансляция функции
(COMPILE-FILE файл)   ; трансляция файла
```

Предупреждение: не во всех диалектах Лиспа транслированные и интерпретируемые программы обязательно работают одинаково. Некоторые интерпретаторы пользуются динамическим вычислением, в то время как их транслятор может быть статическим! В Коммон Лиспе этих проблем нет.

Система документирования и справочная система

Многообразная среда Лисп-системы может иметь огромные размеры. Размер системного кода Лисп-машины, например, исчисляется мегабайтами. Таким образом и количество необходимой документации оказывается большим, а ее использование трудным. Для решения этой проблемы понадобилась разработка специальных интегрированных систем документирования и справочных систем, с помощью которых пользователь может легко получить из системы информацию по некоторому вопросу.

Даже в стандарте Коммон Лиспа учтены требования документирования и возможности для разработки поддерживающих его систем. Во время определения лисповского объекта с ним можно связать текст (documentation string), описывающий его использова-

ние и значение. Поясняющий текст (документация) синтаксически располагается в месте определения и ограничивается с двух сторон двойными кавычками, например:

```
_(defun второй (x)
    "Возвращает второй элемент списка"
    (cadr x))
ВТОРОЙ
```

В определении функции (DEFUN) текст следует за лямбда-списком и перед телом. Кроме формы DEFUN пояснения можно добавлять и в формы DEFVAR, DEFCONSTANT, DEFMACRO и DEFSTRUCT. Документацию, связанную с символом, можно прочитать с помощью специальной функции:

(DOCUMENTATION *символ тип***)**

Параметр ТИП указывает, какую из связанных с символом документаций нужно прочитать. У одного и того же символа могут быть разные описания как переменной (DEFVAR) так и функции (DEFUN), которые можно прочесть, если задать тип VARIABLE и FUNCTION.

```
_(documentation 'второй 'variable)
NIL
_(documentation 'второй 'function)
Возвращает второй элемент списка
```

Все известные системе описания, связанные с символом, можно программно прочитать с помощью формы:

(DESCRIBE *объект***)**

Например:

```
_(describe 'второй)
It is the symbol ВТОРОЙ ; вид объекта
Package: User  ; пространство имен символа
Value: unbound          ; значение символа
Function: interpreted   ; тип функции
Возвращает второй элемент списка
                        ; пояснение
```

Встроенные функции и другие системные объекты описаны заранее:

```
_(describe 'second)
It is the symbol SECOND
Package: SYSTEM
Value: unbound
Function: compiled
SECOND list
This function returns the second element
of the list
```

Наконец, упомянем еще весьма полезную в работе функцию ассоциативного запроса APROPOS:

(APROPOS *строка***)**

APROPOS возвращает все известные системе символы, в именах которых содержится строка знаков, являющаяся параметром. С ее помощью можно, например, восстановить имя функции, точное имя которой забылось. Например, вызов:

```
_(apropoc "pri")
(PRINC PRINT PRIN1 TERPRI ...)
```

выводит все известные системе символы, в имени которых содержится "pri".

Комментарии

Кроме задаваемых при определении пояснений (documentation), которые обрабатываются системой, к программе можно свободно добавлять пояснения и инструкции, или *комментарии* (comment), предназначенные для пользователя, читающего программу.

В Коммон Лиспе комментарий отделяется от остального текста программы точкой с запятой ";". При использовании комментариев в Коммон Лиспе рекомендуется следующая практика, которую поддерживают редакторы, знающие структуру языка, и программы структурной печати:

1. Комментарий, следующий за текстом программы на той же строке, отделяется одним символом ";".
2. Комментарий, находящийся внутри функции и занимающий отдельную строку, помечается двумя символами ";;" и начинается с отступа, соответствующего логическому уровню, к которому относится комментарий.
3. Комментарий, находящийся вне определения функции, т.е. внешний по отношению к функции, помечается тремя символами ";;;" и записывается, начиная с левого края.

Приведем пример комментирования программы:

```
;;; Комплексная арифметика

(defun kplus (x y)
   ;; сумма комплексных чисел x и y
   (list
      (+ (first x)        ; действительная
         (first y))       ; часть
      (+ (second x)       ; мнимая часть
         (second y))))
```

Средства определения количественных характеристик вычислений

Обычно Лисп-системы содержат средства, используя которые можно измерить затраченное на вычисление время и узнать, например, количество новых списочных ячеек, созданных за время вычисления. Само создание в Лиспе списочных ячеек (CONS) требует времени и, кроме того, добавляет работы мусорщику. В Коммон Лиспе время, необходимое для вычисления формы, можно получить с помощью функции:

(TIME *форма***)**

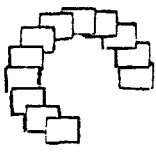
```
_(setq x '(1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
_(time (append x x x x x))
CPU Time: 0.01 sec., Real Time: 0.07 sec.
(1 2 3 ...)                ; значение
```

Целое – это больше, чем сумма частей.

Аристотель

3.2 СРЕДА ИНТЕРЛИСПА

- **Списочный редактор – List Editor**
- **Ассистент программиста – Programmer's Assistant**
- **Структурная печать – Prettyprint**
- **Прерывания – Break Package**
- **Прерывание вычислений и трассировка**
- **Работа с файлами – File Package**
- **Транслятор – Compiler**
- **Анализатор программы – Masterscope**
- **Справочная система – Help System**
- **Исправление ошибок – Do What I Mean**
- **Лисп с фразовой структурой – Conversational Lisp**
- **Оконная система – Window System**
- **Целостность системы – System Integration**
- **Библиотека программ – Lispusers Package**
- **Литература**



Система Интерлисп наряду с Зеталисп занимает ведущее место среди лисповских сред программирования. Многие разработанные в связи с Интерлиспом идеи и методы нашли применение в других Лисп-системах и машинах и уже применяются или начинают применяться в более новых микроЭВМ, рабочих местах и средах программирования для более привычных языков.

Интерлисп не был заранее спроектирован, а развивался в течение длительного времени на основе потребностей, возникавших при различных исследованиях.

Прежде всего в развитии системы нужно отметить роль таких организаций, как MIT, BBN и Xerox PARC. Интерлисп реализован на многих различных вычисли-

тельных машинах. Наиболее популярной реализацией была Interlisp-10, которая работала сначала под управлением операционной системы Tenex, а позже – под TOPS на машинах PDP-10 и DECsystem-20, а также Interlisp-D на Лисп-машинах серии 1100 фирмы Xerox.

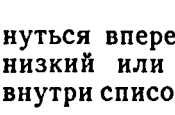
Предлагаемая Интерлиспом среда программирования характеризуется как дружественная, готовая к помощи и совместным усилиям, терпимая к ошибкам. Подсистемы системы образуют открытое целое. Такие подсистемы, как редактор, легко можно расширить и использовать, например, в виде части некоторой прикладной системы. Несмотря на свой состав и сложность, Интерлисп на практике оказался особенно надежным, и он основательно документирован. Характеристики системы достаточно устоялись, и ее новые версии совместимы с более ранними.

Далее рассмотрим различные компоненты системы и оказываемые ими пользователю услуги и процедуры.

Списочный редактор – List Editor

Редактор Интерлиспа является так называемым структурным редактором, который знает синтаксис списочных структур и обрабатывает непосредственно исправляемый объект, а не его копию. Поэтому он может с одинаковым успехом работать как с данными, так и с программой, т.е. с функциями, не выходя при этом за пределы Лисп-системы.

Исправление выражений, являющихся данными, определений функций, списков свойств и других объектов осуществляется с помощью вызова редактора с исправляемым объектом в качестве аргумента, передвижения в выражении на нужное подвыражение и осуществления после этого необходимых преобразований. Передвижение к нужному месту происходит через директивы позиционирования. С их помощью можно передвинуться вперед и назад, а также спуститься на более низкий или подняться на более высокий уровень внутри списочной структуры. Выражение можно найти



и по его форме или содержанию. Директивы редактирования можно задавать по одиночке или по нескольку на одной строке, разделяя их между собой пробелами.

Редактирование осуществляется во внутренней символьной форме списочных структур, что облегчает передвижение внутри списка, преобразование выражений, добавление элементов, удаление и их исправление без боязни, что скобки перепутаются. Списочный редактор заботится о сбалансированности скобок.

В версиях Интерлиспа, работающих в системе разделения времени, редактор работает построчно, но в Лисп-машинах в системе Interlisp-D используются более развитые методы указания и выбора приказов, основывающиеся на использовании мыши и меню.

Внутри редактора рекурсивно (на новом уровне) может быть использована (nested call) вся среда Интерлиспа, например: интерпретатор Лиспа, функции прерывания и даже сам редактор. Так, например, можно эффект исправления проверить непосредственно из редактора без необходимости вернуться на предыдущий уровень под управление интерпретатора Лиспа. Поскольку редактируемые структуры данных находятся в памяти и редактирование осуществляется физически на самом определении, никакие специальные этапы трансляции, загрузки или им подобные перед тестированием не нужны. Если в процессе тестирования встречается ошибка и это приводит к прерыванию, то можно вновь вызвать редактор с более низкого уровня, внести новые исправления и продолжить вычисления с прерванного места на более высоком уровне.

Определения сохраняются в файле функцией MAKEFILES. На основе поддерживаемой системой истории MAKEFILES запрашивает пользователя, какое выражение и в какой файл нужно записывать, и обновляет файлы только в изменившейся части. Поддерживается информация о содержимом различных файлов и о расположении (layout) в них данных, редактируя которые обычным образом, можно влиять на содержимое и расположение файлов.

Ассистент программиста – Programmer's Assistant

Взаимодействие между программистом и системой осуществляется через ассистента программиста (PA). Основная идея PA состоит в том, чтобы программист чувствовал, что ведет диалог не столько с бесстрастно выполняющей приказы машиной, сколько с активным помощником и посредником всей системы, который пытается облегчить и сделать более эффективной работу пользователя.

PA, например, запоминает реплики диалога и дает возможность в последующем сослаться на них. Например, с помощью директивы ?? можно вывести историю диалога (history list). В соответствии с выведенным списком программист может повторить ранее выполненный приказ или группу приказов (REDO). Он также может отменить (UNDO) произведенный ими эффект, будь то хоть загрузка целого файла в систему, т.е. сделать так, чтобы выполненное стало снова невыполненным.

На самом деле существуют две истории, одна – для интерпретатора Лиспа и вторая – для диалога с редактором (editor). Если, например, та же самая последовательность директив редактирования необходима два или большее число раз, то ее можно повторить с помощью истории. Например:

```
*?? FROM F          ; вывести историю, начиная
                    ; с директивы F
15. *F PUTD         ; история
16. *(1 MOVD)
17. *3
18. *(XTR 2)
19. *0
20. *(SW 2 3)
*REDO FROM F        ; повторяются директивы 15-20
```

Последняя директива REDO в свою очередь заносится в историю в виде события под номером (21), и его можно теперь повторить просто директивой REDO.

У пользователя есть возможность самому определить длину истории, в соответствии с которой события более давние могут быть забыты ассистентом. На события можно сослаться по абсолютному или относительному порядковому номеру или, как ранее, по содержимому. Директива FIX вызывает редактор, с помощью которого можно исправить событие из истории перед его новым выполнением.

Поскольку у интерпретатора и у редактора свои собственные списки истории, то исправления, оказавшиеся позже ошибочными, можно легко отменить. После окончания редактирования весь его результат сохраняется в виде одного события в списке истории интерпретатора. Это дает возможность отменить редакцию и позже в течение сеанса работы.

Ассистент программиста использует и другие данные в диалоге, программах пользователя и системных ситуациях, которые он собирает и поддерживает. РА по-разному использует эти данные и предоставляет их другим компонентам системы.

Структурная печать – Prettyprint

В Интерлиспе используется структурная печать Prettyprint (PP), которая выводит определения функций и другие сложные списочные структуры в аккуратном и ясном виде с соответствующими отступами. Списки выводятся лишь до определенной глубины или длины таким образом, что более глубокие структуры заменяются на знак &, а более длинные хвосты – на "...". Это помогает охватить структуру сложных списков и сосредоточить внимание на существенном.

Со структурной печатью связаны специальные шрифты и поддерживающее их использование программное обеспечение (Font Package). Их использование предполагает наличие подходящего оборудования, обычно лазерного принтера.

Прерывания – Break Package

Система прерываний (Break Package) начинает работать при возникновении ошибки или когда пользователь сам прерывает вычисления. Состояние, в котором прерваны вычисления, теперь можно изучать, отследить в обратном порядке приведшую к нему цепочку вызовов и просмотреть их контекст. В ходе этой работы можно просмотреть всю Лисп-систему, но на новом уровне. Можно, например, снова присвоить значения переменным (SETQ), определить новые функции или действия и внести другие изменения в контекст. Так же как и вне прерывания, можно вычислить произвольное выражение. Возможно, при этом возникнет новая ошибка, в этом случае управление будет передано на новый уровень прерывания, предыдущее состояние остается в состоянии ожидания и сохраняется до тех пор, пока к нему не произойдет возврата. Так за время сеанса могут возникнуть несколько наложенных уровней прерывания.

Во время прерывания можно легко исправить прерванное место в программе, поскольку интерпретатор Лиспа и редактор интерпретируют и редактируют одну и ту же физическую структуру программы. Обычно достаточно ввести в состоянии прерывания директиву EDIT, после чего система вызывает редактор, устанавливает текущую позицию на место, вызвавшее прерывание и ждет директив редактирования.

С точки зрения вычислений прерывание выглядит как вызов функции. Оно всегда возвращает некоторое значение, так называемое значение прерывания, после чего вычисление продолжается с прерванного места так, как будто бы прерванная форма возвратила в качестве значения значение прерывания. Если, например, вычисление прервалось на неопределенной переменной, то можно из прерывания пойти дальше, возвращая для переменной некоторое значение.

Прерывание вычислений и трассировка

Система прерываний позволяет устанавливать прерывание по инициативе пользователя и отслеживать ход вычислений. Для выбранной функции можно включить прерывание с помощью директивы BREAK, тогда каждый раз при вызове функции вычисление будет прерываться. Включение трассировки осуществляется директивой TRACE. Во время трассировки аналогично Коммон Лиспу при входе в функцию всегда выводятся имя функции и значения аргументов, а при выходе из нее – значение функции.

Работа с файлами – File Package

Интерлисп – это система резидентного (resident) типа или система с рабочим состоянием (workspace) так же, как, например, APL. В начале сеанса из файлов загружаются необходимые программы, т.е. функции, переменные со значениями и т.п., после чего все находится в рабочем состоянии, выход из которого осуществляется лишь при завершении сеанса. Все нужные для программирования инструменты и вспомогательные средства включены в систему так, что во время сеанса не нужно возвращаться к операционной системе.

При загрузке функций, переменных и других объектов в систему, объекты становятся как бы частью Лисп-системы в этом сеансе и образуют персональное расширение системы пользователя, которое может содержать даже новое определение самой системы. В конце сеанса (или в произвольный момент) новые функции можно сохранить в файлах или снять копию целиком с состояния всей системы.

Программист может не заботиться о форме записи или расположении функции в файле (хотя и это возможно). Достаточно описать для каждой функции файл, в котором она сохраняется. Файловая система учитывает, какие объекты хранятся в каждом файле, в каких файлах и в каком месте они находятся и когда их в послед-

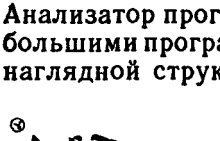
ний раз редактировали. Во многих ситуациях система в состоянии добраться до отсутствующего объекта без явного вызова и позаботиться о том, чтобы сделанные изменения были сохранены. Необходимые данные она получает от редактора и ассистента программиста. Так программист во многом освобождается от файловой бухгалтерии, которая в больших применениях весьма быстро разрастается в трудную проблему.

Транслятор – Compiler

Система Интерлисп, как и любая мало-мальски хорошая система, содержит транслятор. С его помощью можно транслировать как файлы, так и отдельные функции из рабочего состояния системы. Результатом трансляции можно непосредственно воспользоваться или сохранить в файле для дальнейшей загрузки. Транслятором можно транслировать функции частично, и в одной и той же программе могут одновременно быть как интерпретированные, так и транслированные функции. Транслятор может вызываться более развитыми функциями работы с файлами (например, MAKEFILE), который автоматически обновляет файл с исходным текстом на Лиспе и создает его транслированную версию. Используются и функции, непосредственно вызывающие транслятор (COMPILE и другие).

Анализатор программы – Masterscope

Анализатор программ (Masterscope) упрощает работу с большими программами. С его помощью можно сделать наглядной структуру программ, прояснить взаимные ссылки, кто кого вызывает, где переменные получают значение, где они используются и т.д. У анализатора с одинаковым успехом можно запрашивать данные как о рабочем состоянии, так и о программах, находящихся в файлах. Из результатов анализа составляется база данных. Если функции изменяются, то анализатор узнает об этом от редакто-



ра и управления файлами и необходимый новый анализ будет осуществлен по собственной инициативе.

Справочная система – Help System

Справочное руководство по Интерлиспу целиком содержится в базе данных, из которой с помощью справочной системы (Help System) можно делать запросы. Запросы можно делать четкими повелительными предложениями простой структуры. Например:

```
!TELL ME ABOUT функция
...
!WHAT ARE THE ARGS FOR функция
...
!TELL ME ABOUT THE LAST ARGUMENT OF функция
...
!MORE
...
!OK
```

В системе есть указатель в виде списка к справочному руководству, с помощью которого она добирается до подходящих пояснений в базе данных.

Исправление ошибок – Do What I Mean

В результате проведенных исследований, выяснилось, что большая часть ошибок в программах – это просто ошибки написания, такие как неверная буква или знак, изменение порядка двух знаков, ошибка в регистре, пропуск элемента, ошибка в скобках и т.д. Для автоматического исправления синтаксических ошибок такого типа в Интерлиспе существует уникальная система DWIM (Do What I Mean (not what I type)). При возникновении ошибки всегда автоматически запускается DWIM.

Исправление ошибок основывается на данных, имеющихся о программе, на информации о предыду-

щих действиях пользователя (списке истории), а также самого состояния, в котором проявилась ошибка. Например, вероятность ошибки в написании оценивается путем вычисления для активных имен их относительных оценочных чисел на основе длины имен и отличий в знаках. Простые и совершенно явные ошибки система DWIM исправляет самостоятельно, оповещая лишь о сделанных исправлениях и сразу же продолжая вычисления. В более сомнительных ситуациях она обращается за подтверждением к пользователю. Следующий вопрос, например, свидетельствует о том, что DWIM встретила в функции ФАКТОРИАЛ неизвестную функцию с именем COOND и задает вопрос:

```
COOND IN ФАКТОРИАЛ -> COND ?
```

На него можно ответить YES, и в этом случае DWIM исправит ошибку, или NO, в этом случае вычисления продолжаются без исправления и возникает прерывание. Если программист ничего не ответит, то DWIM немного подождет и продолжит работу, предполагая, что молчание – знак согласия. Если программист хочет подумать над решением, то он может ответить пробелом или переводом строки, в этом случае DWIM будет ждать окончательного ответа YES или NO.

DWIM в состоянии исправить некоторые простые синтаксические ошибки, однако она, естественно, не в состоянии исправить ошибки, связанные с логикой программы.

Интерлисп разработан с учетом требований исследователей и опытных пользователей. Многие подсистемы содержат широкий выбор директив и возможностей, существенно повышающих производительность труда, когда их используют по назначению. Кроме того, в системе имеются параметры, с помощью которых пользователь может для себя настроить систему.

Например, для DWIM можно составить следующее описание: пользователь опытный или начинающий, быстро или медленно работает с клавиатурой и т.д. В соответствии с этим описанием DWIM делает работу более удобной. Если вообще нежелательно вмешательство DWIM, то ее можно целиком отключить.

Лисп с фразовой структурой – Conversational Lisp

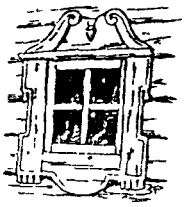
Для программистов, привыкших к языку с фразовой структурой, наверняка будет приятным наличие в Интерлиспе расширения CLisp, которое расширяет синтаксис Лиспа в направлении известных фразовых языков. CLisp дает возможность использовать IF p THEN q ELSE r, FOR e WHILE p DO q и другие типичные структуры управления, а также инфиксную запись для арифметических, логических операций и операций сравнения.

Формы системы CLisp реализованы с помощью макросов, транслирующих выражения в соответствующие им лисповские выражения. Система охватывает разнообразное структурное распознавание (Pattern Match), трансляцию и пакет работы с записями (Record Package), который позволяет определять типы данных на более высоком уровне с ориентацией на их применение.

CLisp автоматически преобразует предложения в соответствующие лисповские выражения. Предложения можно преобразовать и специальной функцией DWIMIFY. Аналогичным образом можно преобразовать через CLisp лисповские выражения и программы с помощью функции CLISPIFY.

Оконная система – Window System

Интерлисп наряду со Смолтолком был первой системой, в которой начали использоваться основанные на битовой карте многооконные методы изображения (Teitelman 1977). Расположенные рядом, наложенные и накрывающие друг друга, свободно перемещаемые окна создают гибкую и естественную рабочую среду, похожую на рабочий стол (deck-top metafora) с бумага-



ми и другими инструментами, а также динамический интерфейс для общения между человеком и машиной.

Одно окно может быть зарезервировано для диалога между пользователем и системой, второе – может отображать исходный текст тестируемой программы, в третьем – может просматриваться трасса выполнения, четвертое – может быть закреплено для справочного руководства, пятое – может содержать почту или диалог с другим пользователем и т.д. Каждое окно соответствует своей задаче, является посредником в обмене причастными к задаче данными и отображает течение событий в нем.

Целостность системы – System Integration

Различные подсистемы Интерлиспа сложным образом связаны друг с другом. Разные функции предполагают, что доступны сложные, автоматические и разумные услуги. С другой стороны это означает усложнение разработки системы и предъявляет большие требования к знанию пользователем системы и к ее использованию по назначению.

Для таких подсистем, как, например, CLisp, все возможные директивы точно не определены. Целью было – дать программисту более широкие возможности строить и экспериментировать с нужными конструкциями. Преимущество такой стратегии системы в том, что программист может легко изменять свои программы, наблюдать на непосредственном эксперименте за результатами их работы и при необходимости отменять неверные исправления.

Библиотека программ – Lispusers Package

С системой Интерлисп связана библиотека программ (Lispusers Package), охватывающая широкий набор различного прикладного программного обеспечения, а также включающая в себя расширение и документацию к системному программному обеспечению, в том числе

программы преобразования между Интерлиспом и другими диалектами языка, программное обеспечение оконной системы, базы данных, сетевой интерфейс и другое. Многие из этих программ уже входят в состав более новой версии Interlisp-D, которая образует интегрированную среду программирования для Лисп-машин Xerox серии 1100 (Xerox 1982). Таким образом, Интерлисп системы разделения времени послужил промежуточным этапом в переходе к Лисп-машинам Xerox серии 1100. Так же развивалась и среда Зеталисп Лисп-машин Symbolics, LMI и TI, опираясь на Маклисп системы разделения времени.

Литература

1. Bates M., Wagreich B., Bobrow R. Interlisp Primer. Bolt Beranek & Newman, Report No. 4353, 1980.
2. Epp B. *Interlisp Programmierhandbuch.* Institut fuer Deutsche Sprache, Mannheim, 1977.
3. Goodwin J. A Guided Tour of the Interlisp System. Universitetet i Linköping, Raport R-77-2, Linköping, 1977.
4. Haraldsson A. Interlisp – en avancerad integrerad programmerings-omgivning för Lisp-språket. Datalogicentrum, TH Linköping, R-82-29, 1982.
5. Sandewall E. Programming in an Interactive Environment: The Lisp Experience. *ACM Computing Surveys,* Vol. 10, No. 1, March, 1978.
6. Seppänen J. Interlisp ohjelmointijärjestelmä – ympäristö ja työvälineet. TKK, Laskentakeskus, 1984.
7. Teitelman W. Toward a Programming Laboratory, *Proceedings of the 1st IJCAI Conference*, 1969.
8. Tietelman W. A Display Oriented Programmer's Assistant. *Proceedings of the 5th IJCAI Conference,* Vol. 2, 1977.
9. Teitelman W., Masinter L. The Interlisp Programming Environment. *Computer*, Vol.14, No.4, 1981.
10. *Interlisp-D Users Guide.* Xerox Electro-optical Systems, Pasadena, California, 1982.
11. *Interlisp Reference Manual.* Xerox PARC, Palo Alto, California, 1978 и 1983.

Все преуменьшать – это мания величия.

Э. Диктуниус

3.3 СРЕДА ЗЕТАЛИСПА

- **Объектная система Flavor**
- **Макрос итерации Loop**
- **Интерфейс пользователя**
- **Оконная система**
- **Интегрированные средства разработки**
- **Экранный редактор Zmacs**
- **Инспектор структур Inspector**
- **Отладчик программ Debugger**
- **Управление файлами**
- **Инспектор состояния Peek**
- **Zmail и работа в сети**
- **Языки и инструменты**
- **Литература**

В качестве второго примера более развитых Лисп-систем рассмотрим Зеталисп. Он основан на разработанной в MIT системе Маклисп и на редакторе EMACS. Однако система заметно расширена и обновлена. Зеталисп содержит, например, следующие более развитые механизмы и черты:

- широкий выбор типов данных;
- возможность объектно-ориентированного программирования в системе Flavor;
- современные управляющие структуры, например инспирированный подсистемой CLisp Интерлиспа макрос LOOP, динамические механизмы передачи управления сопрограммы и процессы;
- гибкий механизм ключевых слов в лямбда-списке и многозначные функции;

- ввод и вывод, основывающийся на потоках;
- пространства имен;
- уже готовые функции, в том числе для сортировки, работы с линейными уравнениями и матричные вычисления.

Объектная система Flavor

По сравнению с Коммон Лиспом Зеталисп в качестве расширения содержит объектную систему Flavor, позволяющую осуществлять объектное программирование. В систему входит набор готовых базовых флейворов (flavor), которые программист может использовать по своему усмотрению. Например, оконная система основывается на объектах, с помощью которых программист может, особенно гибко комбинируя базовые возможности, создавать окна нового типа.

Макрос итерации Loop

Второе расширение по сравнению с Коммон Лиспом составляет макрос итерации LOOP (Loop Macro), обладающий большими возможностями. С его помощью можно изобразить сложные по структуре циклы на языке, напоминающем естественный. Структурами LOOP можно гибко комбинировать известные из языков с фразовой структурой циклические предложения и другие полезные в различных ситуациях формы циклов, такие как FOR, DO, WHILE, REPEAT, UNTIL, WHEN, WITH, APPEND, COLLECT, COUNT, SUM, INITIALLY и FINALLY. Эти итерации, как это видно и из названия, реализованы в виде макросов.

Зеталисп используется на Лисп-машинах LMI, Symbolics и TI и содержит в общей совокупности более 10 000 функций или всего более 20 мегабайтов кода. Наряду с языком Лисп среда Зеталисп содержит разносторонний интерфейс пользователя и целый набор интегрированных рабочих инструментов и других вспомогательных средств (facility). Системное програм-

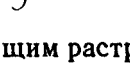
мное обеспечение целиком написано на Лиспе, и оно поддерживается на исходном языке.

Остановимся далее кратко на наиболее важных подсистемах Зеталиспа.

Интерфейс пользователя

Интерфейс пользователя системы Зеталисп состоит из внешних устройств и программного обеспечения, которые обеспечивают пользователю простой доступ ко всем средствам системы. В качестве устройства отображения используется дисплей с битовой картой (bitmap) и черно-белым экраном, обычно содержащим растр 1100 на 800 точек. Возможно использование и цветного дисплея. Для звукового вывода в систему входит DA-модем, усилитель и динамик.

Работу с экраном поддерживает оконная система, которая дает возможность наблюдать и управлять одновременно несколькими различными задачами или действиями, связанными с одной задачей. В качестве устройства ввода кроме клавиатуры используется мышь (mouse), с помощью которой можно "показывать" на устройстве отображения. Перемещение мыши по горизонтальной поверхности приводит в движение указатель на экране дисплея. С помощью кнопки (button) на мыши можно делать выбор (click) в окнах, переходить из одного окна в другое, открывать и закрывать окна, изменять и переносить их, а также запускать действия. Мышь – важное устройство взаимодействия. Наряду с этим директиву можно выдавать и путем ввода командного слова или связанного с ним слова с клавиатуры.



Программное обеспечение интерфейса можно с помощью параметров и новых определений подстроить под нужды пользователя.

Оконная система

Экран разбивается на окна, которые могут использоваться одновременно. Окна можно свободно формиро-

вать и перемещать, и они могут перекрывать друг друга и находиться рядом так же, как листы бумаги на столе. Скрытое окно можно вытащить на самый верх нажатием кнопки. Окно можно разбить на более мелкие окна, которые называются *форточками* (pane), и т.д. Для отображения кратковременных данных могут использоваться так называемые pop-up меню[1].

Интегрированные средства разработки

Ядром средств разработки (development tools) среды Зеталиспа являются оконный интерпретатор Лиспа (Lisp Listener) и (частичный) транслятор. Он взаимодействует с большим количеством более развитых средств программирования, таких как:

- оконный интерфейс пользователя (Window System);
- текстовый редактор (Zmacs);
- программа исследования структур данных (Inspector);
- отладчик программ (Display Debugger);
- программа исследования состояния системы (Peek);
- редактор системы файлов (File System Editor);
- редактор шрифтов (Font Editor) и форматтер текстов (Formatter);
- система электронной почты (Zmail).

Кроме Лиспа в системе доступно множество других языков (Фортран, Паскаль, Ада, Си, Пролог и другие) и другое базовое программное обеспечение, такое как преобразователь для работы с Интерлиспом (Interlisp Compatibility Package) и рабочий инструментарий более высокого уровня (например, Explorer Toolkits).

Различные подсистемы взаимодействуют друг с другом. Так, например, выполнение программ из редактора, исправление и трансляция в момент исполнения возможны таким же образом, как и в Интерлиспе. Программу можно тестировать, даже если не все ее части готовы. Программы можно выполнять по мере

---

[1] Меню, которые раскрываются по команде обращения к ним, как правило, с помощью мыши. *–Прим. ред.*

того, как они определяются. Динамическая загрузка позволяет изменять функции во время счета. Не подвергшиеся изменению части программы не нужно снова транслировать или загружать.

Экранный редактор Zmacs

Экранный редактор Zmacs основывается на разработанном в связи с Маклиспом текстовом редакторе Emacs, знающем синтаксис Лиспа и некоторых языков программирования и применимом для обыкновенной текстовой обработки. Zmacs, как и Emacs, содержит сотни возможностей, которые не встречаются в других редакторах. Новичок, конечно, может обойтись и меньшим подмножеством Zmacs. Редактор содержит возможности для определения характеристик пользователя (user profile), так можно задать новые команды, свойства и режимы редактирования (mode). Команды Zmacs задаются как с клавиатуры, так и с помощью мыши.

Zmacs – это так называемый структурный редактор. Это означает, что он по-разному работает в различных ситуациях или режимах (mode). Например, в лисповском режиме он обрабатывает текст в соответствии с синтаксисом Лиспа, в результате чего логическое перемещение в программе проще и быстрее. Zmacs поддерживает Лисп, например, с помощью следующих свойств:

1. Реакция миганием на открывающую скобку, соответствующую закрывающей скобке. Возможность ссылаться на выражения и работать с ними как с единым целым, при этом скобки остаются сбалансированными.
2. Осуществление отступа, соответствующего логической структуре кода (prettyprint).
3. Умение искать в рабочем пространстве редактируемый объект по имени (как функцию, переменную, макрос или объект).
4. Способность находить вызывающие функции и методы объектов.

Редактор Zmacs признает около 400 команд, и поэтому содержит документацию в оперативном режиме. С редактором тесно связаны программа форматирования (Text Formatter) и редактор шрифтов (Font Editor). Используя редактор шрифтов, можно с точностью до элементов растра (pixel) определить новые типы и размеры букв, специальные символы и изображения (icon) или изменить выбор предлагаемых системой готовых типов.

Инспектор структур Inspector

С помощью инспектора структур можно исследовать и вносить изменения в сложные лисповские объекты.

Инспектор наглядно отображает части объекта, при этом способ отображения зависит от типа объекта. Например, частями списка являются его элементы, частями символа – связанное с ним значение, определение функции или список свойств и т.д. В окне инспектора используется много форточек, в которых можно исследовать различные объекты, показывать меню, вычислять выражения, ссылаться на предшествующие команды и т.д.

Отладчик программ Debugger

Отладчик программ (Display Debugger) Зеталиспа запускается при возникновении ошибки. Он, как и интерпретатор структур, управляется посредством окна со многими форточками. В форточках содержатся, например, данные о предшествовавших ошибочной ситуации этапах вычисления и о связях каждого окружения, выбор команд, а также открываются окна для интерпретатора Лиспа и инспектора структур, используемого для исследования и тестирования ошибочных структур.

Управление файлами

На уровне каталога для работы с файлами и управления ими существует специальный оконный редактор. С его помощью можно, например, удалять и создавать файлы и каталоги, давать им имена, распечатывать на принтере, делать копию на магнитной ленте, а также исследовать и устанавливать характеристики каталогов.

Инспектор состояния Peek

Зеталисп содержит также специальный инспектор (Peek), с помощью которого можно исследовать и модифицировать различные части системы и их действие, например: процессы, различные статистические счетчики, окна, виртуальную память и локальную сеть.

Zmail и работа в сети

Лисп-машины спроектированы как индивидуальные рабочие места, но их можно объединять в локальную сеть (local area network). Таким образом, становится возможным общение между рабочими местами, а также совместное использование файлов, программ и внешних устройств (например, лазерный принтер, дисковод). Работа в сети объединяет преимущества двух подходов: разделения времени и индивидуальных рабочих мест. В локальной сети обычно есть машины (например, для хранения файлов), и, кроме этого, в ней могут быть шлюзы (gateway) в другие локальные или общие сети данных.

Языки и инструменты

Выбор используемых в среде Зеталиспа инструментов и языков программирования зависит от поставщика, причем предлагаемый набор средств все время расширяется. Среди других языков предлагаются Фортран, Паскаль, Ада и Пролог. Для этих языков в среде Зеталиспа существуют особенно развитые программные

окружения, и разработанные в них программы можно выполнять вместе с программами на Лиспе.

Готовые инструменты и прикладные разработки в большом количестве имеются для работы с экспертными системами, с естественным языком и речью, с реляционными базами данных, машинной графики и машинного проектирования.

Литература

1. Hyvönen E., Seppänen J., Syrjänen M. *STeP-84 Tutorial and Industrial Papers.* Suomen Tekoälytutkimuksen Päivät, Tietojenkäsittelytieteen seuran julkaisu 4, Otaniemi, 1984.
2. Stallman R.M. *Emacs* – The Extensible, Customizable, Self-Documenting Display Editor. Memo No. 519, AI Laboratory, MIT, Cambridge, Massachusetts, 1979.

3. *Explorer Technical Summary.* Symbolics Inc., Cambridge, Massachusetts, 1981.
4. *Overview of LMI Lisp Machine Software.* Lisp Machine Inc., Culver City, California, 1982.
5. *Symbolics 3600 Technical Summary.* Symbolics Inc., Cambridge, Massachusetts, 1983.
6. Weinreb D., Moon D.A. *Lisp Machine Manual.* Symbolics Inc., Cambridge, Massachusetts, 1981.

4 ПРИМЕРЫ ПРОГРАММ

Для того чтобы дать представление о структурах, механизмах, принципах Лиспа и технике программирования на нем, в главах книги, посвященных программированию на Лиспе, был представлен и запрограммирован целый набор различных небольших программ. Теперь мы познакомимся с образцами несколько более основательного программирования. В приводимых примерах обычно используются достаточно простые лисповские функции, из совокупного поведения которых и с учетом их взаимодействия возникают большие программы и системы.

Примеры подбирались так, чтобы дать представление о различных областях применения и стилях программирования на Лиспе. Их нельзя рассматривать как примеры практических систем, они только показывают, какие приемы и решения можно применять, создавая в рамках технологии работы со знаниями языки представления данных и способы решения проблем более высокого уровня.

Прежде всего в качестве примера использования Лиспа в системном программировании напишем программу простого интерпретатора Лиспа с помощью самого Лиспа.

Вторым примером будет небольшая алгебраическая система, Миксима, которая умеет анализировать и выполнять простейшие символьные действия, такие как нахождение производной, упрощение выражений и их вычисление.

Третий пример – из гуманитарной области, из области соприкосновения языкознания и фольклора. Он связан с относимыми в настоящее время к детской культуре игровыми и тайными языками, которых только в финском языке насчитывается около тридца-

ти. Запрограммируем здесь один из них – язык сплетника и его расширение – цыганский жаргон.

В следующем примере рассматриваются экспертные системы и продукционное программирование. Построим небольшую базу знаний, содержащую сведения о животных, и запрограммируем для нее машину принятия решений, решающую задачу простой классификации животных по их признакам.

Последний пример знакомит с объектным программированием, используемым в модели движения планет и спутников Солнечной системы.

Каждый пример завершается списком литературы, предназначенным для более детального ознакомления с предметом.

Один пример лучше, чем тысяча доказательств.

У. Гладстон

4.1 ЛИСП НА ЛИСПЕ

- **Интерпретатор Лиспа на Лиспе**
- **Примитивы интерпретатора**
- **Универсальная функция EVAL1**
- **Основная часть интерпретатора: APPLY1**
- **Примеры вычислений**
- **Печать результатов – структурная печать**
- **Программирование диалога**
- **Программирование ввода и вывода**
- **Литература**

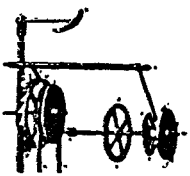
Интерпретатор Лиспа сравнительно легко запрограммировать на самом Лиспе. Но определение какого-нибудь языка на нем самом – это не такая уж очевидная задача. Хорошим пояснением к этому будет история зарождения подобной идеи (Stoyan 1984).

Интерпретатор Лиспа на Лиспе

Маккарти рассказывает, что во время его размышлений о семантике лисповской функции EVAL С. Рассел предложил запрограммировать ее тем же способом, что и другие функции Лиспа:

"... Этот EVAL был написан и опубликован в нашей статье, когда Стив Рассел сказал: "Послушайте, почему мне не запрограммировать этот EVAL и вы получите интерпретатор", на что я сказал ему: "Ну, ну, вы путаете теорию с практикой, наш EVAL предназначен для чтения, а не для вычислений." Но он на этом не остановился и сделал интерпретатор. Таким образом, С. Рассел оттранслировал EVAL из моей статьи в машинный код 704, отладил его и объявил

результат в качестве интерпретатора Лиспа, чем он, конечно, и являлся, ...”



После того как идея была подана, ее осуществление не вызвало уже особых затруднений, так как в Лиспе данные и программы представляются одинаково. Используемые при этом основные средства – это блокировка вычислений, при помощи которой можно при необходимости запретить вычисления; базовые функции; условные операторы и механизмы рекурсии и определения функции.

Рассмотрим ниже простой интерпретатор Лиспа, запрограммированный на нем самом. Он в некоторой степени отличается как от первоисточника, так и от интерпретаторов реальных систем. Тем не менее он дает довольно ясное представление о ядре интерпретатора и его работе, а также показывает, как легко и изящно семантика Лиспа поддается определению и программированию и на других языках.

При необходимости интерпретатор можно расширять, добавляя в систему новые ветви и определения функций. Используя ту же технику, можно также реализовать новые языки программирования высокого уровня для различных приложений. В реализации интерпретатора будем использовать чисто функциональное программирование.

Примитивы интерпретатора

Пусть наш интерпретатор называется EVAL1 в отличие от интерпретирующего его системного интерпретатора EVAL. Соответственно назовем и интерпретируемые нами базовые функции: CAR1, CDR1, CONS1, EQUAL1 и ATOM1. Кроме того, будем использовать соответствующее предложению LAMBDA предложение LAMBDA1 и формы COND1 и QUOTE1. T1, NIL1 и числа будут константами системы.

Во время вычисления предполагается, что вызываемые пользователем функции определены предложением

LAMBDA1, которое является свойством FN имени функции.

Применим в нашем интерпретаторе в отличие от Коммон Лиспа динамический способ вычислений.

Универсальная функция EVAL1

Предположим, что связи переменных хранятся в ассоциативном списке СВЯЗИ. Вычисление форм происходит путем интерпретации их структур и выполнения соответствующих действий. Универсальную функцию EVAL1 можно определить приведенным ниже условным предложением:

```
(defun eval1 (форма &optional (связи nil))
 (cond
   ((atom форма) ; форма атомарная
      (cond
         ((eq форма 't1) 't1)
         ((eq форма 'nil) 'nil1)
         ((numberp форма) форма)
         ((car (assoc форма связи)))
         (t (format t
                "~%У атома нет связи: ~S"
                форма))
   ;; голова формы - атом
   ((atom (car форма))
      (cond
         ((eq (car форма) 'quote1)
            (cadr форма))
         ((eq (car форма) 'cond1)
            (eval-cond (cdr форма) связи))
         ;; вызов функции
         ((get (car форма) 'fn)
            (apply1 (get (car форма) 'fn)
               (eval-список (cdr форма)
                     связи)
               связи))
```

```
        ;; встроенная функция
        (t (apply1 (car форма)
                   (eval-список (cdr форма)
                                связи)
                   связи))))
    ;; LAMBDA1-предложение
    (t (apply1 (car форма)
               (eval-список (cdr форма) связи)
               связи))))
```

Из определения видно, что если ФОРМА – просто переменная (т.е. не константа T1, NIL1 или число), то ее значение получают с помощью функции ASSOC из представленного в форме а-списка вычислительного контекста СВЯЗИ.

Функция EVAL-COND определяет семантику условного оператора COND1:

```
(defun eval-cond (ветви контекст)
 (cond
   ((null ветви) 'nil1)
   ((not (eq (eval1 (caar ветви) контекст)
             'nil1))
      (eval1 (cadar ветви) контекст))
   (t (eval-cond (cdr ветви) контекст))))
```

Если форма – это список, первый элемент которого – имя функции, то выражение можно вычислить, получив LAMBDA1-предложение, соответствующее имени функции, как свойство под именем FN с помощью функции GET и применив его при помощи функции APPLY1. Форма может быть непосредственно LAMBDA1-вызовом с фактическими параметрами:

((LAMBDA1 *фор-параметры тело) факт-параметры***)**

Эту ситуацию обрабатывает последняя ветвь EVAL1.

Основная часть интерпретатора: APPLY1

Функция APPLY1 отвечает за применение функции к значениям своих аргументов. С ее помощью определены также и базовые функции интерпретатора: CAR1, CDR1, CONS1, ATOM1, EQUAL1 и LAMBDA1-предложение:

```
(defun apply1 (функция аргументы связи)
 (cond ((atom функция) ; базовые функции
   (cond
      ((eq функция 'car1)
         (cond ((eq (car аргументы) 'nil1)
                  'nil1)
               ;; Голова NIL1 - это NIL1
               (t (caar аргументы))))
      ((eq функция 'cdr1)
         (cond ((eq (car аргументы) 'nil1)
                  'nil1)
               ;; Хвост NIL1 - это NIL1
               ((null (cdar аргументы))
                  'nil1)
               (t (cdar аргументы))))
      ((eq функция 'cons1)
         (cond ((eq (cadr аргументы)
                    'nil1)
                  (list (car аргументы)))
               (t (cons (car аргументы)
                        (cadr аргументы)))))
      ((eq функция 'atom1)
         (cond ((atom (car аргументы))
                  't1)
               (t 'nil1)))
      ((eq функция 'equal1)
         (cond ((equal (car аргументы)
                       (cadr аргументы))
                  't1)
               (t 'nil1)))
```

```
        (t (format t "~%Неизвестная функция:
                      ~S" функция))))
    ((eq (car функция) 'lambda1)
         ;; лямбда-вызов
       (eval1 (caddr функция)
          (создай-связи (cadr функция)
             аргументы связи)))
    (t (format t
         "~%Это не лямбда-вызов: ~S"
         функция))))
```

Функция ФУНКЦИЯ должна быть либо известной транслятору базовой функцией, либо более сложным вычислением, определенным через выражение LAMBDA1. Функцией СОЗДАЙ-СВЯЗИ в окружение добавляются связи формальных параметров с фактическими:

```
;;; Окружение новыми парами имя-значение

(defun создай-связи
       (формальные фактические окружение)
 (cond
   ((null формальные) окружение)
   (t (acons (car формальные)
             (car фактические)
             (создай-связи (cdr формальные)
                           (cdr фактические)
                           окружение)))))
```

Древнеиудейское дерево жизни.

Вторым аргументом APPLY1 – АРГУМЕНТЫ – является список значений аргументов вызываемой функции, который вычисляется функцией EVAL-СПИСОК на следующей странице.

Заметим, что интерпретатор Лиспа образуется в основном двумя взаимно рекурсивными функциями EVAL1 и APPLY1, через которые семантика языка получила формальное определение. Перво-

```
(defun eval-список (невычисленные связи)
 (cond
   ((null невычисленные) nil)
   (t (cons
        (eval1 (car невычисленные) связи)
        (eval-список (cdr невычисленные)
                     связи)))))
```

начально Маккарти определил интерпретатор в метанотации Лиспа (McCarthy 1963, Lichtman 1986), при использовании которой элегантность определения языка особенно заметна:

```
1 evalquote [fn;x] = apply[fn;x;NIL]

2 apply[fn;x;a]=[atom[fn] --> [eq[fn;CAR] --> caar[x];
3                              eq[fn;CDR] --> cdar[x];
4                              eq[fn;CONS] --> cons[car[x];cadr[x]];
5                              eq[fn;ATOM] --> atom[car[x]];
6                              eq[fn;EQ] --> eq[car[x];cadr[x]];
7                              T      --> apply[eval[fn;a];x;a]];
8   eq[car[fn];LAMBDA] --> eval[caddr[fn];pairlis[cadr[fn];x;a]];
9   eq[car[fn];LABEL] -->
                  apply[caddr[fn];x;cons[cons[cadr[fn];caddr[fn]];a]];
10  eq[car[fn];FUNGAR] --> apply[cadr[fn];x;caddr[fn]]]

11 eval[e;a]=[atom[e] --> cdr[assoc[e;a]];
12       atom[car[e]] --> [eq[car[e];QUOTE] --> cadr[e];
13                         eq[car[e];COND] --> evcon[cdr[e];a];
14                         eq[car[e];FUNCTION] -->
                                        list[FUNARG;cadr[e];a];
15                         T --> apply[car[e];evlis[cdr[e];a];a]];
16       T --> apply[car[e];evlis[cdr[e];a];a]];

17 pairlis[x;y;a]=[null[x] --> a;
                   T --> cons[cons[car[x];car[y]];pairlis[cdr[x];cdr[y];a]]]

18 assoc[x;a]=[eq[caar[a];x] --> car[a]; T --> assoc[x;cdr[a]]]

19 evcon[c;a]=[eval[caar[c];a] --> eval[cadar[c];a];
                   T --> evcon[cdr[c];a]]

20 evlis[m;a]=[null[m] --> NIL;
                   T --> cons[eval[car[m];a];evlis[cdr[m];a]]]
```

Примеры вычислений

Теперь можно опробовать интерпретатор:

```
_(eval1 't) ; T не является константой
У атома нет связи: T     ; нашего
NIL                      ; интерпретатора
_(eval1 'nil1) ; NIL1 является константой
NIL1
_(eval1 '(конс (quote1 a) (quote1 (b c))))
Неизвестная функция: КОНС
NIL
_(setf (get 'конс 'fn)  ; определение
   '(lambda1 (голова хвост) ; функции
      (cons1 голова хвост)))
(LAMBDA1 (ГОЛОВА ХВОСТ) (CONS1 ГОЛОВА
ХВОСТ))
_(trace eval1 apply1)    ; включить
(EVAL1 APPLY1)           ; трассировку
_(eval1 '(конс (quote1 a) (quote1 (b c))))
EVAL1:
ФОРМА = (КОНС (QUOTE1 A) (QUOTE1 (B C)))
СВЯЗИ = NIL
   EVAL1:
   ФОРМА = (QUOTE1 A)
   СВЯЗИ = NIL
   EVAL1 = A
   EVAL1:
   ФОРМА = (QUOTE1 (B C))
   СВЯЗИ = NIL
   EVAL1 = (B C)
   APPLY1:
   ФУНКЦИЯ = (LAMBDA1 (ГОЛОВА ХВОСТ)
   (CONS1 ГОЛОВА ХВОСТ))
   АРГУМЕНТЫ = (A (B C))
   СВЯЗИ = NIL
      EVAL1:
      ФОРМА = (CONS1 ГОЛОВА ХВОСТ)
      СВЯЗИ = ((ГОЛОВА A) (ХВОСТ (B C)))
```

```
          EVAL1:
          ФОРМА = ГОЛОВА
          СВЯЗИ = ((ГОЛОВА A) (ХВОСТ (B
          C)))
          EVAL1 = A
          EVAL1:
          ФОРМА = ХВОСТ
          СВЯЗИ = ((ГОЛОВА A) (ХВОСТ (B
          C)))
          EVAL1 = (B C)
          APPLY1:
          ФУНКЦИЯ = CONS1
          АРГУМЕНТЫ = (A (B C))
          СВЯЗИ = (ГОЛОВА A) (ХВОСТ (B C)))
          APPLY1 = (A B C)
       EVAL1 = (A B C)
   APPLY1 = (A B C)
EVAL1 = (A B C)
(A B C)
_(setf (get 'member1 'fn)
   ;; другое определение
   '(lambda1 (элемент список)
      (cond1
         ((equal1 список nil1) nil1)
         ((equal1 элемент (car1 список))
            список)
         (t1 (member1 элемент
                      (cdr1 список))))))
(LAMBDA1 (ЭЛЕМЕНТ СПИСОК) (COND1 ...))
_(untrace eval1 apply1);снятие трассировки
(EVAL1 APPLY1)
_(eval1 '(member1 (quote1 b)
                  (quote1 (a b c))))
(B C)
_(eval1 '(member1 (quote1 d)
                  (quote1 (a b c))))
NIL1
```

Печать результатов – структурная печать

Ранее говорилось, что в Лисп-систему или в ее редактор в качестве вспомогательного средства входит *структурная печать* (prettyprinter). С ее помощью функцию или произвольный список можно вывести в красивой форме, с определенными логикой отступами и проверить, правильно ли расставлены скобки. Однако можно отказаться от предусмотренной структурной печатью формы вывода, например, при работе с реализованным на Лиспе языком или способом представления данных более высокого уровня. В этом случае придется заново программировать вывод.

В качестве примера формы вывода мы далее запрограммируем небольшую функцию структурной печати PPRINT1 (в Коммон Лиспе эта функция называется PPRINT). Структурная печать рекурсивно выводит списки так, что одно- и двухэлементные списки печатаются в том виде, как они есть, а более длинные преобразуются к более удобной для чтения форме:

```
(defun pprint1 (l &optional (отступ 0))
 (cond
   ((atom l)
      (установи отступ)
      (prin1 l))          ; Вывод атома
   ((< (length l) 3)      ; Короткий список
      (установи отступ)   ; просто выводится
      (prin1 l))
   (t (terpri)            ; Длинный список
      (установи отступ)   ; преобразуется
      (princ "(")
      (prin1 (car l))
      (pp-хвост (cdr l) отступ))))
```

```
;;; Хвост выводится с отступом

(defun pp-хвост (l отступ)
 (cond ((null l) (princ ")") t)
       (t (terpri)
          (установи отступ)
          (pprint (car l) (+ отступ 3))
          (pp-хвост (cdr l) отступ))))

(defun установи (n)
 ;; выводит n пробелов
 (cond ((= n 0) t)
       (t (princ " ")
          (установи (- n 1)))))
```

```
_(pprint '(list (car x) (cons (car y) (cdr
z))))
(LIST
   (CAR X)
   (CONS
      (CAR Y)
      (CDR Z)))
T
_(pprint1 '(cond ((null x) nil) (t (cons
(car x) (cdr y)))))
(COND
   ((NULL X) NIL)
   (T (CONS (CAR X) (CDR Y))))
T
```

Программирование диалога

С помощью функций PPRINT1, READ и PRINx можно запрограммировать цикл диалога (top level) нашего интерпретатора Лиспа EVAL1. В приведенном ниже определении в качестве приглашения интерпретатора будем использовать символ "<–". Кроме

этого, признаком выводимых результатов будет символ "->".

```
(defun интерпретатор nil
 (progn
   (princ "Наш Лисп:")
   (terpri)
   (terpri)
   (princ "<-")
   (do ((выражение (read) (read)))
      ((eq выражение 'конец) 'до-свидания)
      (princ "->")
      (pprint1 (eval1 выражение) 3)
      (terpri)
      (princ "<-"))))
```

```
_(интерпретатор)
Наш Лисп:
<- 5
-> 5
<- (quote рыбий-зонтик)
-> РЫБИЙ-ЗОНТИК
<- (cons1 (quote1 a) (quote1 (b c)))
-> (A B C)
<- конец
ДО-СВИДАНИЯ
_
```

Программирование ввода и вывода

В описанном выше интерпретаторе Лиспа для ввода и вывода выражений мы использовали системные функции ввода и вывода. Чтобы можно было реализовать интерпретатор Лиспа в какой-нибудь другой среде, нам, естественно придется отдельно запрограммировать ввод и вывод выражений. Уточним ввод и вывод списков на системном уровне. Рассматриваемые функции работают со списками, в которых для отличия вместо круглых скобок

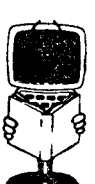
используются угловые скобки "<" и ">". Пустой список обозначается символом "<>" без пробела.

Определим простую функцию ЧИТАЙ-ВЫРАЖЕНИЕ, которая вводит выражения с угловыми скобками и строит из них обычные списки. Для простоты предположим, что скобки пишутся отдельно от символов и друг от друга, чтобы их можно было прочитать с помощью функции READ, не вдаваясь в детали представления атомов на уровне знаков. Затем запрограммируем функцию ВЫВЕДИ-ВЫРАЖЕНИЕ для вывода скобочных выражений с угловыми скобками.

Структуру, или синтаксис, скобочных выражений (последовательности знаков) можно определить следующими рекурсивными правилами в форме Бэкуса–Наура:

выражение → "<" *выражение хвост* (1)
выражение → "<>" (2)
выражение → *атом* (3)
хвост → ">" (4)
хвост → *выражение хвост* (5)

В этой записи символы "выражение" и "хвост" являются нетерминальными символами (non-terminal), а "атом" соответствует произвольному лисповскому атому. "Выражение" – это или атом, или пустой список "<>", или последовательность, состоящая из открывающей скобки "<" и рекурсивного "выражения" и "хвоста". "Хвост" в свою очередь это или закрывающая скобка ">", или последовательность, состоящая из "выражения" и рекурсивного "хвоста".



На основе вышеперечисленных правил можно определить функцию ЧИТАЙ-ВЫРАЖЕНИЕ, которая будет читать выражение, строить соответствующую ему списочную внутреннюю структуру данных и возвращать ее значение.

Функция чтения состоит из двух взаимно рекурсив-

```
(defun читай-выражение ()
 (let ((x (read)))
   (cond ((eq x '<)          ; правило 1
          (cons (читай-выражение)
          (читай-хвост)))
         ((eq x '<>) nil)    ; правило 2
         (t x))))            ; правило 3

(defun читай-хвост.()
 (let ((x (read)))
   (cond ((eq x '>) nil)     ; правило 4
         ((eq x '<)          ; правило 5
          (cons (cons (читай-выражение)
                (читай-хвост))
                (читай-хвост)))
         ((eq x '<>)         ; правило 5
          (cons nil (читай-хвост)))
                             ; правило 5
         (t (cons x (читай-хвост))))))
```

ных функций, ветви которых соответствуют правилам грамматики[1].

```
_(читай-выражение)
a                ; введенное выражение
A                ; списочное представление
_(читай-выражение)
< a < c > < < d <> > > >    ; выражение
(A (C) ((D NIL))) ; списочное представление
```

Теперь можно написать и функцию вывода ВЫВЕДИ-ВЫРАЖЕНИЕ:

```
_(defun выведи-выражение (x)
   (cond
      ((null x) '<>))        ; правило 2
      ((atom x) (prin1 x)) ; правило 3
```

[1] См. с. 186. *–Прим. ред.*

```
      (t (princ "<")          ; правило 1
         (выведи-выражение (car x))
         (выведи-хвост (cdr x)))))
ВЫВЕДИ-ВЫРАЖЕНИЕ
_(defun выведи-хвост (x)
   (cond
      ((null x) (princ ">") t) ; правило 4
      (t (princ " ")           ; правило 5
      (выведи-выражение (car x))
      (выведи-хвост (cdr x)))))
ВЫВЕДИ-ХВОСТ
_(выведи-выражение '(a (b) ((c nil)))
<A <B> <<C <>>>>
T
```

Алгоритмы ввода и вывода списков представлены здесь в упрощенной форме, которую можно реализовать и на других языках. В Коммон Лиспе для программирования ввода новых форм записи существуют специальные макросы чтения (см. т.1, с.177).

Литература

1. Allen J. *Anatomy of Lisp.* McGraw-Hill, New York, 1978.
2. Henderson P. *Functional Programming – Application and Implementation.* Prentice-Hall, London, 1980.
3. Lichtman Z.L. The Function of T and NIL in LISP. *Software – Practice and Experience,* Vol.16, No. 1–3, Jan.1986, pp. 1–3.
4. McCarthy J. *Lisp 1.5 Programmers Manual.* MIT Press, Cambridge, Massachusetts, 1963.
5. Seppänen J. *Lisp kielenopas.* OtaDATA, Espoo, 1972.
6. Stoyan H. Early Lisp History. *In Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* Austin, Texas, Aug. 6–8, 1984.

Из земли прорастет и самое малое семя.

Финская пословица

4.2 МИКСИМА

- **Миксима – символьный вычислитель**
- **Действия и их порядок**
- **Чтение выражения с преобразованием в списочную форму**
- **Преобразование выражения в форму дерева**
- **Представление выражения в форме дерева**
- **Порядок обхода дерева**
- **Интерпретация и вычисление выражений**
- **Упрощение выражений**
- **Снятие скобок и вывод**
- **Диалог с Миксимой**
- **Литература**

Миксима – символьный вычислитель

Максима (Macsyma 1977) – это известная алгебраическая система, разработка которой началась в Массачусетском технологическом институте уже в 60-х годах в рамках проекта MAC. Вначале исследование символьной и алгебраической обработки математических выражений (symbolic and algebraic computing, SAC) было связано с искусственным интеллектом, так как к предлагаемым ею хорошо определенным примерам были применимы общие способы решения проблем. Однако довольно скоро она превратилась в отдельную самостоятельную область исследований, относящуюся сейчас больше к математике, чем к искусственному интеллекту. Помимо Максимы есть другие известные алгебраические системы: Reduce (Hearn et al. 1974), Scratchpad, Аналитик, Smp, MuMath/MuSimp и другие.

Миксима – это небольшая программа символьной математики (Seppänen 1972). Она может обрабатывать предложения как в численном, так и в символьном виде. Миксима умеет распознавать, дифференцировать, упрощать выражения и выводить их без лишних скобок. Диалог с Миксимой происходит путем ввода выражения или операторов присваивания, которые предполагается вычислить или упростить их запись. Например:

```
<= 2 + 3 * 4 !     ; арифметическое действие
=> 14              ; значение
<= a := 2 * b !    ; символьное действие
=> 2 * b           ; символьное значение
<= b := 3 !        ; присваивание
=> 3               ; значение
<= a !             ; имя выражения
=> 6               ; значение выражения
<= f := x + 3 * x !;присваивание выражения
=> x + 3 * x       ; значение
<= f d x !         ; дифференцирование
=> 4               ; производная
```

Далее структура и работа программы, а также необходимые формы представления и алгоритмы будут рассмотрены более детально.

Действия и их порядок

Для простоты предположим, что вводимые для Миксимы символы отделяются друг от друга пробелами. Миксима считывает выражение, вычисляет его значение или преобразовывает его и выводит ответ. В этой главе при программировании ограничимся приведенными ниже действиями:

```
(setq *действия* '(/ * - + := : d))
```

Здесь /, *, – и + соответствуют нормальным арифметическим действиям, := – это присваивание и d – дифференцирование. Допустим также, что несколько операторов можно писать на одной строке, разделяя их двоето-

чием. *Порядок* (precedence) выполнения действий такой же, как и порядок их следования в списке *ДЕЙСТВИЯ*. Например, в выражении

$$x := a * b / c$$

следующий порядок действий:

$$x := (a * (b / c))$$

В остальных случаях вычисления осуществляются слева направо. Например:

a – b – c = (a – b) – c

Для простоты ограничимся бинарными операциями (с двумя аргументами). Тогда отрицательное символьное значение, например –a, будет записываться в виде 0–a.

Чтение выражения с преобразованием в списочную форму

Определим прежде всего функцию ЧИТАЙ, которая считывает вводимое пользователем выражение, оканчивающееся восклицательным знаком, и преобразует его в список:

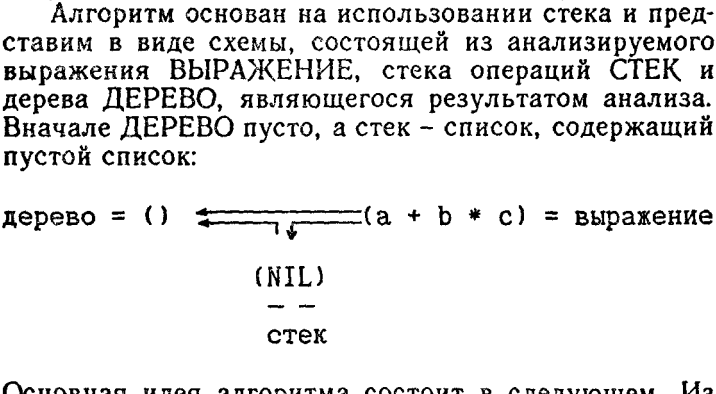
```
(defun читай ()
   ((lambda (x)
         (if (eq x '!) nil
            (cons x (читай))))
      (read)))
```

```
_(читай)
a b c !
(A B C)
```

Преобразование выражения в форму дерева

Язык выражений, рекурсивно построенных из бинарных операций – это простой контекстно-независимый язык (context free). Анализ его выражений может быть осуществлен, например, с помощью следующего довольно простого алгоритма, называемого методом *операторного предшествования* (operator precedence).

Алгоритм основан на использовании стека и представим в виде схемы, состоящей из анализируемого выражения ВЫРАЖЕНИЕ, стека операций СТЕК и дерева ДЕРЕВО, являющегося результатом анализа. Вначале ДЕРЕВО пусто, а стек – список, содержащий пустой список:



Основная идея алгоритма состоит в следующем. Из предложения последовательно друг за другом выбираются символы и помещаются либо прямо в дерево, либо в стек на то время, пока другие символы не будут помещены либо в дерево, либо в стек над этим символом.

Если считанный символ – это величина (константа или переменная), то его помещают прямо в дерево. Если это – операция, то ее помещают или в дерево, или в стек в соответствии с тем, выше или ниже ее приоритет, чем приоритет операции, находящейся в верхней ячейке стека (или NIL). Когда все символы прочитаны до конца, все находящиеся после этого в стеке операции переносятся в дерево и на этом анализ заканчивается. Стек здесь выступает в роли комнаты ожидания, в которой операции с более низким приоритетом ждут, пока не пройдут более высокоприоритетные операции.

Более точно можно сформулировать алгоритм следующим образом:

Алгоритм разбора:

1. Если предложение пусто, то перейти к п. 6, иначе взять следующий символ.
2. Если символ является переменной или константой, то перенести его в дерево и перейти к п. 1.
3. Если символ – это операция, и стек пуст, то поместить символ в стек и перейти к п. 1.
4. Если приоритет операции выше приоритета верхней операции, уже находящейся в стеке, то поместить ее в стек и перейти к п. 1.
5. Если приоритет операции меньше или равен приоритету верхней операции в стеке, то поместить операцию из стека в дерево и перейти к п. 3.
6. Если стек пуст, то дерево разбора готово и алгоритм на этом заканчивается.
7. Переместить символ из верхушки стека в дерево и перейти к п. 6.

Алгоритм преобразует выражения в так называемую *обратную польскую* (reverse Polish) запись, в которой символ операции следует непосредственно за аргументами (операндами), например:

$$(a + b) \Rightarrow (a\ b\ +)$$
$$(a + b * c) \Rightarrow (a\ b\ c\ *\ +)$$
$$(a * b + c) \Rightarrow (a\ b\ *\ c\ +)$$

Если в выражении встречается подвыражение, то его можно анализировать, рекурсивно вызывая тот же алгоритм:

$$((a + b) * c) \Rightarrow (a\ b\ +\ c\ *)$$

По мере перемещения символов в стек или в дерево над ними можно было бы выполнять различные действия, касающиеся их формы или порядка. Перемещая, например, в дерево символ операции, можно выпол-

нить и саму операцию, если только, конечно, у находящихся в дереве символов-аргументов имеются значения, для которых определена и выполнима данная операция. Можно также определить и некоторую форму записи или язык, к которому преобразуется выражение в случае, если выполнение операции еще невозможно.

Представление выражения в форме дерева

С точки зрения символьной обработки предпочтительнее преобразовывать выражения в форму дерева, в которой легко обрабатывать все подвыражения целиком. Так и будем поступать: всегда, когда в дерево перемещается символ операции, будем вносить изменение, преобразующее два верхних элемента дерева и символ операции в дерево в префиксной форме:

```
(... a b) ⇒      ; два верхних элемента дерева и
                 ; символ добавляемой операции
(... (+ a b))    ; преобразуется в дерево в префиксной
                 ;  форме
```

Это преобразование введем в реализующую наш алгоритм разбора функцию АНАЛИЗИРУЙ в виде функции ПРЕОБРАЗУЙ.

```
(defun анализируй (дерево стек выражение)
 (cond
   ((null выражение)
      (if (car стек)
         (преобразуй дерево
            стек выражение)
         (car дерево))
   ((atom выражение) выражение)
   ((atom (car выражение))
      (cond
         ((not (member (car выражение)
                       *действия*))
```

```
                (анализируй
                   (cons (car выражение)
                         дерево)
                   стек
                   (cdr выражение)))
             ((старше (car выражение)
                   (car стек))
                (анализируй дерево
                   (cons (car выражение))
                         стек)
                         (cdr выражение)))
             (t (преобразуй дерево
                   стек выражение)
      (t (анализируй
             (cons (анализируй nil '(nil)
                               (car выражение))
                   дерево)
             стек
             (cdr выражение)))))

(defun преобразуй (дерево стек выражение)
   (анализируй
      (cons (list (car стек)
                  (cadr дерево)
                  (car дерево))
            (cddr дерево))
      (cdr стек)
      выражение))
```

Определять старшинство операций P и Q будет предикат СТАРШЕ, который возвращает значение T, если P старше, т.е. стоит в списке *ДЕЙСТВИЯ* раньше Q, а иначе возвращает NIL:

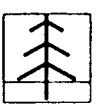
```
(defun старше (p q)
   (or (null q)
      (member q (member p *действия*))))
```

Теперь можно попробовать преобразовать выражение в форму дерева:

```
_(анализируй nil '(nil) '(a + b * c))
(+ A (* B C))
```

Порядок обхода дерева

Выражения, представляющие деревья, могут быть получены с помощью трех различных *порядков обхода* (walk order), позволяющих систематически обойти все узлы дерева. Возможными порядками обхода являются:



1. Прямой порядок (preorder).
   Префиксная запись (prefix): (+ a (* b c)).
   1. Обработать узел.
   2. Обойти левое поддерево.
   3. Обойти правое поддерево.

2. Промежуточный порядок (inorder).
   Инфиксная запись (infix): (a + (b * c)).
   1. Обойти левое поддерево.
   2. Обработать узел.
   3. Обойти правое поддерево.

3. Обратный порядок (postorder)[1].
   Постфиксная запись (postfix, suffix): (a (b c *) +).
   1. Обойти левое поддерево.
   2. Обойти правое поддерево.
   3. Обработать узел.

Первый и третий способы названы польской записью (Polish form) в честь их изобретателя польского математика Л. Лукасевича. Их преимущество перед вторым способом состоит в том, что для них скобки

---

[1] Здесь авторы в выборе терминов исходят из получающегося вида записи. Д. Кнут в книге "Искусство программирования", т.1, дает немного отличающуюся классификацию. *–Прим. перев.*

вообще не нужны. Порядок выполнения операций и так ясен на основании порядка следования символов. Польскую запись часто используют в машинных программах, трансляторах и интерпретаторах.

Интерпретация и вычисление выражений

Рассмотрим в качестве примера вычисление выражений в постфиксной или обратной польской записи. Вычисления можно проводить, используя простой стековый алгоритм по следующей схеме:

Стек = ( ) <——————— *(2 3 4 * +) = Дерево*

1. Из дерева считывается символ и перемещается в стек.
2. Если считанный символ – это символ операции, то операция выполняется над двумя верхними элементами стека и все три символа заменяются результатом.
3. Перейти к п. 1.

Например:

Стек	*Действие*	*Дерево*
( )	←	*(2 3 4 * +)*
(2)	←	*(3 4 * +)*
(2 3)	←	*(4 * +)*
(2 3 4)	←	*(* +)*
*(2 3 4 *)*	*применение*	*(+)*
*(2 (3 4 *))*	*	*(+)*
(2 12)	←	*(+)*
(2 12 +)	+	*( )*
(14)	←	*( )*

6	1	8
7	5	3
2	9	4

Определим теперь функцию ВЫЧИСЛИ, которая действует по тому же принципу, но вместо списочной формы использует дерево в префиксной форме со всеми расставленными скобками. Значение выражения получается применением операции, указанной в узле, к поддеревьям, значения которых

уже предварительно вычислены рекурсивно по тому же алгоритму.

```
(defun вычисли (x &aux значение)
 (cond
   ((numberp x) x)
   ((atom x)
      (if (setq значение
                (get x 'значение))
          (вычисли значение)
          x))
   (t (примени (first x)
          (mapcar (function вычисли)
                  (cdr x))))))
```

Значения переменных будем хранить в списке свойств символов под именем ЗНАЧЕНИЕ.

Функция ПРИМЕНИ выполняет операцию из выражения над поддеревьями:

```
(defun примени (операция args
         &aux (op (get операция 'fn)))
   (if op (apply op args)
      (list op
         (вычисли (first args))
         (вычисли (second args)))))
```

Чтобы ее можно было осуществить, операция должна быть определена как функция, определение которой хранится под признаком FN в списке свойств символа, являющегося именем операции. В нашем случае для обозначения операции можно использовать те же символы (+, −, * и /), что и в самом Лиспе, но вкладывая в них немного измененный смысл. Это необходимо, например, во время упрощения записи выражения или чтобы возвратить операцию в символьном виде, когда выражение невозможно вычислить. Если операция неизвестна, то ПРИМЕНИ возвращает ее в форме списка, аргументы которого все-таки вычислены.

После этого надо только определить операцию. Используем макрос, упрощающий запись функций в список свойств:

```
(defmacro defдействие (действие args тело)
   `(setf (get ',действие 'fn)
      '(lambda ,args ,тело)))
```

Упрощение выражений

С целью упрощения арифметических выражений определим для операций правила упрощения. Будем учитывать только элементарные упрощения, связанные с константами 0 и 1 или с равенством аргументов. Функции упрощения для различных операций можно определить следующим образом:

```
(defдействие + (x y)
   (cond ((zerop x) y)
      ((zerop y) x)
      ((and (numberp x) (numberp y))
         (+ x y))
      (t `(+ ,x ,y))))

(defдействие - (x y)
   (cond ((zerop y) x)
      ((and (numberp x) (numberp y))
         (- x y))
      (t `(- ,x ,y))))

(defдействие * (x y)
   (cond ((equal x 1) y)
      ((equal y 1) x)
      ((or (zerop x) (zerop y)) 0)
      ((and (numberp x) (numberp y))
         (* x y))
      (t `(* ,x ,y))))
```

```
(defдействие / (x y)
   (cond ((zerop x) 0)
      ((zerop y) 'inf)
      ((equal x y) 1)
      ((and (numberp x) (numberp y))
         (/ x y))
      (t `(/ ,x ,y))))
```

Семантика команды двоеточие, предназначенной для разделения операторов и присваивания, несколько отличается от семантики арифметических операций, но и для нее применим использованный выше прием:

```
(defдействие : (x y) ; вычисли выражения x
   y) ; и y и верни значение y в качестве
      ; результата
(defдействие := (x y)
 (if (symbolp x)
   (setf (get x 'значение) y)
   (error "~%~S нельзя присвоить значение"
          x)))
```

Операцию дифференцирования d можно определить так:

```
(defдействие d (l x)
    (вычисли (дифференцируй1 l x)))
```

Функцию дифференцирования выражения ДИФФЕРЕНЦИРУЙ1 мы уже определили, когда рассматривали программирование, управляемое данными.

Снятие скобок и вывод

Нам нужна еще программа вывода результатов, которая преобразует выражение в префиксной форме, полученное в результате, в инфиксную форму и опускает в нем ненужные скобки. Лишними будут скобки, не изменяющие порядка выполнения операций:

```
(defun снять-скобки (x)
 (if (atom x)
   x
   (append
      (снять-у-оператора
            (first x) (second x))
      (list (first x))
      (снять-у-оператора
            (first x) (third x)))))

(defun снять-у-оператора
      (оператор выражение
      &aux (x (снять-скобки выражение)))
   (if (or (atom x)
           (старше оператор (second x)))
      (list x)
      x))
```

Диалог с Миксимой

Наконец необходима еще программа, поддерживающая диалог между пользователем и Миксимой:

```
(defun миксима (&aux выражение)
 (princ "Миксима: ")
 (loop
   (terpri)
   (princ "<= "
   (setq выражение (читай))
   (when (equal выражение '(конец))
      (return))
   (princ "=> ")
   (prin1 (снять-скобки (вычисли
               (анализируй nil '(nil)
                           выражение))))
```

```
_(миксима)
Миксима:
<= a := 4 ; b := c + a * d !
=> (C + 4 * D)
<= ...
```

Литература

1. Bobrow J.G. Symbol Manipulation Languages and Techniques. *Proceedings of the IFIP Working Conference,* Pisa, 1967.
2. Moses J. Algebraic Simplification: A Guide for the Perplexed. *JCACM,* Vol. 14, No. 8, Aug., 1971.
3. *Macsyma Reference Manual.* The Mathlab Group, Laboratory for Computer Science, MIT, 1977.
4. *Proceedings of the Macsyma User's Conference.* Berkeley, 1977; Washington DC, 1979.
5. Hearn A.C. et al. *Reduce 2 User's Manual.* University of Utah, Report No. UCP–19, Utah, 1974.
6. Husberg N. *Reduce 2 and Analitik-74, a Comparison.* Computing Centre, Helsinki University of Technology, Report No. 20, 1982.
7. Husberg N., Seppänen J. ANALITIK: Principal Features of the Language and its Implementation. *Proceedings of Eurosam '74, SIGSAM Bulletin,* Vol. 8, No. 3, Aug., 1974.
8. Seppänen J. *Symbolinen pöytälaskukone.* – В кн.: Seppänen J. *Lisp kielenopas,* Otadata, Espoo, 1972.
9. Winston P., Horn B. *Lisp.* Second Edition, Addison-Wesley, Reading, Massachusetts,1984.

Кто не знает чужих языков, тот ничего не знает и о своем языке.

И. Гёте

4.3 ЯЗЫК СПЛЕТНИКА

- **Исчезающие народные традиции**
- **Язык сплетника и цыганский жаргон**
- **Анализ правил и их программирование**
- **Выбор места разбиения слова на части**
- **Перевод слова и ключа**
- **Долгота и созвучие гласных**
- **Перевод слов и предложений**
- **Расширение до цыганского жаргона**
- **Литература**

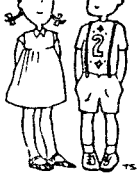
Обработка естественного языка (natural language processing) и программирование моделей языка являются одними из важнейших областей применения языка Лисп. Обработка языка принадлежит области, называемой *компьютерной лингвистикой* (computational linguistics) и лежащей между языкознанием, или лингвистикой, и вычислительной техникой.

Лисповские списочные структуры особенно хорошо подходят для описания языковых конструкций и проблем языка. Также хорошо сочетаются с решением этих проблем функциональное программирование и другие методы и формализмы программирования в области искусственного интеллекта и символьной обработки.

Однако в приводимом нами примере мы не будем иметь дело с типичными приложениями компьютерной лингвистики, такими как модели грамматик, методы анализа или синтеза предложений, а познакомимся с особым языковым явлением – с игровым или тайным языком. Хотя игровые и тайные языки, наверное, больше относятся к народным традициям, чем к

области языкознания, тем не менее они предлагают хорошие и увлекательные примеры и для программирования.

Исчезающие народные традиции



Особые игровые и тайные языки встречаются в культуре почти всех народов. В литературных языках они в основном утрачены, хотя для многих первобытнообщинных образований они все еще реальны. Эта очень редкая традиция сохранилась и в Финляндии. В начале века использовалось несколько десятков таких языков (Ojansuu 1907). Но и сейчас многие из них довольно известны и широко распространены, например среди школьников (Mäntylä 1970, Knuuttila 1977).

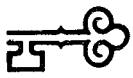
В настоящее время игровые и тайные языки относят к детской культуре (Virtanen 1972), но в свое время они занимали центральное место в различных культовых обрядах. Еще в средние века их использовали в основном нищие, бродяги, коробейники, а также злоумышленники и преступники с целью скрыть свои разговоры и замыслы от посторонних (Ojansuu 1916). Наиболее ранним письменным источником, упоминающим о тайном языке в Финляндии, является работа (Juslenius 1712), а первая классификация языков по типам была представлена Готтлундом (Gottlund 1818). Более подробные лингвистические исследования данной темы были сделаны в основном в работах (Ojansuu 1907 и 1916) и (Campbell 1976).

Язык сплетника и цыганский жаргон

Запрограммируем в качестве примеров язык сплетника и цыганский жаргон, которые являются, вероятно, самыми известными из наших игровых языков. Их правила не так уж просты, как может показаться вначале. Однако правила относительно легко формализовать и запрограммировать на компьютере. Может быть, вычислительная машина отчасти поможет

оживить эту интересную древнюю традицию и принесет пользу в обучении детей определенного возраста с целью развития у них сообразительности и чувства языка.

По своему типу язык сплетника и цыганский жаргон относятся к так называемым языкам с ключевым словом, так как в них используются секретные ключевые слова. Слова языка с ключевым словом получаются при перестановке слов определенным образом, к примеру начальных и конечных частей с частями ключевого слова. Обычно названия языков происходят от используемого в них ключевого слова. Например, ключевое слово языка сплетника – "kontti" (сплетня). В цыганском жаргоне в качестве ключевого словом всегда используется следующее слово.



Запрограммируем язык сплетника и цыганский жаргон. Программа транслирует предложения с финского языка на эти языки, опознавая видовые черты слов и применяя к ним определенные языком правила преобразования. Будем использовать методы функционального программирования. Примеры нескольких десятков других запрограммированных языков приведены в работе (Seppänen 1982).

Анализ правил и их программирование

Прежде чем программировать язык сплетника, нам надо проанализировать условия, преобразования слов, и правила согласования. Слово переводится на язык сплетника делением его и ключа 'kontti' по определенным правилам на две части. Затем из этих частей получаются слова на языке сплетника путем перестановки начальных частей слов между собой. Если части представлять в форме списка, то получим пример:

((si nä) (ko ntti)) → *((ko na) (si ntti))*
((aa mu) (ko ntti)) → *((koo mu) (an tti))*

Кроме того, в частях могут происходить изменения гласных по долготе и звучанию в зависимости от

формы слова и созвучия гласных в нем. Рассмотрим влияющие на это факторы по порядку.

Выбор места разбиения слова на части

Первая и важнейшая задача – это принятие решения о месте разбиения слова. Место разбиения необязательно приходится на границу слогов слова, хотя это иногда и случается. Это разбиение не обязательно попадает и на границу морфемы, оно зависит прежде всего от метрических и звуковых свойств слов. Приведем наиболее важные случаи и правила:

Начальный слог, кончающийся на гласную. Если начальный слог начинается с согласной и кончается на гласную, то слово делится по границе слога:

((sa na) (ko ntti)) → *((ko na) (sa ntti))*
((se) (ko ntti)) → *((ko) (se ntti))*

Начальный слог из короткой гласной. Если начальный слог образуется одиночной гласной, то слово делится также по границе слога:

((i so) (ko ntti)) → *((ko so) (i ntti))*

Начальный слог из долгой гласной. Если слово начинается с долгой гласной (двойная гласная), то обе гласные переходят к ключу:

((aa mu) (ko ntti)) → *((koo mu) (a ntti))*
((aa tto) (ko ntti)) → *((koo tto) (a ntti))*

Однако заметим, что в языке сплетника долгие гласные укорачиваются и короткие удлиняются так, чтобы ритмическая основа слова сохранилась.

Три или больше гласных. К ключу не переходит больше двух гласных:

((vaa an) (ko ntti)) → *((koo an) (va ntti))*

В составных словах могут быть три или даже больше последовательных гласных.

Дифтонги. Если в начальном слоге есть дифтонг (т.е. слог содержит последовательность разных гласных), то переходит (обычно) только первая гласная дифтонга:

((ki eli) (ko ntti)) → *((ko eli) (ki ntti))*
((ka uan) (ko ntti)) → *((ko uan) (ka ntti))*
((a ie) (ko ntti)) → *((ko ie) (a ntti))*

Для различных дифтонгов можно выделить более специфические случаи и их интерпретацию, которые могут зависеть от чувства языка самого говорящего. Эти случаи мы подробнее рассматривать не будем.

Для разбиения слова потребуется функция (ДЕЛИ *слово*), которая возвращает в качестве значения разбитые по правилам начальную и конечную части слова в виде списков букв.

```
;;; (дели 'kontti) -> ((#\K #\O) (#\N #\T
;;;                    #\T #\I))

(defun дели (слово)
   "Преобразует слово в список его букв"
   (дели-слово nil
      (coerce (string слово) 'list)))

(defun дели-слово (начало конец)
   "Делит слово на две части"
 (cond
   ((null конец) (list начало конец))
   ((согласная? (first конец))
      (дели-слово
         (в-конец начало (first конец))
         (rest конец)))
```

```
    ((долгая-нач? конец)
       (list (append начало
                     (list (first конец)
                           (second конец)))
             (cddr конец)))
    (t (list (в-конец начало (first конец))
             (rest конец)))))

(defun в-конец (список элемент)
   "Добавляет элемент в конец списка"
   (append список (list элемент)))
```

Функция ДЕЛИ-СЛОВО обрабатывает слово знак за знаком и в аргументе НАЧАЛО, который вначале пуст, собирает начальные буквы, строя из них список. По окончанию работы возвращается список, сформированный из двух списков – НАЧАЛО и КОНЕЦ. При делении слова используются предикаты ГЛАСНАЯ?, СОГЛАСНАЯ? и ДОЛГАЯ-НАЧ?:

```
(defun гласная? (буква)
   "Определяет, является ли буква гласной"
   (member буква *гласные*))
(setq *гласные* '(#\A #\E #\I #\O #\U #\Y
        #\ä #\ö)); ä и ö в Коммон Лиспе
(defun согласная? (буква)
"Определяет, является ли буква согласной"
   (not (гласная? буква)))
(defun долгая-нач? (слово)
   "Определяет, начинается ли слово с
   долгой гласной"
   (and (гласная? (first слово))
      (eql (first слово) ; EQL применяется
         (second слово)))) ; для сравнения
                           ; знаков
```

Перевод слова и ключа

Теперь мы можем определить функцию ПЕРЕВЕДИ-СЛОВО, которая в соответствии с ключом переводит слово на язык сплетника следующим образом:

```
;;; (переведи-слово 'sana 'kontti)
;;;       -> (kona santti)

(defun переведи-слово (слово ключ)
   "Переводит слово согласно ключу"
   (let ((частислова (дели слово))
         (частиключа (дели ключ)))
   (долгота-гласной (first частислова)
                    (second частислова)
                    (first частиключа)
                    (second частиключа))))
```

Долгота и созвучие гласных

Исходя из ритмики языка, долгая гласная в начале исходного слова приводит к удлинению гласной в начальной части ключа:

((luu ta) (ko ntti)) → *((koo ta) (lu ntti))*

Сохранение продолжительности гласной выполняется функцией ДОЛГОТА-ГЛАСНОЙ, которую мы определим в общем виде, в том числе и для слов, оканчивающихся на долгую гласную:

```
(defun долгота-гласной
       (начало1 конец1 начало2 конец2)
   "Заботится о сохранении долготы гласных
   при переводе"
```

```
(cond
  ((долгая-кон? начало1)
     (cond
        ((долгая-кон? начало2)
         (поменяй-части начало1 конец1
                        начало2 конец2))
        (t (поменяй-части
              (укороти начало1) конец1
              (удлини начало2) конец2))))
  ((долгая-кон? начало2)
     (поменяй-части
        (удлини начало1) конец1
        (укороти начало2) конец2))
  (t (поменяй-части начало1 конец1
                    начало2 конец2))))
```

Предикат ДОЛГАЯ-КОН? проверяет наличие долгой гласной в конце слова:

```
(defun долгая-кон? (слово)
   "Определяет, кончается ли слово на дол-
   гую гласную"
   (долгая-нач? (reverse слово)))
```

Короткие слоги удлиняются, а длинные укорачиваются функциями УДЛИНИ и УКОРОТИ:

```
(defun укороти (слог)
   "Удаляет последний элемент списка"
  (if (not (rest слог))
      nil
      (cons (first слог)
            (укороти (rest слог)))))

(defun удлини (слог)
   "Добавляет в конец списка его последний
   элемент"
```

```
(if (null (rest слог))
    (cons (first слог) слог)
    (cons (first слог)
          (удлини (rest слог)))))
```

Функция ПОМЕНЯЙ-ЧАСТИ меняет начальные части ключа и переводимого слова и вызывает функцию СОЗВУЧИЕ, которая приводит гласные к созвучию:

```
(defun поменяй-части
       (начало1 конец1 начало2 конец2)
   "Меняет начальные части слова и ключа"
   (list (созвучие начало1 конец1)
         (созвучие начало2 конец2)))
```

Созвучные слова в финском языке содержат либо гласные переднего, либо заднего ряда, т.е. в одном слове не может быть одновременно гласных как переднего (a o u), так и заднего ряда (ä ö y). В то же время гласные среднего ряда (e i) могут встречаться в одном и том же слове с гласными и переднего, и заднего рядов. Функция СОЗВУЧИЕ объединяет части слова, приводя при необходимости конечную часть слова в созвучие с начальной:

```
(defun созвучие (начало конец)
   "Согласовывает звучание конечной части
   слова"
  (cond ((переднее начало)
         (соедини начало (вперед конец)))
        (t (соедини начало (назад конец)))))
(defun переднее (слово)
   "Определяет звучание гласной в слове"
   (intersection слово передние-гласные))

(setq *передние-гласные* '(#\Y #\ä #\ö))
```

Заметьте, что буква Y – прописная, а ä и ö – строчные. Это следствие того, что ä и ö в коде ASCII не используются как буквы и Коммон Лисп не заменяет их в имени атомов на прописные.

Согласование звучания происходит при помощи функций ВПЕРЕД и НАЗАД:

```
(defun вперед (слово)
   "Преобразует задние гласные в передние"
  (sublis
    '((#\U . #\Y) (#\A . #\ä) (#\O . #\ö))
     слово))

(defun назад (слово)
   "Преобразует передние гласные в задние"
  (sublis
    '((#\Y . #\U) (#\ä . #\A) (#\ö . #\O))
     слово))
```

Функция (SUBLIS *а-список выражение*) – встроенная функция, которая заменяет вхождения в список ВЫРАЖЕНИЕ заданных в а-списке ключей на соответствующие им значения. Например:

```
_(sublis '((один . one) (два . two))
         '(один два three))
(ONE TWO THREE)
```

Перевод слов и предложений

Теперь созвучные части слова в форме списков букв можно соединить в атом функцией СОЕДИНИ:

```
(defun соедини (начало конец)
   "Строит символ из двух списков знаков"
   (intern (coerce (append начало конец)
                   'string)))
```

И далее мы уже можем попробовать перевод одиночных слов:

```
_(переведи-слово 'sana 'kontti)
(KONA SANTTI)
_(переведи-слово 'ääni 'kontti)
(KOONI ÄNTTI)
```

Чтобы можно было переводить целые предложения, определим еще функцию ПЕРЕВЕДИ-ПРЕДЛОЖЕНИЕ:

```
(defun переведи-предложение
        (предложение ключ)
   "Переводит слова предложения согласно
   ключу"
  (if (null предложение) nil
      (append
         (переведи-слово
            (first предложение) ключ)
         (переведи-предложение
            (rest предложение) ключ))))
```

```
_(переведи-предложение
   '(sinähän olet aika veitikka) 'kontti)
(KONAHAN SINTTI KOLET ONTTI KOIKA ANTTI
KOITIKKA VENTTI)
```

(На русском языке это выглядело бы так:

```
_(переведи-предложение
   '(ты ведь большой чудак) 'конти)
(КО ТЫНТИ КОДЬ ВЕНТИ КОЛЬШОЙ БОНТИ КОДАК
ЧУНТИ)
```

– *Перев.*)

Аналогичным образом, подбирая лишь новый ключ, можно определить другие родственные языку сплетника языки. Самыми известными являются, например, языки с ключевыми словами: vede, risa, raappa, punka, vis и lekkeri, а также с

шведским словом fikon. Их мы получим всего лишь простой заменой ключа:

```
_(переведи-предложение
   '(sinähän olet aika veitikka) 'vede)
(VENÄHÄN SIDE VELET ODE VEIKA ADE VEITIKKA
VEDE)
_(переведи-предложение
   '(kan du tala svenska) 'fikon)
(FIN KAKON FI DUKON FILA TAKON FINSKA
SVEKON)
```

Расширение до цыганского жаргона

При помощи функции ПЕРЕВЕДИ-СЛОВО можно также легко определить цыганский жаргон, в котором ключевым словом всегда является следующее слово. Если последнее слово остается без пары, то его можно переводить или в одиночку, или хотя бы с тем же словом kontti:

```
(defun нацыганский (предложение)
   "Переводит предложение на цыганский
   жаргон"
 (cond
   ((null предложение) nil)
   ((null (rest предложение))
      ;; последнее слово
      (переведи-слово
         (first предложение) 'kontti))
   (t (append
         (переведи-слово
            (first предложение)
            (second предложение))
         (нацыганский
            (cddr предложение))))))
```

```
_(нацыганский
   '(sinähän olet aika veitikka))
(ONAHAN SILET VEIKA AITIKKA)
_(нацыганский
   '(sinähän aika veitikka olet))
(ANAHAN SIIKA OITIKKA VELET)
_(нацыганский '(savolaiset kutsuvat
sellaista puhetta tyrän ompelukseksi))
(KUVOLAISET SATSUVAT PULLAISTA SEHETTA
ORAN TYMPELYKSEKSI)
```

Известно много и других видов игровых и тайных языков. Их можно грубо разделить на три класса: (1) слоговые языки, основанные на слогах (например так называемые птичьи языки), в которых некоторые слоги добавляются, повторяются, удаляются или замещаются или меняется их порядок; (2) неслоговые языки, такие, например, как только что представленные языки с ключевыми словами; (3) их различные сочетания.

Литература

1. Campbell L. *Generative Phonology vs. Finnish Phonology: Retrospect and Prospect.* In (Harms 1976).
2. Gottlund K.A. *De Proverbiis Fennicis Dissertatio.* Uppsala, 1818.
3. Harms T.R., Karttunen F. *Papers from the Transatlantic Finnish Conference.* Texas Linguistic Forum, No. 5, Dept of Linguistics, Texas University, Austin, 1976.
4. Juslenius D. *Suomenkielen sukulaisuudesta kreikan ja heprean kanssa.* Uppsala, 1712.
5. Ojansuu H. Suomen salakielistä. *Virittäjä,* No. 1, 1907.
6. Ojansuu H. Suomen kielen tutkimuksen työmaalta. Sarja esitelmiä 1., Gummerus, Jyväskylä, 1916.
7. Knuuttila S. *Leena Heinosen pihaperinne.* Lisensiaatfityö, Helsingin yliopisto, 1977.

8. Mäntylä A.-L. Leikkikielistä. Seminaariesitelmä. Kansanrunoustieteen laitos, Helsingin yliopisto, 1970.
9. Seppänen J. Sananmuodostus leikki- ja salakielissä. Sananmuodostuksen ongelmia, Suomen kielitieteellisen yhdistyksen julkaisuja, No. 7, 1981.
10. Seppänen J. Computing Families of Natural Secret Languages. An Exercise in Functional Linguistics, Report No. 23, TKK Laskentakeskus, 1982.
11. Seppänen J. Recursive Functions for Computation of Natural Secret Languages. Coling–82, Prague, 1982.
12. Seppänen J. Обучая ЭВМ фольклору. *The Sixth International Finno-Ugric Congress,* Сыктывкар, 1985.
13. Seppänen J. Tietokoneestako kieliniekka? Ohjelmoimme Leikki- ja salakieliä, Noppi, Nokian opetusjärjestelmien asiakaslehti, 2/1985.
14. Virtanen L. *Antti pantti pakana.* WSOY, Porvoo, 1972.

Специалист – это человек, который перестал мыслить. Он только знает.

Ф. Райт

4.4 ДАРВИН

- **Структура экспертной системы**
- **Представление знаний**
- **Машина вывода**
- **Факты и правила**
- **Правила вывода базы знаний**
- **Стратегия обратного вывода**
- **Работа системы Дарвин**
- **Примеры запросов**
- **Расширение системы Дарвин**
- **Литература**

Экспертная система (expert system, knowledge based system) – это программная система, знания и умения которой сравнимы с умением и знаниями специалистов в какой-нибудь специальной области знаний. Экспертные системы вместе с системами обработки естественных языков являются наиболее важными в коммерческом плане областями использования искусственного интеллекта. Многие системы естественного языка можно считать и экспертными системами. Они являются экспертами в области лингвистики и, возможно, общего языкознания. Многие алгебраические системы также считаются экспертными системами.

В рамках исследований искусственного интеллекта созданы многочисленные экспертные системы для разных областей знания, таких, например, как медицинская диагностика и обследование пациентов, генные и молекулярные исследования, составление конфигурации вычислительных машин, образование,

поиск неисправностей в устройствах и системах и многие другие практические приложения. В этой главе мы запрограммируем небольшую "зоологическую" экспертную систему (Winston и Horn 1981), которую назовем Дарвин.

Структура экспертной системы

Типичная экспертная система состоит из двух главных частей: *машины вывода* (inference engine) и *базы знаний* (knowledge base). База знаний содержит данные и знания из области применения; способ решения проблем, используемый в машине вывода, не привязан к *данным из предметной области* (domain knowledge). Кроме того, в систему обязательно входит какой-нибудь язык взаимодействия человека с машиной, посредством которого пользователь-специалист ведет диалог с системой, а также в нее входят средства, при помощи которых *инженер знаний* (knowledge engineer) и эксперт(ы) (expert) поддерживают базу знаний.

Представление знаний

Для представления знаний используют различные формализмы и языки представления данных. Наиболее часто встречается представление знаний с помощью правил типа ЕСЛИ–ТО. В системе Дарвин правила, описывающие принятие решения, можно задавать в форме, похожей на естественный язык:

```
(ЕСЛИ условие-1
      И условие-2
      ...
      И условие-i
      ТО вывод-1
      И вывод-2
      ...
      И вывод-j)
```

Условия и выводы – это простые предложения естественного языка. Например:

```
(ЕСЛИ на лампочку подано напряжение
    И лампочка не горит
    ТО лампочка, вероятно, перегорела)

(ЕСЛИ читатель перестал понимать
    И читатель хочет учиться
    И читатель еще может читать
    ТО читателю нужно начать все сначала
    И читателю следует быть повнимательней)
```

Машина вывода

Машина вывода – это универсальный механизм (программа или аппарат), который с помощью правил базы знаний строит новые выводы, задает дополнительные вопросы и так далее до тех пор, пока не придет к какому-нибудь приемлемому конечному результату или ответу. Исходные данные и выводы, полученные в результате принятия решений, в дальнейшем будем называть известными системе *фактами* (fact).

В более крупных системах база знаний содержит помимо правил знания и других типов: факты об объектах проблемной области и их свойства, данные об обстоятельствах и событиях, иерархии понятий, метаданные и т.д.

Работа экспертных систем основывается в первую очередь на обширной базе знаний. Работа машин вывода часто довольно проста и прямолинейна. Рассмотрим более подробно структуру машины принятия решений программы Дарвин.

Факты и правила

В системе Дарвин факты представляются просто в виде списков символов. Например, следующий список мог бы описывать наши знания о каком-нибудь животном:

```
((животное имеет шерсть)
 (животное полосатое)
 (животное жвачное))
```

База знаний системы Дарвин образуется из списков, подобных ранее описанным правилами ЕСЛИ-ТО. Например:

```
(setq правило12
   '(ЕСЛИ животное жвачное
      И животное полосатое
      ТО животное зебра))
```

Чтобы правила можно было свести к фактам, их условия и выводы необходимо представлять списками фактов. Являющиеся правилами предложения можно для этого преобразовать в структуры, которые состоят из имени правила и условий и выводов, представленных в виде списка фактов:

```
(defstruct правило имя условия выводы)
```

Соответствующую предыдущему правилу ПРАВИЛО12 структуру можно теперь создать при помощи вызова:

```
(make-правило
   :имя      'правило12

   :условия '((животное жвачное)
              (животное полосатое))
   :выводы  '((животное зебра)))
```

Если условия правила являются элементами какого-нибудь списка фактов, то применение правила к этому списку фактов можно реализовать путем добавления в список выводов из правила. Например, применив ПРАВИЛО12 к ранее представленному списку данных, можно сделать заключение: животное о котором идет речь – это зебра.

Правила вывода базы знаний

Знания системы Дарвин о животных содержатся в базе знаний, содержащей следующие 14 правил:

```
(setq ПРАВИЛО1
   '(ЕСЛИ ЖИВОТНОЕ ИМЕЕТ ШЕРСТЬ
      ТО ЖИВОТНОЕ МЛЕКОПИТАЮЩЕЕ))
(setq ПРАВИЛО2
   '(ЕСЛИ ЖИВОТНОЕ КОРМИТ ДЕТЕНЫШЕЙ
   МОЛОКОМ ТО ЖИВОТНОЕ МЛЕКОПИТАЮЩЕЕ))
(setq ПРАВИЛО3 '(ЕСЛИ ЖИВОТНОЕ ИМЕЕТ ПЕРЬЯ
      ТО ЖИВОТНОЕ ПТИЦА))
(setq ПРАВИЛО4
   '(ЕСЛИ ЖИВОТНОЕ УМЕЕТ ЛЕТАТЬ
      И ЖИВОТНОЕ НЕСЕТ ЯИЦА
      ТО ЖИВОТНОЕ ПТИЦА))
(setq ПРАВИЛО5
   '(ЕСЛИ ЖИВОТНОЕ ЕСТ МЯСО
      ТО ЖИВОТНОЕ ХИЩНИК))
(setq ПРАВИЛО6
   '(ЕСЛИ ЖИВОТНОЕ ИМЕЕТ ОСТРЫЕ ЗУБЫ
      И ЖИВОТНОЕ ИМЕЕТ КОГТИ
      И ГЛАЗА ЖИВОТНОГО ПОСАЖЕНЫ ПРЯМО
      ТО ЖИВОТНОЕ ХИЩНИК))
(setq ПРАВИЛО7
   '(ЕСЛИ ЖИВОТНОЕ МЛЕКОПИТАЮЩЕЕ
      И ЖИВОТНОЕ ИМЕЕТ КОПЫТА
      ТО ЖИВОТНОЕ ЖВАЧНОЕ))
(setq ПРАВИЛО8
   '(ЕСЛИ ЖИВОТНОЕ МЛЕКОПИТАЮЩЕЕ
      И ЖИВОТНОЕ ЖУЕТ ЖВАЧКУ
      ТО ЖИВОТНОЕ ЖВАЧНОЕ))
(setq ПРАВИЛО9
   '(ЕСЛИ ЖИВОТНОЕ МЛЕКОПИТАЮЩЕЕ
      И ЖИВОТНОЕ ХИЩНИК
      И ЖИВОТНОЕ ЖЕЛТО-КОРИЧНЕВОЕ
      И ЖИВОТНОЕ ИМЕЕТ ТЕМНЫЕ ПЯТНА
      ТО ЖИВОТНОЕ ГЕПАРД))
(setq ПРАВИЛО10
   '(ЕСЛИ ЖИВОТНОЕ МЛЕКОПИТАЮЩЕЕ
```

```
        И ЖИВОТНОЕ ХИЩНИК
        И ЖИВОТНОЕ ЖЕЛТО-КОРИЧНЕВОЕ
        И ЖИВОТНОЕ ПОЛОСАТОЕ
        ТО ЖИВОТНОЕ ТИГР))
(setq ПРАВИЛО11
   '(ЕСЛИ ЖИВОТНОЕ ЖВАЧНОЕ
        И ЖИВОТНОЕ ДЛИННОШЕЕЕ
        И ЖИВОТНОЕ ДЛИННОНОГОЕ
        И ЖИВОТНОЕ ИМЕЕТ ТЕМНЫЕ ПЯТНА
        ТО ЖИВОТНОЕ ЖИРАФ))
(setq ПРАВИЛО12
   '(ЕСЛИ ЖИВОТНОЕ ЖВАЧНОЕ
        И ЖИВОТНОЕ ПОЛОСАТОЕ
        ТО ЖИВОТНОЕ ЗЕБРА))
(setq ПРАВИЛО13
   '(ЕСЛИ ЖИВОТНОЕ ПТИЦА
        И ЖИВОТНОЕ НЕ УМЕЕТ ЛЕТАТЬ
        И ЖИВОТНОЕ ДЛИННОШЕЕЕ
        И ЖИВОТНОЕ ДЛИННОНОГОЕ
        И ЖИВОТНОЕ ЧЕРНО-БЕЛОЕ
        ТО ЖИВОТНОЕ СТРАУС))
(setq ПРАВИЛО14
   '(ЕСЛИ ЖИВОТНОЕ ПТИЦА
        И ЖИВОТНОЕ НЕ УМЕЕТ ЛЕТАТЬ
        И ЖИВОТНОЕ ПЛАВАЕТ
        И ЖИВОТНОЕ ЧЕРНО-БЕЛОЕ
        ТО ЖИВОТНОЕ ПИНГВИН))
```

Описанные выше правила легко понять программисту и создателю системы, и поэтому они не требуют описания. Следующая функция АНАЛИЗИРУЙ берет список правил, анализирует каждое правило и преобразует его к ранее описанной структуре (ПРАВИЛО), полями которой являются: имя правила, условия и выводы:

```
(defun присоедини (x y)
   (append x (list y)))
```

```
(defun анализируй (правила)
   "Анализирует правила и составляет из
   них список"
   (mapcar #'анализируй-правило правила))
(defun анализируй-правило (имя)
   "Преобразует правило в форме предложе-
   ния в структуру"
   (let ((правило (eval имя)))
      (make-правило :имя имя
         :условия (условия правило)
         :выводы (выводы правило))))
(defun условия (предложение)
   "Возвращает условия в виде списка"
 (предложение-и (cdr предложение)nil nil))
(defun выводы (предложение)
   "Возвращает выводы в виде списка"
 (предложение-и
   (cdr (member 'то предложение))nil nil))
(defun предложение-и
      (предложение часть результат)
   "Выделяет предложения, разделяемые сим-
   волом И"
   (cond
      ((null предложение)
         ;; условие окончания
         (присоедини результат часть))
      ((eq (car предложение) 'то)
         ;; условие окончания
         (присоедини результат часть))
      ((eq (car предложение) 'и)
         ;; новое предложение
         (предложение-и (cdr предложение)
            nil
            (присоедини результат часть)))
      (t (предложение-и (cdr предложение)
         (присоедини часть ; следующее
            (car предложение))
            результат))))  ; слово
```

Правила в виде записей легко обрабатывать, и функциям более высокого уровня не нужно знать об анализе данных или деталях представления правил. Впоследствии это облегчает возможную модификацию программ и их дальнейшее развитие. Если бы правила представлялись, например, в виде списков, состоящих из условий и выводов, то на более высоком уровне пришлось бы иметь информацию о порядке следования элементов в списке. Теперь же на части правил можно указывать логическим именем.

Правила системы Дарвин хранятся в списке *БАЗА-ЗНАНИЙ*:

```
(setq *база-знаний*
    '(ПРАВИЛО1 ПРАВИЛО2 ПРАВИЛО3 ПРАВИЛО4
    ПРАВИЛО5 ПРАВИЛО6 ПРАВИЛО7 ПРАВИЛО8
    ПРАВИЛО9 ПРАВИЛО10 ПРАВИЛО11 ПРАВИЛО12
    ПРАВИЛО13 ПРАВИЛО14))
```

Проанализированные варианты правил сохраним в переменной *ПРАВИЛА* следующей командой:

```
(setq *правила* (анализируй *база-знаний*))
```

Значением переменной *ПРАВИЛА* будет список, состоящий из структур.

Предположим, что система Дарвин сохраняет известные ей факты в списке *ФАКТЫ*, состоящем из элементов данных. В этом случае применимость правила можно проверить функцией ПРОВЕРЬ-ПРАВИЛО и выводы правила можно при необходимости добавить к фактам функцией ДОБАВЬ-ВЫВОДЫ:

```
(defun проверь-правило (правило)
    "Проверяет применимо ли правило"
    (подмножество
        (правило-условия правило) *факты*))
```

```
(defun подмножество
        (подмножество множество)
  "Проверяет, является ли множество
  подмножеством"
 (equal подмножество
  (intersection подмножество множество)))

(defun добавь-выводы (правило)
"Расширяет список фактов выводами правила"
  (do ((выводы (правило-выводы правило)
               (cdr выводы)))
      ((null выводы) *факты*)
      (if (member (car выводы) *факты*)
          nil
          (format t "Согласно правилу ~S :"
                  (правило-имя правило))
      (выведи-элементы (car выводы))
      (push (car выводы) *факты*))))

(defun выведи-элементы (список)
  "Выводит элементы списка"
  (mapc #'(lambda (элемент)
            (princ элемент) (princ " "))
        список)
  (terpri) t)
```

Стратегия обратного вывода

Машина вывода применяет правила к известным в текущий момент системе фактам для нахождения новых фактов до тех пор, пока в результате применения не будет получен искомый результат. Если у системы недостаточно информации для дальнейшего поиска решения, то она запрашивает у пользователя дополнительную информацию и сохраняет ее в своем списке фактов, а затем пытается еще раз применить к своим правилам новые дополнительные факты и т.д.

В исследованиях по искусственному интеллекту созданы различные способы нахождения решения и выполнения действия. В системе Дарвин мы применим *обратный вывод* (backward chaining), в котором решение пытаются найти, идя в обратном направлении от конечного результата.

Работа системы Дарвин

Систему Дарвин на самом верхнем уровне можно представить как распознавателя животных, который пытается на основе своих правил доказать некоторую гипотезу.

Представим возможные конечные результаты в виде списка *ГИПОТЕЗЫ* :

```
(setq *гипотезы*
 '((ЖИВОТНОЕ ПИНГВИН)
   (ЖИВОТНОЕ СТРАУС)
   (ЖИВОТНОЕ ЗЕБРА)
   (ЖИВОТНОЕ ЖИРАФ)
   (ЖИВОТНОЕ ТИГР)
   (ЖИВОТНОЕ ГЕПАРД)))
```

Все эти гипотезы встречаются в части вывода некоторых правил. При обратном выводе условия этих правил можно интерпретировать как новые гипотезы, если вывод является конечным результатом, и так далее. Так система порождает гипотезы более низкого уровня до тех пор, пока не обнаружит, что новых правил для порождения гипотез больше нет. Тогда система запрашивает условия правила непосредственно у пользователя, которые он на этом уровне уже, вероятно, сможет задать сам и не будучи специалистом, затем система с помощью новой информации пытается продвинуться дальше в своих выводах:

```
;;; Глобальные динамические переменные
(defvar *гипотезы*)   ; значение определено
                      ; выше
```

```
(defvar *правила*)    ; значение определено
                      ; выше
;;; Главная программа Дарвин

(defun Дарвин ()
   "Экспертная система для распознавания
   животных"

   ;; внутренние динамические переменные
   ;; определим формой DECLARE

   (declare
      (special *факты*  ; найденные факты
          *запросы*))   ; утверждения, о
                        ; которых задавался
                        ; вопрос пользователю
   (знаток-зверей *гипотезы*))

(defun знаток-зверей (гипотезы)
"Пытается проверить какую-нибудь гипотезу"
 (cond
   ((null гипотезы)
      "Не могу доказать никакую из извест-
      ных мне гипотез")
   ((докажи (car гипотезы))
      (car гипотезы))   ; результат
   (t (знаток-зверей (cdr гипотезы)))))
                        ; новая попытка
```

Доказательство гипотезы или утверждения, можно выполнить при помощи следующей процедуры:

Процедура Докажи:

Дано: Доказываемая гипотеза.

Значение: Значение функции – истина, если данную гипотезу можно доказать.

Действия:

1. Если гипотеза уже есть в списке фактов, значит она доказана.

2. Если это не так, то соберем все правила, с помощью которых можно было бы доказать гипотезу, т.е. те правила, в части вывода которых есть проверяемая гипотеза. Если какое-нибудь из этих правил можно прямо применить к списку фактов, то, значит, гипотеза доказана.

   Если никакое из собранных правил нельзя применить, то попытаемся доказать применимость какого-либо правила, доказав все условия правила, взяв их как новые гипотезы и рекурсивно применив процедуру Докажи.

3. Если описанные действия не дают результата, то спросим у пользователя, верна ли гипотеза (если только это уже не спрашивалось), и если она верна, то присоединим утверждение к списку фактов. Присоединим утверждение к списку *ЗАПРОСЫ* независимо от ответа, чтобы потом это же самое не пришлось повторно спрашивать.

Эту рекурсивную процедуру можно непосредственно программировать. Все три ветви вышеописанной процедуры помечены в определении соответствующими комментариями:

```
(defun докажи (гипотеза)
 (let ((правила))
   (cond ((member гипотеза *факты*)
          t)                              ; ветвь 1
         ((setq правила
            (возможные гипотеза))         ; ветвь 2
          (if (прямо гипотеза правила) t
           (рекурсивно гипотеза правила)))
         (t (cond
              ((member гипотеза *запросы*)
                nil)                      ; ветвь 3
```

```
          ((and
             (princ "Это правда, что: ")
             (выведи-элементы гипотеза)
             (eq (read) 'Да))
            (setq *факты*
               (union (list гипотеза)
                      *факты*)) t)
          (t (push гипотеза *запросы*)
             nil))))))
(defun возможные (гипотеза)
   "Возвращает правила, доказывающие по-
   тенциальную гипотезу"
   (mapcan
      #'(lambda (x)
            (if (member (правило-выводы x)
                  гипотеза :test #'equal)
                (list x)))
      *правила*))
(defun прямо (гипотеза возможные)
   "Проверяет, можно ли доказать гипотезу
   непосредственно при помощи какого-
   нибудь правила"
 (cond
   ((null возможные) nil)
   ((null *факты*) nil)
   ((проверь-правило (car возможные))
      (добавь-выводы (car возможные)))
   (t (прямо гипотеза (cdr возможные)))))
(defun рекурсивно (гипотеза возможные)
   "Рекурсивно проверяет гипотезу"
 (cond
   ((null возможные) nil)
   ((проверь-непрямо
         (правило-условия
               (car возможные)))
      (добавь-выводы (car возможные)))
   (t (рекурсивно гипотеза
               (cdr возможные)))))
```

```
(defun проверь-непрямо (условия)
   "Рекурсивно проверяет все условия"
   (every #'докажи условия))
```

На основании своих правил программа Дарвин может задавать пользователю лишь разумные с точки зрения гипотезы вопросы. Однако первую гипотезу приходится выбирать наобум.

Примеры запросов

Теперь немного побеседуем с программой Дарвин. В первом примере пользователь видит перед собой пингвина, во втором – зебру, но животных он не знает. С помощью программы Дарвин он может определить виды этих животных, давая системе простые ответы на ее вопросы:

```
_(Дарвин)
Это правда, что: ЖИВОТНОЕ ИМЕЕТ ПЕРЬЯ
                  ; вопрос
Да                ; ответ пользователя
Согласно правилу ПРАВИЛО3 : ЖИВОТНОЕ ПТИЦА
Это правда, что: ЖИВОТНОЕ НЕ УМЕЕТ ЛЕТАТЬ
Да
Это правда, что: ЖИВОТНОЕ УМЕЕТ ПЛАВАТЬ
Да
Это правда, что: ЖИВОТНОЕ ЧЕРНО-БЕЛОЕ
Да
Согласно правилу ПРАВИЛО14 : ЖИВОТНОЕ
ПИНГВИН
(ЖИВОТНОЕ ПИНГВИН)          ; результат

_(Дарвин)
Это правда, что: ЖИВОТНОЕ ИМЕЕТ ПЕРЬЯ
Нет
Это правда, что: ЖИВОТНОЕ УМЕЕТ ЛЕТАТЬ
Нет
```

```
Это правда, что: ЖИВОТНОЕ ИМЕЕТ ШЕРСТЬ
Да
Согласно правилу ПРАВИЛО1 : ЖИВОТНОЕ
МЛЕКОПИТАЮЩЕЕ
Это правда, что: ЖИВОТНОЕ ИМЕЕТ КОПЫТА
Да
Согласно правилу ПРАВИЛО7 : ЖИВОТНОЕ
ЖВАЧНОЕ
Это правда, что: ЖИВОТНОЕ ПОЛОСАТОЕ
Да
Согласно правилу ПРАВИЛО12 : ЖИВОТНОЕ
ЗЕБРА
(ЖИВОТНОЕ ЗЕБРА)
```

Расширение системы Дарвин

Похожие на Дарвина *консультирующие* (consulting) экспертные системы обычно способны в какой-то мере обосновывать разумность своих вопросов и сделанных выборов, отвечая на поставленные пользователем вопросы. Обосновывающую и другую информацию можно запросить во время диалога, задавая вопросы типа: ЧТО-ЗНАЕШЬ-ОБ, ПОЧЕМУ-СПРАШИВАЕШЬ и КАК-ТЫ-РЕШИЛ-ЧТО. Так пользователь системы может убедиться в разумности действий системы и может сам обучать систему, вводя в нее свои знания.

Дополнить систему Дарвин элементами такого рода довольно просто. На вопрос ЧТО-ЗНАЕШЬ-ОБ программа могла бы отвечать перечислением известных ей фактов или подходящим их подмножеством. На вопрос ПОЧЕМУ-СПРАШИВАЕШЬ можно было бы сообщить выводы правила, относящегося к доказываемой гипотезе (условию) или выдать всю цепочку выводов. На вопрос КАК-ТЫ-РЕШИЛ-ЧТО можно было бы в удобной форме вывести в качестве ответа цепочку выводов утверждения (гипотезы), относящуюся к вопросу.

Систему Дарвин можно было бы расширить и так, чтобы имелась возможность с помощью переменных задавать правила в более общей форме точно так же, как это делалось в ранее рассмотренном сопоставлении с образцом или в интерпретаторе Пролога. Например:

```
(ЕСЛИ (? x) это некоторый (? y)
   и (? y) цвета (? z)
ТО (? x) цвета (? z))
```

Было бы также полезно, чтобы в качестве базы знаний и фактов можно было наряду с правилами и простыми элементами данных использовать и другие виды представления данных, такие как иерархии понятий, объекты и другие.

Оставляем читателю возможность развития этих идей и их реализацию, а также применение машины вывода системы Дарвин к другим базам знаний.

Литература

1. Charniak E., Riesbeck C. и McDermott D. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1979.
2. Hyvönen E. *Asiantuntijajärjestelmien tietämystekniikka*. Knowledge Engineering Ky, 1986, Helsinki.
3. Masinter L., Shrobe H. AI Programming technology: Languages and Machines. AAAI–83 Conference Tutorial, Washington D.C., 1983.
4. O'Shea T., Eisenstadt M. (Eds.) *Artificial Intelligence: Tools, Techniques and Applications*. Harper et. Row, New York, 1984.
5. Shapiro S. *Techniques of Artificial Intelligence*. D. Van Nostrand Company, New York, 1979.
6. Thompson H., Richie G. Implementing Natural Language Parsers. In (O'Shea and Eisenstadt 1984).
7. Winston P. и Horn B. *Lisp*. Addison-Wesley, Reading, Massachusetts, 1981 (первое издание), 1984 (второе издание).

Вселенная – это великий символ Бога.

Т. Карлейль

4.5 СОЛНЕЧНАЯ СИСТЕМА

- **Сначала были созданы небо и Земля**
- **Окно в космос**
- **Солнце, планеты и спутники**
- **И все-таки она вертится**
- **Вращением спутника управляет демон**
- **Создание небесных тел**
- **Запуск Солнечной системы**
- **Литература**

В этой главе в качестве примера объектного программирования запрограммируем астрономическую модель, представляющую и делающую более наглядным строение и функционирование Солнечной системы (Symbolics 1984). Модель описывает Солнце и планеты Меркурий, Венеру, Землю, Марс и их спутники. При программировании будем использовать Зеталисп Лисп-машины и систему Flavor. Цель главы – прояснение принципов и возможностей объектного программирования. Что касается деталей Зеталиспа, то сошлемся лишь на описание (Weinreb и Moon 1981).

Система работает следующим образом: на экране открывается окно, и в нем появляется мигающее Солнце. Планеты со своими спутниками располагаются вокруг Солнца на своих астрономических местах. Планеты начинают вращаться вокруг Солнца по своим орбитам с правильным соотношением скоростей. В то же время спутники начинают вращаться вокруг своих планет по траекториям, складывающимся из двух вращательных движений: вращения планеты вокруг Солнца и вращения спутника вокруг своей планеты.

Сначала были созданы небо и Земля

Сначала определим центр Солнца и Солнечной системы как точку на экране с координатами (375, 375):

```
(defconst солнце-x 375)
(defconst солнце-y 375)
```

Другие константы – это диаметр Солнца и радиус орбиты Земли:

```
(defconst диаметр-солнца 10)
(defconst орбита-земли 149)
```

Планеты и их спутники определим как переменные:

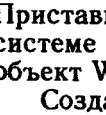
```
(defvar меркурий)
(defvar венера)     ; Вечерняя звезда
(defvar земля)
(defvar луна)       ; Спутник Земли
(defvar марс)
(defvar фобос)      ; У Марса два спутника
(defvar деймос)
```

Окно в космос

"Окно в космос" можно определить с помощью оконной системы Зеталиспа как объект-окно простого типа TV:WINDOW:

```
(defflavor космос-окно () (tv:window))
```

Приставка TV: указывает на используемое в оконной системе пространство имен TV, которое содержит объект WINDOW.



Создадим окно TV:WINDOW функцией TV:MAKE-WINDOW и присвоим его значение переменной КОСМОС, чтобы обеспечить себе возможность в дальнейшем ссылаться на него. Аргументами вызова TV:MAKE-WINDOW будут тип создаваемого окна КОСМОС-ОКНО,

а также заголовок окна (:LABEL), координаты углов окна (:EDGES). Благодаря параметру :BLINKER-P = NIL окно создается без курсора.

```
(setq космос
   (tv:make-window 'космос-окно
           ':label "Внутренние_планеты"
           ':blinker-p nil
           ':edges '(0 0 749 749)))
```

Солнце, планеты и спутники

Чтобы обобщить определения разных небесных тел, определим базовый объект с именем ТЕЛО:

```
(defflavor тело
  ((x 0)          ; Текущие координаты
   (y 0)          ; тела
   (центр-x x) ; Центр, вокруг которого
   (центр-y y) ; тело вращается
    радиус-орбиты ; Радиус орбиты
   (след nil)      ; Если СЛЕД есть T, то
                  ; тело оставляет за собой
                  ; линию, состоящую из точек
   (угол 1)       ; Угол обращения тела
    скорость      ; Относительная скорость
                  ; тела (= 0.1) относительно
                  ; скорости Земли
    размер)       ; Диаметр тела
   ()             ; Не принадлежит другим
                  ; классам объектов
    :settable-instance-variables)
                  ; Можно присваивать
                  ; переменным состояния
```

Планеты и спутники так же, как и Солнце, – это небесные тела. Их можно определить как подкласс тел, другими словами, как подаромат аромата ТЕЛО:

```
(defflavor планета ((спутник-список nil))
   (тело)
   :settable-instance-variables)
(defflavor спутник ()
   (тело))
```

Для объектов типа ПЛАНЕТА к переменным состояния объекта типа ТЕЛО добавлена переменная СПИСОК-СПУТНИКОВ, в которой будем сохранять спутники каждой планеты. По умолчанию у планеты нет спутников, СПИСОК-СПУТНИКОВ = NIL.

И все-таки она вертится

Галилео Галилей.

Вращение как планет, так и спутников вокруг центрального тела происходит по одним и тем же законам природы. Для планет телом, вокруг которого они вращаются, является Солнце, а для каждого спутника – некоторая планета. Это движение для всех небесных тел можно определить одним свойством, связанным с телом методом :ВРАЩАЙСЯ:

```
(defmethod (тело :вращайся) ()
  (let ((относит-угол скорость)
        перемещение); Угловое изменение
                    ; положения тела
    (loop until (= относит-угол 0)
       do
       (if (> относит-угол 10)
          (progn
             (setq относит-угол
                (- относит-угол 10))
             (setq перемещение 10))
          (progn (setq перемещение
                       относит-угол)
             (setq относит-угол 0)))
```

```
        (setq угол (+ угол перемещение))
        (send s-w :draw-filled-in-circle x y
            размер tv:alu-xor)
        (setq x (fix (+ центр-x
                        (* sind градус)
                        радиус-орбиты))))
        (setq y (fix (- центр-y
                        (* (cosd градус)
                        радиус-орбиты))))
        (send s-w :draw-filled-in-circle x y
              размер tv:alu-xor))))
```

Идея метода состоит в осуществлении движения тела наращиванием углового перемещения с шагом в десять градусов (переменная УГОЛ). Перемещение каждого тела вычисляется в виде относительной величины, зависящей от значения его скорости (переменная СКОРОСТЬ). С помощью функций SIND и COSD (синус и косинус) при каждом изменении угла вычисляются новые координаты положения планеты.

Чтобы можно было нарисовать в космосе точку, соответствующую небесному телу (окно КОСМОС), полученные значения округляются функцией FIX до целых чисел. Рисование осуществляется активацией в окне КОСМОС системного метода, заполняющего окружность :DRAW-FILLED-IN-CIRCLE, которому в качестве аргументов передаются координаты x и y центра закрашиваемой окружности, диаметр и параметр цвета, с помощью которого определяется логика и изменение цвета точек экрана – TV:ALU-XOR. Вследствие того, что закрашивание происходит через каждые десять градусов, небесное тело выглядит на экране равномерно двигающимся кружком.

Вращением спутника управляет демон

Во время обращения планет необходимо следить кроме их перемещения и за тем, чтобы менялся центр обращения спутников планет. Это можно запрограммировать при помощи следящего демона, входящего в метод :ВРАЩАЙСЯ. Демон в соответствии с перемещением

планеты изменяет центр обращения спутника и запускает вращение спутника с прежнего углового положения.

Следящий за перемещением центра обращения демон определен только для объектов типа ПЛАНЕТА, но не для объектов типа ТЕЛО, как метод :ВРАЩАЙСЯ. Перемещение спутника происходит только тогда, когда метод :ВРАЩАЙСЯ применяется к объекту типа ПЛАНЕТА:

```
(defmethod (планета :after :вращайся) ()
 (loop for спутник in спутник-список do
   ;; Установка координат центра обращения
   ;; объекта СПУТНИК.
   (send спутник :set-центр-x x)
   (send спутник :set-центр-y y)
   ;; Изменить положение спутников.
   (send спутник :вращайся)))
```

Создание небесных тел

Теперь можно определить функцию СОЛНЕЧНАЯ-СИСТЕМА, создающую Солнечную систему и поддерживающую ее функционирование:

```
(defun солнечная-система ()
 ;; Подготовка окна
 (send космос ':expose)
 (send космос :clear-window)
 ;; Очерчивание солнечного круга
 (send космос :draw-circle
              солнце-x солнце-y
              диаметр-солнца tv:alu-xor)
 ;; Отметка орбиты Земли
 (send космос :draw-circle
            x y орбита-земли tv:alu-xor)
```

```
(setq марс
  (make-instance 'планета
                 ':радиус 228
                 ':скорость .053
                 ':размер 4))
(setq фобос
  (make-instance 'спутник
                 ':центр-x (send марс :x)
                 ':центр-y (send марс :y)
                 ':радиус 7
                 ':скорость 114.4
                 ':размер 1))
;; Создание планет и спутников
(setq меркурий
  (make-instance 'планета
                 ':радиус 58
                 ':скорость .416
                 ':размер 3))
(setq венера
  (make-instance 'планета
                 ':радиус 108
                 ':скорость .416
                 ':размер 5))
(setq земля
  (make-instance 'планета
                 ':радиус радиус-земли
                 ':скорость .1
                 ':размер 6))
(setq луна
  (make-instance 'спутник
           ':центр-x      ; Определяется
           (send земля :x); по Земле
           ':центр-y
           (send земля :y)
           ':радиус 15
           ':trace t      ; Спутник ос-
           ':скорость 1.3 ; тавляет след
           ':размер 2))
```

```
(setq деймос
  (make-instance 'спутник
                 ':центр-x (send марс :x)
                 ':центр-y (send марс :y)
                 ':радиус 12
                 ':скорость 30.4
                 ':размер 1))
;; Присваивание спутников Земле и Марсу
(send земля
  :set-спутник-список (list луна))
(send марс :set-спутник-список
      (list фобос деймос))
;; Запуск Солнечной системы
(loop do
  ;; Пусть всегда светит солнце!
  (send космос :draw-circle центр-x
     центр-y (- диаметр-солнца 3)
     tv:alu-xor)
  (send космос :draw-circle центр-x
     центр-y (- диаметр-солнца 1)
     tv:alu-xor)
  (send космос :draw-circle центр-x
     центр-y диаметр-солнца tv:alu-xor)
  ;; Пусть вращаются планеты!
  (send меркурий :вращайся)
  (send венера :вращайся)
  (send земля :вращайся)
  (send марс :вращайся)))
```

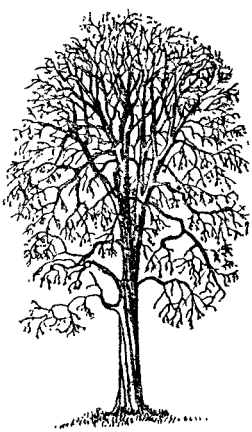
Запуск Солнечной системы

В конце функции СОЛНЕЧНАЯ-СИСТЕМА запустим Солнечную систему, послав объектам необходимые сообщения (:ВРАЩАЙСЯ).

Литература

1. Weinreb D., Moon D. *Lisp Machine Manual.* Fourth Edition, Symbolics Inc., Massachusetts, 1981.

2. Tapola P. Oliokeskeinen ohjelmointi – ZetaLispin maku-käsite. – В сб.: E. Hyvönen, J. Seppänen, M. Syrjänen (eds.) *STeP-84 Symposium Papers*. Publications of Finnish Society of Information Processing Science, No. 3, Espoo, 1984.
3. Symbolics: Solar-system-esimerkkiohjelma. ZetaLisp-kurssin oppimateriaalia. Symbolics Inc., Massachusetts, 1984.



5 РАЗВИТИЕ ЯЗЫКА ЛИСП И ЛИСП-СИСТЕМ

Любой язык программирования – это не только формальная система обозначений. С языком связаны также история его возникновения, развития, распространения и использования, а также его влияние на общее развитие науки и техники; у каждого языка программирования есть также социальная и историческая стороны, являющиеся частью истории науки и техники. Именно история и показывает истинные роль и значение конкретного языка программирования или открытия. В этой главе рассмотрим историю развития Лиспа и Лисп-систем от зарождения языка до появления Лисп-машин.

Джон Маккарти, "отец Лиспа".

В предыдущих главах было показано, что ядро Лиспа сравнительно легко реализовать и его можно расширять при помощи самого Лиспа. Начиная с 60-х годов появилось много реализаций языка для различных машин. Разными путями развивались различные версии или диалекты, каждый из которых расширял лисповское мышление, черты и формы языка.

Со временем вокруг разных диалектов Лиспа и их реализаций возникли и развились различные школы и направления исследований, а также программное обеспечение, т.е. формировалась Лисп-культура. В этой главе мы даним представление о развитии различных Лисп-систем и соответствующих "культур" и об их наиболее важных научных и географических признаках.

История учит тому, что из истории мы ничему не учимся.
Б. Шоу

5.1 ИСТОРИЯ ЛИСПА

- **Отец Лиспа – Джон Маккарти**
- **Обработка списков и искусственный интеллект**
- **Значение Лиспа**
- **Ранние реализации Лиспа**
- **Литература**

В настоящее время Лисп – старейший язык после Фортрана из широко используемых языков программирования, но было время, когда Лисп не занимал такого значимого положения в программировании, какое он занимает сейчас. До 80-х годов Лисп в основном использовался теоретиками программирования как формальный язык и был средством лабораторных исследований искусственного интеллекта. Настоящий расцвет языка, по-видимому произойдет в ближайшем будущем, когда ожидается появление большого числа систем и машин с искусственным интеллектом.

Отец Лиспа – Джон Маккарти

Лисп не является результатом коллективной деятельности комитетов или поиска компромиссов, он прежде

всего отражает личное видение сущности и возможностей программирования одного человека, Джона Маккарти. Возраст языка и продолжающийся рост его применения вне рамок чисто научных лабораторий показывают, что в идеях Маккарти есть нечто значимое и непреходящее.

Джон Маккарти родился в 1927 г., он изучал математику в Калифорнийском техническом университете (Caltech) и окончил его в 1948 г. Там он работал некоторое время помощником Джона фон Неймана, а затем продолжил учебу в Принстонском университете. Здесь он познакомился с Марвином Минским, также учившимся в Принстоне. В 1952 году Клод Шеннон – пионер теории информации – пригласил Маккарти на работу в лаборатории Bell System.

Клод Шеннон.

Четыре года спустя Маккарти присвоили звание доцента Дартмутского колледжа, откуда он в 1958 г. перешел в Массачусетский технологический институт (MIT) на должность профессора по связи. Здесь он вместе с Марвином Минским начал работы по проекту искусственного интеллекта и взялся создать такой язык программирования, который бы хорошо подходил для программирования задач искусственного интеллекта. Свою пионерскую работу по разработке Лиспа Маккарти осуществлял в MIT с 1958 по 1963 г., после чего он перешел в Стенфордский университет в Калифорнии, где стал работать профессором по искусственному интеллекту.

Обработка списков и искусственный интеллект

Интерес Маккарти к алгебраической обработке списков и к исследованию искусственного интеллекта побудил его принять участие в 1956 г. в летней школе по искусственному интеллекту в Дартмуте (США), которая считается первой организованной встречей по исследованиям в этой области. На этой школе А.Ньюэлл, Дж.Шоу и Г.Саймон представили разработанный ими язык IPL (Newell et al. 1968). IPL являлся языком обработки списков для вычислительной машины Johnniac фирмы Rand Corporation. Он был предназначен для программы искусственного интеллекта Логик-теоретик вывода теорем логики. Позднее Герберт Саймон получил Нобелевскую премию (1978 г.) за свой

Алан Ньюэлл.

вклад в применение вспомогательных средств принятия решений в области экономики и организации деятельности.

IPL не получил широкого распространения из-за своей чрезмерной близости к машинному языку. Однако благодаря ему родилась идея хранения программ и данных в памяти в виде структур, связанных ссылками, независимыми от физического расположения структур, в виде списков. Для списков не нужно было заранее резервировать в памяти некоторую область, их можно было создавать, изменять и удалять динамически. Первоначальная идея списков заимствована из психологии и ранних исследований по ассоциативной памяти.

Дж. К. "Cliff" Шоу.

Маккарти в то время работал в MIT на вычислительной машине IBM 704, подаренной университету фирмой IBM. В это же время в IBM уже проектировался первый язык программирования высокого уровня – Фортран. Сначала у Маккарти была мысль реализовать обработку списков IPL в Фортране. Однако этой идее не суждено было сбыться. Маккарти участвовал также в работе Комитета по языку Алгол 60, но и там не нашел понимания и одобрения своих мыслей. Так он постепенно пришел к своему особенному языку программирования.

Значение Лиспа

Маккарти осознавал значимость структур и форм представления в задачах искусственного интеллекта. Он уже давно говорил о специальной теории представления (representation theory). *Представление знаний* (knowledge representation) составляет центральную часть современных исследований по искусственному интеллекту.

Для символьных цепочек IPL Маккарти придумал основанную на скобках форму представления, иерархическую списочную запись, а также точечную нотацию соответствующую внутреннему представлению. Ис-

пользуя понятие пустого списка – NIL, ему удалось элегантно определить основные алгебраические действия по обработке списков. Так родились примитивные функции и предикаты: CAR, CDR, CONS, ATOM и EQ, необходимые для анализа, конструирования и сравнения списков. Из лямбда-исчисления Черча (Church 1941) для определения функций он заимствовал абстрактное и достаточно простое понятие лямбда-выражения. Этот подход позволил естественным образом описывать функции и передавать им параметры.

"Дерево жизни".

Может быть самой важной была идея об единообразном представлении программ и данных в виде списков, благодаря чему стала возможной как работа с программой и ее интерпретация в качестве данных, так и интерпретация данных и работа с ними как с программой. Единая форма представления программы и данных дала возможность определения семантики языка и программирования его интерпретатора в самом языке. Благодаря единообразному представлению программ и данных функции можно передавать в качестве параметров другим функциям (функционалы), также стало возможным преобразование, самоизменение и порождение новых программ во время их исполнения (сравните с обучением, адаптацией и т.п.).

Лисп входит в число наиболее значимых и долговечных языков программирования. Его влияние на исследования в области искусственного интеллекта и технологии знаний особенно проявилось в интерактивном программировании, методах функционального программирования, в распространении в программировании идей рекурсивного подхода, а также в том, что многие идеи в последствии были включены в более новые языки программирования. Развитие Лисп-систем повлияло на распространение систем разделения времени в 60-х и 70-х годах, а также на распространение в 80-х годах дисплеев, позволяющих работать с окнами (desktop metafora). Однако наиболее существенное

Китайский символ долголетия.

значение заключается все-таки в математической системе понятий и формализме, оказавшем большое влияние на развитие программирования как науки.

Ранние реализации Лиспа

Первая реализация интерпретатора Лиспа появилась осенью 1958 г., когда в MIT начались исследования по проекту искусственного интеллекта, предложенного Маккарти и Минским. Первый интерпретатор был реализован на компьютере IBM 704. Позднее на основе этой версии были разработаны более развитые версии Лиспа для вычислительной машины IBM 7090 и в дальнейшем для серий IBM 360 и 370.

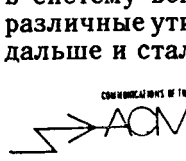
Марвин Минский.

Компьютер IBM 704 плохо подходил для работы в интерактивном режиме. Убедившись в этом, небольшая группа разработчиков, в том числе и из фирмы IBM, отделилась, чтобы основать свою собственную фирму для разработки и производства более совершенных машин. Так родилась вычислительная машина PDP-1 (Programmed Data Processor) и небольшая мастерская под названием Digital Equipment Corporation, которая с тех пор превратилась во второго в мире по величине производителя компьютеров.

Именно новая вычислительная машина и вычисления в интерпретирующем режиме и создали более благоприятные предпосылки для интерактивного программирования. Написание, тестирование и отладку программ можно было выполнять, не выходя из интерпретатора Лиспа. Этот принцип, известный теперь как "среда программирования", появился уже в системе Lisp 1, впервые представленной в 1960 г. С тех пор диалоговая и интерпретирующая среда программирования, а также пошаговое программирование, тестирование и отладка (incremental programming) признаны не только в Лисп-системах, но и во многих других интегрированных средах. Входящие

Герберт Саймон.

в систему вспомогательные программные средства и различные утилиты в течение десятилетий развивались дальше и стали более разнообразными и разумными.



В том же 1960 г. вышла ставшая классической статья Маккарти в журнале "Communications of ACM" (McCarthy 1960). В ней синтаксис и семантика языка Лисп определялись в виде алгебраического формализма при помощи самого Лиспа. Этот формализм с тех пор стал наиболее употребимым языком записи в литературе по теории программирования.

Вскоре в Lisp 1 были обнаружены недостатки, которые привели к разработке новой версии. Через два года была готова система Lisp 1.5, и ее описание вышло в издательстве MIT Press (McCarthy et al. 1962). Поскольку в описании системы Lisp 1.5 пояснялась также и ее реализация, то эта система стала основой, на которой базировались реализации для многих других машин. Эта книга долго пользовалась спросом, и поэтому регулярно выходили ее переиздания.

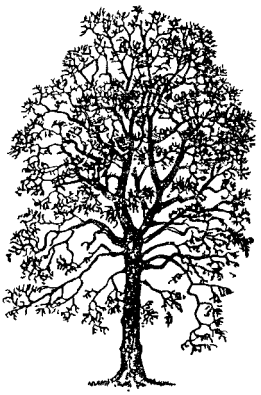
LISP 1.5

После опубликования в 1962 г. описания Lisp 1.5 начали появляться различные реализации Лиспа в США и язык стал распространятся в том числе и за рубежом. В следующем разделе мы рассмотрим развитие Лисп-культуры за пределами США и после этого остановимся на наиболее важных используемых в настоящее время реализациях.

Литература

1. Berkeley E.C., Bobrow D.G. *The Programming Language Lisp: Its Operation and Applications.* MIT Press, Cambridge, Massachusetts, 1964.
2. Church A. *The Calculi of Lambda Conversion.* Princeton Press, Princeton, California, 1941.
3. McCarthy J. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part 1, *CACM,* Vol. 3, April, 1960.

4. McCarthy J., Abrahams P., Edwards D. et al. *Lisp 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 1962.
5. McCarthy J. History of Lisp. *Proceedings of the ACM Conference on the History of Programming Languages,* Los Angeles, 1977.
6. McCarthy J. Lisp – Notes on its Past and Future. *Conference Record of the 1980 Lisp Conference,* Stanford University, 1980.
7. Newell A. et al. *Information Processing Language – Manual.* The Rand Corporation, 1968.
8. Sammet J. *Programming Languages: History and Fundamentals.* Englewood Cliffs, 1969.
9. Stoyan H. *Lisp – Anwendungsgebiete, Grundbegriffe, Geschichte.* Academie-Verlag, Berlin, 1980.
10. Stoyan H. Early Lisp History. *In 1984 ACM Symposium on Lisp and Functional Programming,* Austin, Texas, 1984.



Объяснение всей истории – география.

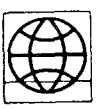
Р. Уоррен

5.2 ЛИСП РАСПРОСТРАНЯЕТСЯ ПО СВЕТУ

- **Развитие Лиспа в других странах**
- **Лисп в Западной Европе**
- **Лисп в Восточной Европе**
- **Лисп в далеких странах**
- **Лисп в Скандинавии**
- **Лисп в Финляндии**
- **Литература**

Развитие Лиспа в других странах

В этой главе дадим краткий обзор распространения Лиспа за пределами США. Лисп довольно рано начал широко использоваться, но нигде в мире, за исключением некоторых университетов и исследовательских учреждений, не достиг заметного положения среди языков программирования. Ситуация быстро изменилась в 80-е годы, но только время покажет, какое место займет Лисп в будущем. Обзор в основном опирается на подробное исследование, сделанное в работе Стояна (Stoyan 1980).



Лисп в Западной Европе

Алан Тьюринг.

Великобритания. За пределами США интерес к Лиспу прежде всего пробудился в Великобритании, которая считается второй после США страной в области исследований искусственного интеллекта. Сразу после статьи Маккарти (1960) язык начали внедрять в Royal Radar Establishment (Woodward и Jenkins 1961). В начале 60-х годов в Англии приобрела популярность обработка

списков (Foster 1968, Fox 1966), но, как правило, Лисп использовался физиками, а не исследователями искусственного интеллекта или программистами. Цель последних заключалась в дальнейшем развитии подобных Алголу языков и включении в них средств работы со списками. Такими языками были BCPL и CPL, являющиеся предшественниками языка Си, а также язык POP-2 (Burstall et al. 1971), из которого впоследствии развился POPLOG.

В 1964 году в Imperial College начали использовать IBM 7090, для которой получили из Стенфорда систему Lisp 1.5. Это дало толчок для реализации Лиспа на вычислительной машине Атлас, разработанной в Манчестерском университете (d'Inverno 1969). В то время Атлас был одной из самых производительных вычислительных машин в мире, а по архитектуре – прототипом машин третьего поколения. Из других ранних реализаций необходимо упомянуть Лисп-систему, реализованную на компьютере ICL 1905 в Университете Куинз и Интерлисп, реализованный на ICL 4 в Эдинбургском университете.

CAMBRIDGE LISP 68000

Из английских реализаций Лиспа для микро-ЭВМ нужно отметить BBC Lisp (Norman и Cattel 1983) и XLisp (Betz 1984), последняя из которых уже содержит объекты.

Франция. Во Франции Лисп появился в Институте Блеза Паскаля в Париже и Университете Гренобля (Gohen 1965). Особенно важным для дальнейшего распространения Лиспа было издание учебника по Лиспу (Ribbens 1969) на французском языке. В 70-х годах Лиспом начали заниматься Шестой и Восьмой университеты Парижа и исследовательский центр Inria.

Первыми реализациями Лиспа на французских компьютерах были системы, созданные для CAE-510 и Telemechanique 1600 (Greussay 1972 и 1975). На основе этого опыта возник французский диалект Лиспа –

VLisp (Greussay 1976, 1977) и впоследствии – LeLisp (Chailloux 1980). И VLisp, и LeLisp родственны Маклиспу, но на LeLisp, кроме этого, повлияли новые диалекты: Franz Lisp, NIL, Зеталисп и Коммон Лисп. Однако следует отметить, что LeLisp и Коммон Лисп совпадают лишь в части ядра языка (Kernel Common Lisp).

Le Lisp

В 80-х годах работы, связанные с Лиспом, велись во Франции довольно оживленно. Реализации языка LeLisp были осуществлены на многих различных компьютерах, в том числе на вычислительных машинах, созданных на базе микропроцессоров M68000 и Intel 8088/8086, на VAX-11 и IBM 30XX. Кроме того, разрабатывается множество других реализаций (Bel Mac 32, Sel 32, NS16000, DPS7, Mini 6, PR1ME и Norsk Data (Chailloux 1985)). Созданы также реализации языка LeLisp для персональных компьютеров Macintosh и Lisa (Hardebeck 1985), а также сокращенная версия LeLisp 80 – для 8-битовых микро-ЭВМ (Chailloux 1983). Быстрому распространению системы способствовала простота переносимости, обусловленная виртуальным характером ее определения (Chailloux 1984, Devin 1984). В исследовательском центре Inria на базе этого определения сейчас создается Лисп-машина.

По отношению к Лиспу, как и во многих других областях, французы шли в авангарде. Например, в интересах программистов с дефектами зрения для языка VLisp был разработан вывод по системе Брайля.

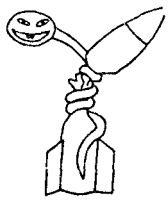
IRCAM

Из специальных областей применения, где может использоваться Лисп, следует отметить исследования в области компьютерной музыки, которые проводятся в IRCAM (Institut de Recherche et Coordination Acoustique/Musique) – центре музыкальных исследований Франции. Размещается это учреждение в Париже рядом с Центром Помпиду. Цель исследований заключается в применении методов символьной обработки символов и объектного программирования в програм-

мировании музыкальных форм и структур (Cointe и Rodet 1984).

Во Франции издается единственный посвященный Лиспу информационный бюллетень (Lisp Bulletin).

ФРГ. В ФРГ Лисп появился в начале 60-х годов также благодаря IBM 7090. Первыми учреждениями, где он начал использоваться, были GMD (Gesellshaft fuer Mathematik und Datenverarbeitung, Bonn) и университеты Бонна, Дармштадта и Майнца. Вначале так же, как и в Англии Лисп, преимущественно использовался физиками.



"Frieden Schaffen ohne Waffen", символ движения за мир и против размещения ракет Першинг в ФРГ.

Первые реализации Лиспа для немецких компьютеров TR4 фирмы Siemens были созданы в Марбургском (Melenk 1973) и Штутгартском университетах (Laubsch 1976). Однако более важным считается решение фирмы Siemens начать сопровождение системы Интерлисп. Интерлисповская версия языка была реализована в исследовательском центре немецкого языка (Institut fuer Deutsche Sprache) в Мангейме (Epp 1977, Guntermann 1978) совместно с Университетом города Уппсала.

В ФРГ созданы также Лисп-системы для микро-ЭВМ. Из них прежде всего стоит упомянуть чисто учебную систему TLC–Lisp (Görz 1983), разработанную в Университете города Эрланген-Нюрнберг.

В 1985 году Герберт Стоян основал в ФРГ музей Лиспа. Музей размещается в Эрлангенском университете и занимается сбором любых материалов, касающихся Лиспа. В данный момент в музее собрано уже около 1000 экземпляров различных изданий по Лиспу, которые также занесены в базу данных.

Основание музея Лиспа – факт, свидетельствующий не о том, что Лисп – это музейный

язык программирования, а означающий лишь то, что язык занял свое место в истории культуры программирования, будучи одним из самых первых языков программирования в мире.

Лисп в Восточной Европе

Советский Союз. В Советском Союзе Лисп не получил очень широкого распространения, хотя Маккарти посетил страну еще в 1968 г. Основную часть времени Маккарти провел в Новосибирске, где заложил основу реализации Лиспа на машине БЭСМ-6 в Вычислительном центре Сибирского отделения АН. Кроме того, при содействии Б.Беркли, входившего в эту группу, в Москве в ВЦ АН СССР началась реализация интерпретатора Лиспа для той же БЭСМ-6 (Лавров и Силагадзе 1969). БЭСМ-6 была мощной 48-битовой машиной, предназначенной для научно-технических расчетов, с быстродействием около 1 миллиона операций в секунду, созданной в 60-е годы.

ЯЗЫК ЛИСП

В начале 70-х годов разрабатывавшие интерпретатор Лиспа для БЭСМ-6 С.Лавров и Г.Силагадзе переехали из Москвы, С.Лавров перешел в Ленинградский университет, а Г.Силагадзе вернулся в Тбилиси в ВЦ Грузинской академии наук. Так появились два новых центра развития Лиспа и новые реализации языка на машинах серии ЕС. В Ленинграде Лисп был реализован также на польском компьютере Odra 1204. Были и другие ранние реализации: версия для БЭСМ-6 в Москве, совместимая с версией Лиспа для английского компьютера ICL 4, версия для серии машин ЕС в Московском энергетическом институте (МЭИ) и в Дальневосточном научном центре во Владивостоке. Кроме этого, в Советском Союзе на западных компьютерах используются некоторые зарубежные

ВЦ АН СССР

Лисп-системы, такие как Stanford Lisp и UT–Lisp в Дубне на IBM 370 и CDC 6600[1].

В 1975 г. в Тбилиси состоялась четвертая международная конференция по проблемам искусственного интеллекта – IJCAI–75. Она ознаменовала заметное оживление в СССР исследований в этой области и распространение Лиспа во многих университетах и научных центрах. Вскоре после этого появился первый учебник по Лиспу на русском языке (Лавров и Силагадзе 1978).

В 80-х годах в СССР были начаты работы по созданию компьютеров пятого поколения, основанных на Прологе и Лиспе.

ГДР. В ГДР исследования в области искусственного интеллекта и развитие Лиспа начались в 1969 г. в Дрезденском объединении Роботрон. Первая реализация на машинах серии ЕС заработала уже в следующем году, хотя описание вышло в свет в конце следующего десятилетия (Stoyan 1978).

Вторая реализация Лиспа на машинах серии ЕС была создана в Берлине в Институте кибернетики и информатики АН ГДР. На основе языка ZKI–Lisp создан интерфейс с базами данных на немецком языке (Koch et al. 1984).

Польша. В Польше в Варшавском университете создан интерпретатор Lisp 1.5, написанный на Алголе (Zelman 1969, Waligorski 1970), для компьютера Odra 1204 и затем Stanford Lisp 1.6 для Odra 1304.

Чехословакия. В Чехословакии работы начались с внедрения Lisp 1.5 на микро-ЭВМ завода Tesla в 1971 г. Система была готова через несколько лет (Kolar и Mueller

[1] В Институте проблем передачи информации в Москве в конце 70-х годов была разработана весьма развитая система ЭКЛИСП, предназначенная для использования на мини-ЭВМ ECLIPS. Большую популярность в СССР получила также систем Нордстрема (Швеция) Лисп-на-Фортране. – *Прим. ред.*

1974). Сейчас в Чехословакии Центр исследований искусственного интеллекта находится в Братиславе – это Институт технической кибернетики (Ustav technickej kybernetiky), который одновременно является координационным центром исследований искусственного интеллекта в странах СЭВ.

Венгрия. В Венгрии в 1973 г. в Институте автоматики и вычислительной техники Венгерской АН (SZTAKI) создали небольшой интерпретатор Лиспа (около 40 функций) для ЭВМ Videoton R10. Некоторое время спустя в Институте информатики (TKI) для той же самой ЭВМ была разработана система большего размера, состоявшая примерно из 200 функций (Potari 1975). В 1977 году в Институте вычислительной техники (SZAMKI) началась реализация Интерлисп-системы для ЭВМ H80 фирмы Honeywell.

Румыния. В Румынской академии наук была разработана система DMLisp для мини-ЭВМ Coral и I100 (Sotirescu и Stefanescu 1981), а также в Техническом университете Бухареста (Stefan et al. 1984) была создана архитектура машины, ориентированной на Лисп.

Лисп в далеких странах

Япония. В Японии Лисп стал применяться сравнительно поздно, но его развитие было бурным. При этом особенно заметна попытка создания наиболее эффективных реализаций Лиспа. В Японии разработано несколько Лисп-машин.

Первые реализации Лиспа были выполнены в конце 60-х годов для машин Hitac–5020 и Tosbac–3400 фирм Хитати и Тосиба (Nakanishi 1968). В 70-х годах появилось много новых систем, наиболее известная из которых HLisp фирмы Хитати (Goto 1974). К числу других ранних реализаций относятся: Lisp/P Токийского университета для Facom 230 фирмы Фудзицу, Olisp

Университета в городе Осака для машины ACOS System 800 фирмы NEC, а также Lisp 1.5 университета Окаяма для машины Melcom Cosmo 700 фирмы Мицубиси.

В 1974 году в Японии прошел первый симпозиум по символьной обработке. Одновременно проводились соревнования между Лисп-системами (Lisp Contest (Takeuchi 1978)), позднее ставшие своего рода традицией. Под влиянием этого интерес к Лиспу продолжал быстро расти. В 1978 г. в соревнованиях участвовало уже 20 систем, три из которых были зарубежными. Распространению языка способствовали также появившиеся учебники по Лиспу на японском языке (Goto 1974, Nakanishi 1977).

京都市左京区北白川追分町
京都大学数理解析研究所

В Японии разрабатывались и Лисп-машины. В конце 70-х годов существовало 6 проектов (Kurokawa 1979). Эти проекты были различны по своему подходу и отличались от соответствующих американских проектов. Самый известный из них – машина Alpha фирмы Фудзицу, единственная доступная на рынке (с 1985 г.). В числе остальных проектов – проект NK3, разработанный в Университете в Киото, ALPS/I, предложенный Университетом Аояма Гакуин, а также FLATS и PULCE (Kurokawa 1979).

Kyoto Common Lisp

В 80-х годах Япония объявила о начале работ над нашумевшим проектом ЭВМ пятого поколения (FGCS), который в большей степени опирается на язык Пролог, чем на Лисп. В 1984 г. был изготовлен первый прототип Пролог-машины. Однако под влиянием этого проекта появляется также вычислительная техника, ориентированная на Лисп. На это указывают непрекращающиеся попытки включить в Пролог многие черты Лиспа и объектного программирования.

ICOT
新世代コンピュータ
New Generation Computer

Китай. Первый интерпретатор Лиспа в KHP Lisp-130 для китайской мини-ЭВМ был создан в 1980 г. в Шеньянском институте автоматики АН КНР (Xinsong

1983). Это реализация системы Lisp 1.5, которая содержала 94 функции, запрограммированные на машинном языке. В 1982 г. в Пекинском университете на языке WISP реализована переносимая система Lisp–3000.

Мексика. В Мексику Лисп попал еще в начале 60-х годов вместе с переехавшими туда американскими исследователями (Guzman & McIntosh 1966). В Политехническом институте Лисп был реализован на компьютерах IBM 709 и Q–32, а позднее в Университете Мехико-Сити на компьютере Burroughs 6700 (Magidin и Segovia 1974).

Канада. В Канаде в Университете Ватерлоо реализован Lisp/360 (Bolce 1968). Новые системы позже не разрабатывались, так как их можно было легко получить из США.

Лисп в Скандинавии

В Скандинавии первыми на Лисп обратили внимание в Норвегии, хотя существенного развития он там не получил.

В Дании в конце 60-х годов был основан Североевропейский университетский вычислительный центр – NEUCC. Центр разместился в Техническом университете Лингби и получил в свое распоряжение большой компьютер IBM 7090 и вместе с ним Лисп (Gardner 1968). Однако вплоть до 80-х годов Лисп в Дании не был широко распространен, оставаясь всего лишь увлечением.

Наибольший интерес к Лиспу проявился в Швеции, в Уппсала его стали использовать уже в конце 60-х годов. Оттуда Лисп попал в Линчепинг и в Умео. В Стокгольмском университете и в Королевском техническом университете он заинтересовал физиков и матема-

NIGSAM

тиков с точки зрения обработки алгебраических выражений.

Первая Лисп-система заработала в Университете города Уппсала на машине CDC 3600. В систему внесли столько различных изменений (Mäkilä 1969, Nordström 1968), что пришлось написать свое руководство по Лиспу (Urmi 1970).

Из других созданных в Швеции Лисп-систем стоит упомянуть написанную на Фортране систему F1 Lisp – позднее F2 и F3 – (Nordström et al. 1970), которая благодаря своей хорошей переносимости получила сравнительно широкое распространение даже за пределами Швеции.

В конце 60-х годов Сандевал и Урми посетили США и познакомились в фирме BBN с разрабатываемым тогда там Интерлиспом (тогда еще BBN–Lisp). Вернувшись из США, в 1970 г. Сандевал основал в Уппсала лабораторию и начал работу над проектом, целью которого была реализация подобной Интерлиспу системы PILS (Paged Interactive Lisp System) на мини-ЭВМ Siemens 305, которую предполагалось использовать в лаборатории. Однако мини-ЭВМ они не получили, так как вычислительный центр университета решил приобрести большую машину – IBM/370. Поэтому было решено осуществить реализацию системы на ней, сделав ее по возможности близкой Интерлиспу. Этот проект удался, и система получила название Interlisp 360/370 (Urmi 1975).

Шведская реализация Интерлиспа была продана за 40 тысяч шведских крон в ФРГ, Швейцарию, Японию. Эту версию перенесли и на совместимые с машинами IBM компьютеры: в Германии – на компьютеры Siemens 4004, а в Японии – на компьютеры серии M фирмы Фудзицу. Позднее обе компании занялись также сопровождением системы.

В 1975 году группа разделилась, и Сандевал стал руководителем лаборатории Datalogi в открывшемся в Линчепинге Техническом университете. После этого

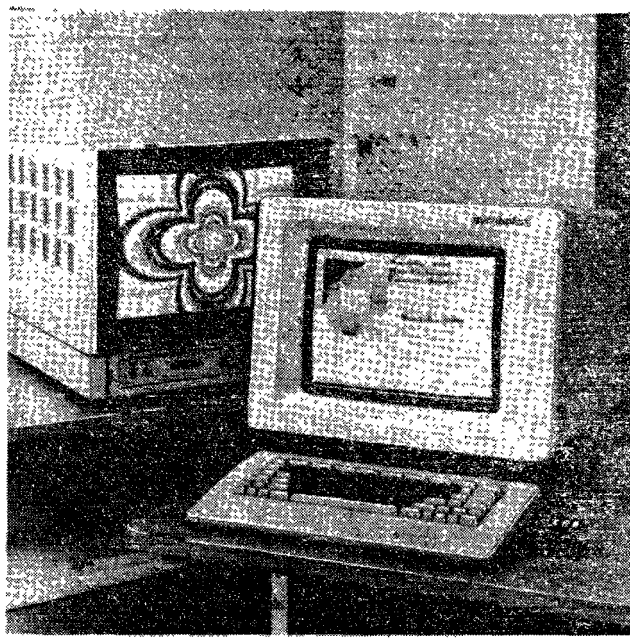
Первая Лисп-машина появилась в Финляндии в 1984 г. на факультете языкознания Хельсинкского университета.

работы по Лиспу и искусственному интеллекту развивались столь бурно, что лаборатория в Линчепинге получила мировую известность в этой области. В ее распоряжении в 70-е годы были вычислительные машины DEC–10 и DEC–20, в 80-е годы – Лисп-машины фирмы Xerox.

Разработки продолжались также и в Уппсала, хотя там больше внимания уделяется не Лиспу, а Прологу. Университет в Уппсала первым в Европе приобрел сразу три Лисп-машины в 1982 году. В Уппсала реализована Пролог-система LMI–Prolog для Лисп-машин фирмы LMI (Kahn 1983).

Лисп в Финляндии

В Финляндии Лисп привлек внимание в середине 60-х годов. В Университете города Тампере Лисп изучают с 1968 г. как часть углубленного курса программирования. Студентами осуществлена реализация ин-



В бюджете 1986 г. было предусмотрено 2 миллиона марок для покупки Лисп-машин для университетов и вузов страны. На фотографии изображена Лисп-машина серии Symbolics 3600.

терпретатора Лиспа на ЭВМ H1644 (Ruohola 1973). С 1971 года Лисп включен в программу семинаров по углубленному курсу программирования также и в Хельсинкском университете. В 1972 г. на ЭВМ B6700 реализован интерпретатор Лиспа на Алголе (Bärlund 1972).

UNIVAC

LISP

UNIVAC 1108

В вычислительном центре Технического университета (TKK) в 1969 г. был организован семинар по использованию Лиспа (Seppänen 1969). Первая Лисп-система появилась в Финляндии вместе с компьютером Univac 1108 (Norman

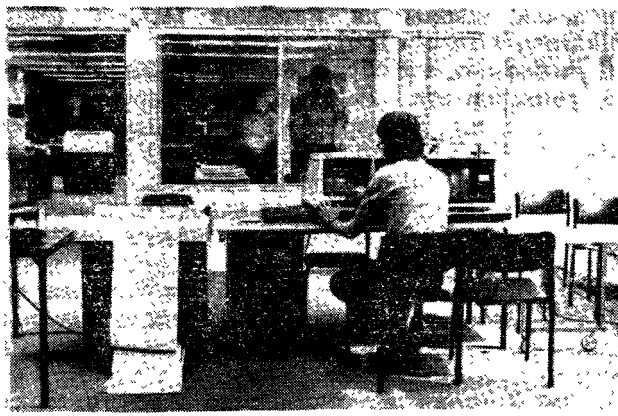
1968). В 1971 и 1972 годах в Отделении по обработке информации организованы курсы по Лиспу и символьной обработке. Материалы лисповских курсов и семинарских занятий были изданы отдельной книгой (Seppänen 1972), ставшей учебным пособием университетов Тампере и Хельсинки. Появление книги по Лиспу пробудило большой интерес к нему, но отсутствие хорошей Лисп-системы стало препятствием на пути развития Лиспа в Финляндии.

Джон Маккарти позирует для финской делегации на конференции IJCAI-83 в 1983 г., году вторжения искусственного интеллекта в Европу.

Особенно отрицательно на развитии Лиспа и исследованиях по искусственному интеллекту в Финляндии сказалась задержка на десять лет с поступлением в Технический университет компьютера PDP-10. Стремление получить именно PDP-10 объясняется его большой популярностью в MIT и других университетах США и наличием для него хороших программ на Лиспе и в области искусственного интеллекта. Однако начатые еще в 1968 г. хлопоты так и не увенчались успехом, и поэтому временно пришлось сосредоточиться на UNIVAC 1108. Лишь в 1978 г. Технический университет получил компьютер DEC-20. Только тогда в Финляндии появилась возможность развернуть серьезные работы в области символьной обработки, искусственного интеллекта и финского языка.

DECSYSTEM-20

В 1984 году в Отаниеми организованы первые Дни исследований искусственного интеллекта в Финляндии STeP-84 (Hyvönen et al. 1984a, 1984b). На этой конфе-



В Техническом университете в 1978 г. началась эксплуатация вычислительной машины DECSYSTEM-20 с системами Интерлисп (1979 г.) и Маклисп (1982 г.), а также разработанной на месте Plisp (1981 г.).

ренции по Лиспу и Прологу работали отдельные секции. В 1984 году в Финляндии появились Лисп-машины, и они тоже были представлены на этой конференции. В том же 1984 г. фирма Xerox поставила первую Лисп-машину в Центр изучения языков Хельсинкского университета. Фирма Nokia заключила договор о представлении в Северной Европе Лисп-машин фирмы Symbolics. В 1985 году появилась еще и машина Explorer фирмы Texas Instruments.

STeP-84

Наконец перечислим созданные в Финляндии Лисп-системы.

В 1980 году в вычислительном центре Технического университета начата работа над проектом PLisp (Portable Lisp), с целью запрограммировать на Паскале Portable Standard Lisp (Marti et al. 1979) Ютаской

лисповской школы на машинах DEC-20 (Malmi, Pietikäinen и Tapola 1981).

В учебном отделе обработки информации Технического университета реализована базирующаяся на системе Хендерсона (1980) LispKit система HutLisp (Heino 1982) для ЭВМ VAX под управлением операционной системы UNIX. В результате студенческой работы ядро интерпретатора LispKit было запрограммировано на Паскале также и в лаборатории цифровой техники Технического университета (Hyvönen и Karvinen 1981).

В лаборатории телефонии Технического университета запрограммирован интерпретатор Лиспа для микро-ЭВМ (Noko 1983).

В лаборатории акустики того же университета реализован интерпретатор Лиспа на базе микропроцессора M6809 с учетом требований обработки речевой информации (Karjalainen 1984).

ENTITY SYSTEMS

Для микро-ЭВМ Mikko 2 в Финляндии реализована система Mikko Lisp, основывающаяся на Коммон Лиспе (Erling, Lassila и Pirinen 1984b, Erling и Lassila 1986). В систему Mikko Lisp входит система объектно-ориентированного программирования Flavors, а также она предоставляет на микро-ЭВМ многочисленные возможности, присущие Лисп-машинам. Mikko Lisp – это коммерческий продукт, и он доступен для микро-ЭВМ Mikko 2 и 3 фирмы Nokia, а также для IBM PC/XT/AT под названием ES-Lisp. Mikko Lisp – результат совместной работы акционерного общества Entity Systems и Nokia Informaatiojärjestelmät (Информационные системы фирмы Nokia). Вторая финская Лисп-система, являющаяся коммерческим продуктом – это реализованная на Commodor 64 (J.Sinkkonen) система Comlisp (Siilasmaa 1985).

Литература

Великобритания

1. Betz D. Xlisp: An Experimental Object Oriented Language. Manchester, 1984.
2. Betz D. Xlisp Tutorial. *Byte,* Vol. 10, No. 3, March, 1985.
3. Burstall R., Collins J., Poppelstone R. *Programming in POP–2.* Edinburgh University Press, Edinburgh, 1971.
4. Foster J. *List Processing.* American Elsevier, London, 1968.
5. Fox L. *Advances in Programming and Nonnumerical Computing.* Oxford, 1966.
6. d'Inverno R. ALAM – Atlas Lisp Algebraic Manipulator. *Computer Journal,* Vol. 12, No. 2, 1969, pp. 124–127.
7. Norman A., Cattel G. *Lisp on the BBC Microcomputer.* Acornsoft, Cambridge, England, 1983.
8. Woodward P., Jenkins R. Atoms and Lists. *Computer Journal,* Vol. 4, No. 1, 1961, pp. 47–53.

Франция

1. Chailloux J. *Le modele Vlisp: description, evaluation, et interpretation.* These de 3eme cycle, Universite de Paris 6, 1080.
2. Chailloux J. *Vlisp 10.3 Manuel de Reference.* RT–16–78, Universite Paris 8, Vincennes, 1978.
3. Chailloux J. *LeLisp 80 version 12, le manuel de reference.* No. 27, Inria, 1983.
4. Chailloux J. La machine virtuelle LLM3. Inria, 1984.
5. Chailloux J. LeLisp, a Portable and Efficient Lisp System. In Boyer S. (ed.) *ACM Symposium on Lisp and Functional Programming.* Austin, Texas, 1984.

6. Chailloux J. *LeLisp de l'Inria, Le Manuel de reference.* Inria, 1985.
7. Cohen J., Nguyen H.D. Definition de Procedures Lisp en Algol, Examples et Utilisation. *Rev. Francaise Traitement Information,* Vol. 8, No. 4, 1965.
8. Cointe P., Rodet X. Formes: an Object and Time Oriented System for Music Composition and Synthesis. In Boyer S. (ed.) *ACM Symposium on Lisp and Functional Programming.* Austin, Texas, 1984.
9. Devin M. Le portage du systeme LeLisp: mode d'emploi. Inria, 1984.
10. Greussay P. *Manuel Lisp 510:* Description et Utilisation. Institut d'Intelligence Artificielle, Universite Paris 8, Vincennes, 1972.
11. Greussay P. *Lisp T 1600: Manuel de Reference.* Dept d'Informatique, Universite Paris 8, Vincennes, 1975.
12. Greussay P. Iterative Interpretation of Tail Recursive Lisp Procedures. Dept. d'Informatique, Universite Paris 8, Vincennes, 1976.
13. Greussay P. Contribution a la definition interpretative et a l'implementation des lambda-langages. Universite de Paris 6, 1977.
14. Hardebeck E. LeLisp Macintosh. *ACT Informatique,* Paris, 1984.
15. Kirmijian, Roy J.-P. *Lire Lisp, le langage de l'Intelligence Artificielle.* Cedic-nathan, Paris, 1985.
16. Queinnec C. *Langage d'un autre type: Lisp.* Eyrolles, Paris, 1984.
17. Queinnec C. *Lisp – Mode d'emploi.* Eyrolles, Paris, 1984.
18. Ribbens D. *Programmation non numerique Lisp 1.5.* Dunod, Paris, 1969.
19. Wertz H. *Lisp, une introduction a la programmation.* Masson, Paris, 1985.

ФРГ

1. Epp B. *Interlisp Programmierhandbuch.* Institut für Deutsche Sprache, Mannheim, 1977.
2. Görz G. *Lisp 1.5 Handbuch für die CDC 3300.* Universität Erlangen-Nuernberg, Nr. 11, 1972.
3. Görz G. Die Verwendung von Lisp an den Wissenschaftlichen Rechenzentren der BRD. Universität Erlangen-Nuernberg, Rechenzentrum, Nr. 63, 1976.
4. Görz G. A Programming Environment for TRC-Lisp. *Rundbrief der FGKI in der GI,* Nr. 30, 1983.
5. Guntermann R., Kolb D. *Siemens-Interlisp, User Manual.* Siemens AG, Muenchen, 1978.
6. Hamann C.-M. *Einführung in das Programmieren in Lisp.* de Gruyter, Berlin, 1982.
7. Laubsch J., Krause D., Hess K., Schatz W. *Maclisp Manual.* Memo Nr. 3, Institut für Informatik, Universität Stuttgart, 1976.
8. Melenk H., Roitzsch R. Lisp 1.5 für den TR440. GRZ fur die Wissenschaft, Berlin, 1980.
9. Stoyan H., Görz G. *Lisp – Eine Einführung in die Programmierung.* Springer-Verlag, Berlin, 1984.

СССР

1. Юфа В. О новых функциях в системе Лисп-БЭСМ-6. *Обработка символьной информации,* No. 4. ВЦ АН СССР, Москва, 1978.
2. Лавров С., Силагадзе Г. *Входной язык и интерпретатор системы программирования на базе языка Лисп для машины БЭСМ-6.* ВЦ АН СССР, Москва, 1969.
3. Лавров С., Силагадзе Г. *Автоматическая обработка данных – Язык лисп и его реализация.* Наука, Москва, 1978.

4. Силагадзе Г. Компилятор с языка Лисп для машины БЭСМ-6. *Вычислительная математика и программирование.* ВЦ АН ГССР, Тбилиси, 1975.

Другие страны СЭВ

1. Koch D., Busse J., Friedrich H., Geske U., Heicking W. German Language Access to Databases. AdW der DDR, Berlin, 1984.
2. Kolar J., Mueller K. *Programmovaci jazyk Tesla–Lisp 1.5.* Prag, 1974.
3. Potari F. *Lisp 1.5/R10 Programming System.* TKI, Budapest, 1975.
4. von Rozsa P. *Rekursive Funktionen in der Komputer Theorie.* Akademiai Kiado, Budapest, 1976.
5. Sotirescu D., Stefanescu M. *DMLisp Manual of Reference.* IPB, Bucharest, 1981.
6. Stefan G. Dialisp – A Lisp Machine. In Boyer S. (ed.): *ACM Symposium on Lisp and Functional Programming.* Austin, Texas, 1984.
7. Stoyan H. *Lisp–Programmierhandbuch.* Akademie-Verlag, Berlin, 1978.
8. Waligorski S. Przetwarzanie napisov – wyniki prac prowadzonych na universytecie Warszawskim. *Naukowe problemy maszyn matematicznych,* Warszawa, 1970.
9. Zelman H. *Interpreter Lisp 1.5.* Uniwersytet Warszawskiego, 1969.

Япония

1. Goto E. *Introduction to Lisp.* Kyoritsu Shuppan, 1974.
2. Nakanishi M. *KLISP Reference* Manual. Keio Institute of Information Sciences, Yokohama, 1968.
3. Nakanishi M. *The Programming Language Lisp.* Kindai Kagakusha, 1977.
4. Kurokawa T. *Lisp 1.6 Users Manual.* EPICS–5–ON, 1973.

5. Kurokawa T. *Lisp 1.8 Users Manual.* Toshiba Research Center, Tokyo, 1974.
6. Kurokawa T. New Marking Algorithms for Garbage Collection. *2. USA–Japan Computer Conference,* 1975, pp. 580–584.
7. Kurokawa T. *Lisp 1.9 Users Manual.* EPICS-5-ON-2, Tokyo, 1976.
8. Kurokawa T. A New Fast and Safe Marking Algorithm. Tokyo, 1976.
9. Kurokawa T. Why is the Lisp 1.9 Interpreter so Fast? Tokyo, 1976.
10. Kurokawa T. Lisp Activities in Japan. IJCAI-1979 Proceedings, Tokyo, 1979.
11. Takeuchi I. (ed.) Report of the 2nd Lisp Contest. Information Processing Society of Japan, 1978.

Китай

1. Xinsong J. AI Research in China: A Review. *IJCAI–83 Proceedings,* Karlsruhe, 1983.

Канада

1. Bolce J., Cooper R. A Description of the University of Waterloo Lisp 1.5 Interpreter for the IBM/360. Computing Center, 1968.

Мексика

1. Guzman A., McIntosh H. CONVERT. *Comm. ACM,* Vol. 9, No. 8, 1966, pp. 604–615.
2. Magidin M., Segovia R. Implementation of Lisp 1.6 on the B6700 Computer. Communicaciones tecnicas de CIMAS, No. 70, Mexico City, 1974.

Скандинавские страны

1. Kahn K. Unique Features of Lisp Machine Prolog. UPMAIL Report 14, Uppsala University, 1983.
2. Mäkilä K. LAP for CDC 3600. Uppsala University, 1969.

3. Nordström M. RA Routine for Lisp 3600. Uppsala University, 1968.
4. Nordström M., Sandewall E., Brezlaw D. Lisp F1, a Fortran Implementation of Lisp 1.5. Datalogielaboratoriet, Uppsala University, 1970.
5. Nordström M. Lisp F2, Changes in the Lisp Code. Datalogielaboratoriet, Uppsala University, 1974.
6. Sandewall E. A Proposed Solution to the Funarg-Problem. Datalogielaboratoriet, Uppsala University, 1970.
7. Sandewall E. Lisp: Principles. University of Uppsala, Datalogielaboratoriet, 1976.
8. Sandewall E. Programming in the Interactive Environment – the Lisp Experience. *Computing Surveys,* Vol. 10, No. 1, 1978.
9. Urmi J. *3600 Lisp U3 Users Manual.* Uppsala University Data Center, 1975.
10. Urmi J. *Interlisp 360/370 User Reference Manual.* Uppsala University Data Center, 1975.

Финляндия

1. Bärlund O. Lisp B1 – tulkki Lisp 1.5 kielelle. Helsingin Yliopisto, Tietojenkäsittelyopin laitos, 1972.
2. Erling O., Lassila O., Pirinen P. The ALLL Environment – Using Object-Oriented Concepts in Lisp. Статья в (Hyvönen, Seppänen, Syrjänen 1984a).
3. Erling O., Lassila O., Pirinen P. *Mikko Lisp Manual.* Nokia NITEC, 1984.
4. Erling O., Lassila O. Mikko Lisp, Making Common Lisp Viable on Microcomputers. Статья в (Karjalainen, Seppänen, Tamminen 1986).
5. Heino A. The HutLisp Programming System. TKK, Tietojenkäsittelyopin laitos, raportti B41, 1982.
6. Hyvönen E., Karvinen J. An Implementation of the LispKit Lisp System. TKK, Digitaalitekniikan laboratorio, raportti C2, 1981.

7. Hyvönen E., Seppänen J. Lisp – Symbolienkäsittelyn ja tekoälyn ohjelmointikieli. Opetusmoniste, TKK ja Nokia NITEC, Helsinki, 1984.
8. Hyvönen E., Seppänen J., Syrjänen M. *STeP-84 Symposium Papers.* Tietojenkäsittelytieteen Seura Ry., julkaisu 3, Espoo, 1984 (a).
9. Hyvönen E., Seppänen J., Syrjänen M. *STeP-84 Tutorial and Industrial Papers.* Tietojenkäsittelytieteen Seura Ry., julkaisu 4, Espoo, 1984 (b).
10. Karjalainen M. Lisp-mikrotietokone ja tulkki MC6809-prosessoria käyttäen. Статья в (Hyvönen, Seppänen, Syrjänen 1984a).
11. Karjalainen M., Seppänen J., Tamminen M. *STeP-84 Symposium Papers: Methodology.* Tietojenkäsittelytieteen Seura Ry., Espoo, 1986.
12. Laaja A. *Funktionaalisten ohjelmointikielten toteutustavoista.* Laudaturtyö, Tampereen Yliopisto, 1982.
13. Malmi L., Pietikäinen P., Tapola P. PLISP Pascal-pohjainen standardi Lisp systeemi. TKK, Laskentakeskus, opas, 1981.
14. Noko T. NokoLisp ja Osborne. *Tietokone,* No. 2–3, 1983.
15. Ruohola T. *Yksinkertaisen prosessorin implementointi.* Tampereen Yliopisto, laudaturtyö, 1973.
16. Seppänen J. Univac 1108 Lisp. TKK, Laskentakeskus, moniste, 1969.
17. Seppänen J. *LISP kielenopas.* OtaData D6, Teknillinen korkeakoulu, Tietojenkäsittelyopin laitos, 1972.
18. Seppänen J. Lisp ohjelmointi ja kielenkäsittely. Luentomoniste, Helsingin Yliopisto, Yleisen kielitieteen laitos, 1981.
19. Siilasmaa R. Kotimainen Lisp-tulkki 64:lle. Bitti 2/85.

Mikko Lisp

Настоящее полно прошедшего, а будущее заключено в недрах настоящего.

Г. Лейбниц

5.3 ЛИСП-СИСТЕМЫ

- **Маклисп – основной диалект восточного побережья**
- **BBN-Lisp, Xerox и Интерлисп**
- **Standard Lisp и PSL**
- **Franz Lisp**
- **NIL – New Implementation of Lisp (новая реализация Лиспа)**
- **Диалект T Йельского университета**
- **Зеталисп Лисп-машин**
- **Вавилонская башня мира Лиспа**
- **Стандарт Коммон Лиспа**
- **Производители Лисп-машин приходят к договоренности**
- **Литература**

В предыдущих разделах дан обзор ранней истории Лиспа и его распространения по всему свету. Важнейшие из используемых и наиболее признанных в мире современных диалектов Лиспа происходят из США. В следующих главах мы дадим обзор разработки, свойств и реализаций важнейших современных диалектов Лиспа.

Маклисп – основной диалект восточного побережья

В начале 60-х годов в MIT был начат проект MAC (Multiple Access Computer или Machine Aided Cognition). Его цель – исследование использования вычислительных машин в диалоговом режиме, а также реализация подходящих операционных систем и языков

программирования. Маклисп (Moon 1974), заложивший в MIT лисповские традиции и основы программирования задач искусственного интеллекта, получил свое название от проекта MAC. В рамках проекта MAC возникла и лаборатория машинных знаний, а позднее лаборатория искусственного интеллекта.

Работа над проектом MAC, проводимая в вычислительном центре MIT и над операционными системами CTSS и Multics компьютеров IBM 7094 и GE 634 привела к созданию новых методов, которые позднее получили широкое распространение. Маккарти при разработке языка Lisp 1 изобрел способ, позволявший запускать интерпретатор Лиспа параллельно с пакетными работами (time stealing). Именно этот способ дал начало новой форме использования ресурсов ЭВМ, получивший известность под названием "системы разделения времени" (time sharing).

MacLisp

Вначале Маклисп был реализован на базе Lisp 1.5 на компьютере PDP-6 (1964), а затем на более новой PDP-10 (1968). На широко используемом в дальнейшем компьютере PDP-10 Маклисп работал как под управлением операционной системы ITS (Incompatible Timesharing System), разработанной в лаборатории искусственного интеллекта MIT, так и с операционной системой TOPS-10/20 фирмы Digital. Другая реализация Маклиспа была создана для машин серии GE-600 под управлением операционной системы Multics.

TOPS-20

Кроме символьной обработки Маклисп широко использовался в традиционных числовых вычислениях, применяемых, например, в обработке речи и изображений. Кроме исследователей искусственного интеллекта и разработчиков алгебраической системы Максима на Маклисп оказали влияние и работы групп в MIT по робототехнике, обработке речи и изображений. Исходя из требований, предъявляемых этими областями, в Маклисп были включены новые математические типы данных, такие как матричная и битовая обработка, а также

DECsystem-10

широкий набор арифметических функций и средств. Быть может, важнейшая среди них – возможность вычислений с неограниченной точностью, основывающаяся на созданных Д.Кнутом (1969) алгоритмах.

Маклисп был также первой Лисп-системой, для которой создан хороший транслятор. Транслятор генерирует машинную программу, получившую имя LAP (List Assembly Program), в форме списков. Машинный код в виде списка легко обрабатывать и результирующий код для числовых задач получался эффективней, чем у лучших фортрановских трансляторов для PDP–10 (Fateman 1973, Steele 1977). Это объясняется "разумностью" транслятора Маклиспа при обработке выражений и оптимизации кода.

Однако большую часть своих свойств Маклисп приобрел под влиянием стоящих перед исследователями искусственного интеллекта проблем и накопленного ими опыта. Так в язык попали макросы чтения (macro character) и таблицы чтения (read table), позволявшие легко изменять и расширять структуру языка. Таким же образом из требований к программам и окружению возникли управляющие структуры (такие, как catch и throw), механизмы обработки прерываний и ошибок, а также использование управляющих символов, создан и интегрирован в систему экранный редактор Emacs, появились управление и взаимодействие параллельных процессов.

Всего в Маклиспе используется около 400 функций. Вероятно, самым большим недостатком системы является то, что ее никогда не документировали должным образом. Документация по этой системе разбросана по разным отчетам и руководствам. Маклисп был исследовательской системой и не предназначался для обучения и промышленного использования.

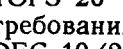
BBN-Lisp, Xerox и Интерлисп

В середине 60-х годов Лиспом заинтересовались вооруженные силы США, а также многие исследовательские центры, такие как SDC (System Development Corporation), BBN (Bolt, Beranek and Newman Inc.), SRI

(Stanford Research Institute). Фирма BBN, работавшая недалеко от MIT и являющаяся одной из ведущих в США исследовательских фирм в области искусственного интеллекта, в 1966 г. начала разрабатывать свою версию Лиспа для компьютеров SDS-930 (Scientific Data Systems) и PDP-10. Именно эта версия для PDP-10 способствовала тому, что компьютеры PDP-10/20 оставались важнейшими машинами для исследований искусственного интеллекта вплоть до 80-х годов.

BBN-Lisp был разработан на PDP-10 как подсистема созданной на самой фирме BBN операционной системы TENEX. TENEX опирается на аппаратный механизм страничной организации памяти, созданный специально для поддержки больших лисповских программ. Помимо страничной организации памяти в систему команд были добавлены команды работы со стеком и расширен набор регистров для ускорения переключения от одной задачи к другой (context switching). Используя эти решения, фирма DEC разработала операционные системы TOPS-10 и TOPS-20 для машин DEC-10/20. Таким образом, требования Лиспа привели к тому, что компьютеры DEC-10/20 стали наиболее подходящими машинами для работы с разделением времени.

В 1972 г., когда началось сотрудничество с фирмой Xerox, BBN-Lisp превратился в Интерлисп (Teitelman 1978), так как Xerox стала сотрудничать с BBN. Именно Xerox купила обанкротившуюся в то время компанию SDS, на машинах которой работал BBN-Lisp. Машины SDS не имели успеха, но несмотря на это, Интерлисп развивался дальше. К 1978 г., когда вышло описание Интерлиспа (Teitelman 1978), язык и система уже достаточно стабилизировались.



Интерлисп уже не был языком в том же смысле, что и Маклисп или другие Лисп-системы или обычные традиционные системы программирования. Он представлял собой интегрированную среду программирования, в которую вошло множество различных вспомога-

тельных средств, описанных нами в главах, посвященных программному окружению. Интерлисп стал классическим примером хорошо развитых программных сред (Teitelman и Masinter 1981) и средств в системах разделения времени.

В 1974 году Xerox начала разработку для Интерлиспа персональной лисповской рабочей станции под названием Alto. В реализации Лиспа для Alto впервые применили спроектированную специально для языка Лисп микропрограммируемую систему команд и мини-ЭВМ, способную с более высокой производительностью, чем универсальные ЭВМ, интерпретировать лисповские программы. Из этой машины Alto впоследствии развились Лисп-машины серии 1100 фирмы Xerox (Burton et al. 1981) Dolphin, Dorado, Dandelion, а также их сокращенный вариант – рабочие станции Star.

Interlisp-D

На основе версии Интерлиспа, работавшей в системе разделения времени, была создана совместимая снизу вверх версия Лиспа Interlisp-D, используемая на Лисп-машинах серии 1100. В ее пользовательский интерфейс входили многооконное взаимодействие, графика с высокой разрешающей способностью, средства выбора из меню и мышь, а также ориентированный на использование экрана инспектор структур данных. Идея разделения экрана на многие независимые окна (desktop metafora) родилась в Xerox Learning Group (XLG). Алан Кэй уже в конце 60-х годов предложил идею Dynabook подхода к компьютерам будущего и интерфейсу между человеком и машиной. Работа XLG привела в 70-х годах к разработке языка программирования Smalltalk (Смолтолк) (Goldberg и Robson 1983) и объектного программирования.

При создании Интерлиспа работа велась весьма тщательно. Система хорошо документирована и более новые версии совместимы с более ранними. Так преимуществом системы стало непрерывно пополняющееся большое количество переносимого программного обеспечения. С другой стороны, ограничение системы

старым зафиксированным уже в конце 70-х годов диалектом сделало систему отчасти устаревшей и трудно расширяемой.

В Интерлиспе среди прочего отсутствуют иерархические типы данных, объекты и замыкания. К тому же он базируется на динамическом связывании, тогда как новые версии Лиспа – статические. Однако из Интерлиспа берет начало новая версия – Коммон Лисп (1986). Для программирования на более высоком уровне в Интерлиспе разработаны такие средства, как LOOPS (Bobrow и Stefik 1983), в которых уже присутствовали объекты.

Интерлисп – столь замкнутая система, что доступна только ее оттранслированная версия в машинных кодах. В некоторых других системах, таких как, например, Зеталисп, поддерживается версия Лиспа на исходном языке, которая доступна пользователю и может им модифицироваться. Развитие закрытых систем, похожих на Интерлисп, связано с ресурсами, имеющимися у создавших их лабораторий.

Интерлисп использует свыше 500 функций и большое количество системных имен и флажков, с помощью которых можно настроить и подогнать систему. Интерлисп реализован в системе разделения времени на многих больших ЭВМ, таких как DEC-10/20, IBM 370, Siemens 4004, ICL-4, CDC 3300, Burroughs 6700, Fujitsu M200 и на других ЭВМ. Интерлисп доступен и для VAX-а (Bates et al. 1982).

TOPS-10/20

Standard Lisp и PSL

В связи с переездом Маккарти в Стенфорд в 1963 г. и Лисп перекочевал на западное побережье США. Из разработанных в Калифорнии систем прежде всего необходимо упомянуть стенфордский Lisp 1.6 (Quam и Diffie 1972), UCI Lisp (University of California, Irvine) и Stanford Lisp/360. Вместе с Э.Херном Лисп попал в Солт-Лейк-Сити в штат Юта. Коллеги Херна были заинтересованы в методах символьной математики для исследований в

THE UNIVERSITY OF UTAH

области теоретической физики, которые он запрограммировал на Лиспе. Таким образом возникла одна из наиболее популярных алгебраических систем, известная под названием Reduce.

Школа Юта вскоре занялась созданием своей версии Лиспа, которая должна была больше подходить для математических манипуляций и быть по возможности более машинно-независимой и переносимой. В 1966 году Херн опубликовал спецификацию Standard Lisp (Стандарт Лисп) в качестве предложения по стандартизации языка (Hearn 1966). Предложение, однако, не получило широкой поддержки, так как оно не понравилось исследователям искусственного интеллекта и они не одобрили его. Мешающим фактором в языке была, например, привязанность к типам.

REDUCE

Однако Ютовская школа реализовала на основе стандарта легко устанавливаемую систему – Portable Standard Lisp (PSL), которая используется на многих различных ЭВМ для поддержки системы Reduce (DEC-10/20, VAX/UNIX, HP9000, Apollo, Wicat, IBM и Cray). Сопровождение PSL и его дальнейшее развитие осуществляется и в исследовательском центре фирмы Hewlett Packard в Калифорнии.

PSL

PSL проектировался специально для переноса программ системы Reduce на новые устройства (так же как Franz Lisp и NIL для переноса Максимы, см. ниже). В стандарт PSL для улучшения переносимости входит сокращенный набор примитивных лисповских функций и структур.

Реализация PSL основана на машинно-независимом промежуточном языке SYSLisp, представляющем собой специальный близкий машине Лисп. С целью улучшения переносимости, сопровождения и расширяемости ядро PSL реализовано на языке SYSLisp, а средства более высокого уровня на самом PSL. На языке SYSLisp написан кросс-компилятор (на PDP-10), с помощью которого ядро PSL можно без труда перенести на новую аппаратуру. С точки зрения переносимости PSL напоминает T-Lisp (см. ниже). Ядро легко переноси-

$$(1-X^2)\ \frac{dY}{dX} + 1$$

мо, но не совместимо с другими лисповскими диалектами. Во всяком случае по своим решениям PSL представляет собой устаревающую Лисп-культуру.

Franz Lisp

Маклисп стал основой для многих новых диалектов Лиспа, первым из которых был Franz Lisp (Foderaro 1979). Эта версия Лиспа названа в честь известного венгерского композитора. Главным мотивом разработки Franz Lisp было желание получить современную Лисп-систему для новых машин VAX, чтобы на них можно было использовать систему Максима и другое лисповское программное обеспечение. Franz Lisp в довольно большой степени напоминает Маклисп, на котором первоначально была реализована Максима. Однако в диалекте отсутствуют некоторые устаревшие особенности Маклиспа и содержатся более новые системные идеи, заимствованные из разрабатываемых в то время в MIT Лисп-машин для Зеталиспа.

Новый диалект был реализован в Университете в Беркли на ЭВМ VAX 780/11 на языке Си под управлением системы UNIX (Fateman 1981). Franz Lisp довольно широко используется как под управлением UNIX, так и под управлением VAX/VMS и в настоящее время является наиболее часто используемой версией Лиспа для систем разделения времени. Кроме того, он широко используется и на 32-битовых микро-ЭВМ и рабочих станциях, работающих под управлением UNIX.

Благодаря своей хорошей переносимости Franz Lisp получил распространение во многих университетах и исследовательских учреждениях. Сопровождение системы также разошлось в различных исправлениях системных ошибок, реализациях наиболее эффективных алгоритмов, а также в расширениях языка. Важнейшими разработчиками диалекта были кроме Университета Беркли Университет Пенсильвании, лаборатории Bell System, лаборатории Ливермора и Университет Карнеги–Меллона.

NIL – New Implementation of Lisp (новая реализация языка)

Еще одна новая реализация Маклиспа на машине VAX получила название NIL (New Implementation of Lisp). Проект NIL был начат в MIT в 1979 г. (White 1979). Его цель состояла в определении похожей на Лисп-машину виртуальной VAX-машины и разработке для нее современной и эффективной системы программирования. Однако это удалось не в полной мере, поскольку микропрограммируемость на машинах VAX носит ограниченный характер. Однако NIL содержит большое ядро на машинном языке, в котором в некоторой степени удалось преодолеть ограничения, налагаемые архитектурой VAX. С другой стороны, NIL можно рассматривать как ответную реакцию MIT на Franz Lisp. Одним из мотивов осуществления проекта NIL был перенос Максимы на VAX.

В 1981 году исследовательская группа, разрабатывавшая NIL, распалась, но развитие системы продолжалось, хотя ожидания в этом отношении несколько уменьшились. Однако начиная с 1982 г. благодаря деятельности группы в MIT по робототехнике в лаборатории искусственного интеллекта регулярно выходили корректно обновляющиеся версии системы.

NIL испытал большое влияние со стороны Зеталиспа Лисп-машин. От них NIL позаимствовал объектное программирование в виде системы Flavors. NIL часто используется в организациях, где есть как устройства VAX, так и Лисп-машины с Зеталиспом, поскольку разница между ними почти незаметна, хотя NIL образует лишь небольшое подмножество Зеталиспа.

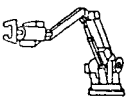
Диалект T Йельского университета

Проект NIL распался, поскольку часть участников считала свои идеи о том, как должна выглядеть хорошая Лисп-система, лучше чужих. И когда стало ясно, что компромисс не возможен, эта группа отделилась от проекта NIL и взялась разрабатывать свою

версию, которой они дали слегка ироничное название T (True Lisp).

Версия T разрабатывалась в основном в Йельском университете (Rees et al. 1982, 1984). В отличие от предыдущих Лисп-систем он основывается на статической области действия переменных, хотя программист может при желании использовать и динамические связи. Кроме того, функции в T – это "полноправные граждане", т.е. функции могут без ограничений возвращаться в качестве значений функций.

В отличие от NIL система T содержит довольно маленькое (около 600 строк) ядро, написанное не на языке Лисп. Идея состояла в расширении этого маленького ядра все новыми и новыми функциями и построениии определения системы с помощью техники раскрутки (bootstrap). Используя хороший оптимизирующий транслятор, разработчики языка надеялись создать более производительную систему, чем путем реализации большей части системы вручную на машинном языке. Благодаря такому подходу система получилась легко переносимой. Техника раскрутки заметно облегчает как разработку самой системы, так и ее сопровождение.



Кроме использования новой стратегии разработки в T была сделана попытка обновить и сделать более единообразными имена функций Лиспа, например имена всех предикатов оканчивались на вопросительный знак. Преимуществом было более легкое изучение языка, недостатком же стала несовместимость T с другими диалектами языка. Однако в системе доступен пакет макросов, с помощью которых диалект T становится совместимым со стандартом Коммон Лиспа.

T–Лисп как элегантный и чистый язык использовался в обучении и в системе разделения времени, и на рабочих станциях (Abelson и Sussman 1983). В настоящее время T доступен в качестве коммерческого продукта не только для компьютеров VAX, но и для многих 32-битовых рабочих станций.

T испытал большое влияние со стороны системы Scheme, разработанной в 70-х годах в MIT. Scheme –

это чистый Лисп, возникший из статического варианта Лиспа, разработанного в рамках проекта Лисп-машины (Steele и Sussman 1976, Steele 1976). Это элегантно определенное ядро Лиспа, которое достаточно прозрачно и эффективно реализуется.

Scheme оказал значительное влияние и на другие более новые разработки. Во время его создания исследовались статические замыкания, отложенные вычисления, применимость в Лиспе механизмов передачи сообщений (message passing) объектного программирования и оптимизация рекурсии. Многие его черты использованы не только в T, но и в Зеталиспе Лисп-машин и в Коммон Лиспе.

Scheme

Зеталисп Лисп-машин

При знакомстве с программным окружением мы уже встречались с Зеталиспом (Weinreb 1981), или Лиспом Лисп-машин (Lisp Machine Lisp). Зеталисп также опирается на Маклисп. Он создан в 70-е годы в MIT в рамках проекта Лисп-машины, финансированного оборонным агенством DARPA. С самого начала его целью было изготовление коммерческого продукта. В 1979 году в связи с проектом родились два предприятия, изготавливающие Лисп-машины: Symbolics Inc. и Lisp Machine Inc. (LMI). После этого в 80-е годы работа по развитию Зеталиспа продолжалась в них независимо друг от друга на коммерческой основе. В какой-то мере системы отличаются друг от друга, но в части Зеталиспа машины почти совместимы. В дальнейшем мы еще вернемся к Лисп-машинам.

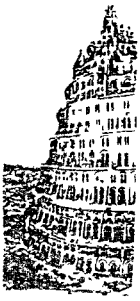
Лисп-машин

Zetalisp

Вавилонская башня мира Лиспа

После возникновения основной версии Lisp 1.5 (McCarthy 1960, 1963) на его основе было разработано бесчисленное множество диалектов и реализаций. Особенно много новых диалектов возникло в конце

70-х, начале 80-х годов. Это нисколько не удивительно, так как с самого начала Лисп проектировался как открытый язык, предпосылки роста и развития которого встроены в сам язык. Каждый программист в состоянии расширить язык по своему усмотрение.



Так в отношении Лиспа появилось множество разных точек зрения, представляющих различные области применения школ и исследовательских традиций, каждая из которых расширяла язык и подгоняла программное обеспечение в соответствии со своими потребностями. Сложившуюся ситуацию можно сравнить с библейской Вавилонской башней. Так случилось, что различные Лисп-культуры разошлись настолько, что они не совместимы друг с другом.

Такое разнообразие не доставляло особых неприятностей, пока язык использовался лишь в исследованиях. Даже напротив, идеи Лиспа таким образом могли постоянно развиваться, и поэтому в язык вошли новые, оказавшиеся полезными свойства, в том числе из других языков. Таким образом в настоящее время в Лиспе можно найти почти все свойства и методы, известные в программировании. Однако недостатки, возникающие из-за несовместимости как между диалектами, так и по отношению к другим языкам становятся все более существенными по мере роста числа практических применений Лиспа.

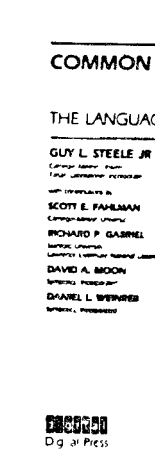
Ситуация особенно обострилась в начале 80-х годов. Впервые за все время существования языка на него появился коммерческий спрос и шансы на рыночный успех. Стало ясно, что переход из лабораторий в практическую деятельность потребует сосредоточения сил и стандартизации языка. Хотя различные точки зрения, с одной стороны, способствуют развитию языка, с другой стороны, рассредоточение сил в разработке Лисп-систем приводит к созданию небольших проектов, в которых часто наибольшее

внимание уделяется второстепенным вопросам, а кроме того происходит повторение одинаковых разработок.

Предложения стандартизовать язык, как отмечалось ранее, делались, уже начиная с 1960 г. (Hearn 1966, Marti et al. 1979), но они не дали результата. Может быть и к лучшему, поскольку для множества последующих разработок несомненно было полезно, что язык не заковали в стандарты слишком рано. Однако теперь подоспело время для формирования подходящего стандарта.

Стандарт Коммон Лиспа

Стандарт пытались подготовить в Университете Карнеги–Меллона как продолжение работы по проекту Spice Lisp, целью которого было определение и реализация развитой лисповской программной среды для персональной рабочей станции[1]. Начальный проект по стандартизации назвали Коммон Лисп, с целью определить единый базовый язык, на основе которого можно было бы реализовать совместимые системы и одновременно допустить в каждой реализации разработку своих расширений и своей среды.



В определении языка участвовало около 80 специалистов из разных университетов, исследовательских центров и фирм. Проектирование языка и формирование стандарта происходили в основном через сеть Агра, через которую было послано свыше 3000 сообщений. Наибольшее влияние на эту работу оказали диалекты Маклиспа с восточного побережья, но также и Интерлисп. Многие новые черты заимствованы из Зеталисп Лисп-машин. Таким образом, остановились на спецификации (Steele

[1] У нас иногда используют название АРМ, т.е. автоматизированное рабочее место. *–Прим. ред.*

1982a, 1982b, 1984), которая по объему в 10 раз больше предыдущего практического стандарта – Lisp 1.5.

Однако Коммон Лисп (1985) не является официальным стандартом, он только лишь представляет рекомендуемую спецификацию языка, которая де-факто стала стандартом. Коммон Лисп определяет сотни встроенных функций, форм и системных имен. Они отражают свойства, известные по различным Лисп-системам и которые, как показалось, можно зафиксировать на настоящем этапе. Одной из целей было также обеспечение совместимости с ранними версиями, однако при этом избежав трудностей старых систем.

Эта спецификация оставлена открытой: принципиальным является то, что осталась возможность в будущем, когда подойдет время и будет достигнуто согласие, добавить новые свойства и методы. Эта идея как раз соответствует духу Лиспа.

На настоящем этапе Коммон Лисп содержит важнейшие черты современных Лисп-систем: разнообразные типы данных, возможности определения типов, императивные управляющие структуры, макросы, с помощью которых легко определяются новые синтаксические формы, функционалы, замыкания, пространства имен, последовательности, ориентированные на использование потоков ввод и вывод, а также синтаксический интерпретатор и транслятор.

Коммон Лисп отнюдь не является готовой программной системой в том же смысле, что и Интерлисп, поскольку вопросы среды в основном оставлены открытыми. В стандарте не определено, каким должен быть редактор или другие вспомогательные средства. Сказано лишь в самом общем виде, каким образом они вызываются. Для того чтобы обеспечить быстрое развитие, среда и инструментальные средства еще не затронуты стандартизацией, и поэтому есть возможность создавать различные среды для различных целей. Коммон Лисп не определяет также и интерфейс пользователя.

В Коммон Лисп на современном этапе не включены даже средства объектного программирования, хотя и определены необходимые для этого базовые механизмы

(замыкание и прочее). Таким образом, объекты можно реализовать на Лиспе. Однако уже ведется подготовка к стандартизации средств и форм объектного программирования.

Похоже, что несмотря на критику, Коммон Лисп становится первым широко принятым стандартом Лиспа. На это повлияло и то обстоятельство, что Министерство обороны США (DoD) параллельно с языком Ада приняло его в качестве своего стандарта. Поэтому все большее число изготовителей компьютеров берут этот стандарт за основу своих реализаций (VAX Lisp, Golden Common Lisp и др.) и разработок программных систем в области искусственного интеллекта.

Критики Коммон Лиспа есть как в Европе, так и в США (Stoyan 1982, Brooks и Gabriel 1984). В Коммон Лиспе много внимания уделено практическим требованиям, и, вероятно, поэтому не все его черты эстетичны и чисты. Несомненно, что и другие Лисп-системы будут использоваться в дальнейшем, и их также необходимо развивать.

Коммон Лисп предназначен не только для работы со списками или для символьной обработки, он является универсальным языком программирования, включающим в себя особенно хорошие средства для численных вычислений, управления данными и связи. На Коммон Лиспе с одинаковым успехом можно писать программы в традиционных операторном, процедурном, фразовом стиле, а также и в свойственных Лиспу стилях. В языке содержатся предпосылки для использования различных способов и стилей программирования, таких как операторное, функциональное, макропрограммирование, программирование, управляемое данными, и продукционное программирование, а также средства, необходимые для логического и объектного программирования и реализации других средств более высокого уровня.

Можно смело сказать, что Коммон Лисп содержит почти все, что на сегодняшний день могут дать другие известные языки программирования, и, кроме того, он

предусматривает средства для расширения языка. Это бросает серьезный вызов языку Ада, от которого ожидалось, что он станет языком программирования будущего, но который показал свою несостоятельность в применении к задачам по символьной обработке списков и по исследованиям в области искусственного интеллекта, ставшими главными задачами для компьютеров нового поколения.

Конкуренция на лисповском рынке обостряется. Лисп-машины фирмы Texas Instruments и Explorer фирмы Sperry (теперь Unisys) вышли на рынок в 1985 и 1986 гг.

Производители Лисп-машин приходят к договоренности

Положение Коммон Лиспа как будущего стандарта подкрепляется решением производителей Лисп-машин приспособиться к нему. Для компьютеров, в основу которых заложен Зеталисп, это не означает больших изменений, так как они основываются на использовании Маклиспа и уже по своему происхождению близки Коммон Лиспу. В этом отношении большую уступку сделала фирма Хегох, решив сделать Интерлисп похожим на Коммон Лисп. Средства Интерлиспа и интегрированную среду можно перенести в Коммон Лисп с небольшими изменениями, но функции и ядро языка, например способ вычислений, требуют больших изменений.

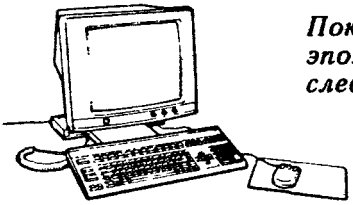
Кубок.

Литература

1. Abelson H., Sussman G. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts, 1983.
2. Bates R., Dyer D., Koomen J. Implementation of Interlisp on the VAX. *Conference Record of the ACM Symposium on Lisp and Functional Programming.* Pittsburgh, Pennsylvania, Aug. 15–18, 1982.
3. Bobrow G., Stefik M. *The LOOPS Manual.* Xerox PARC, Palo Alto, 1983.
4. Brooks R.A., Gabriel R. A Critique of Common Lisp. In (Boyer 1984).
5. Fateman R.J. Reply to an Editorial. *SIGSAM Bulletin,* March, 1973.
6. Fateman R.J. Is a Lisp Machine Different from a Fortran Machine? *SIGSAM Bulletin,* No. 12, 1978.
7. Fateman R.J. Views on Transportability of Lisp and Lisp-Based Systems. *ACM Conference, SYMSAC'81,* Snowbird Utah, 1981.
8. Feigenbaum E., McCorduck P. *The Fifth Generation. Artificial Intelligence and Japan's Computer Challenge to the World.* Addison-Wesley, London, 1983.
9. Foderaro J. *The Franz Lisp Manual – A Document in Four Movements.* University of California, Berkeley, California, 1979.
10. Goldberg A., Robson D. *Smalltalk-80. The Language and its Implementation.* Addison-Wesley, London, 1983.
11. Hearn A. Standard Lisp. *ACM SIGPLAN Notices,* Vol 4, No. 9, 1966.
12. Henderson P. *Functional Programming, Application and Implementation.* Prentice-Hall, London, 1980. [Имеется перевод: Хендерсон П. Функциональное программирование. Применение и реализация. –М.: Мир, 1983.]
13. Knuth D. *The Art of Computer Programming. Vol. 2, Seminumerical Algorithms.* Addison-Wesley, London, 1969. [Имеется перевод: Кнут Д.

Искусство программирования. Т.2.– М.: Мир, 1977.]
14. Levitan S., Bonar J. (1981) Three Microcomputer Lisps. *Byte,* September. 1981.
15. Marti J., Griss M., Griss C. Standard Lisp Report. *ACM SIGPLAN Notices,* Vol. 14, No. 10, 1979, pp. 48–68.
16. McDonald D., Clippinger J. *Artificial Intelligence Computers and Software: Technology and Market Trends.* Brattle Research Corp., Cambridge, Massachusetts, 1984.
17. Moon D. *MacLisp Reference Manual.* MIT, Laboratory for Computer Science, Cambridge, Massachusetts, 1974, 1978.
18. Moore J.S. *The Interlisp Virtual Machine Specification.* Xerox PARC, Palo Alto, 1976.
19. Norman E. *Lisp Reference Manual for the Univac 1108.* University of Maryland, Maryland, 1968.
20. Quam L., Diffie W. *Stanford Lisp 1.6 Manual.* AI Operating Note 28.7, Stanford University, California, 1972.
21. Pitman K.M. *The Revised MacLisp Manual.* AI Laboratory, MIT LCS, 1983.
22. Rees J., Adams N. T: A Dialect of Lisp or, LAMBDA: the Ultimate Software Tool. *Conference Record of the 1982 Symposium on Lisp and Functional Programming,* Pittsburgh, Pennsylvania, Aug. 15–18, 1984.
23. Rees J., Adams N., Meehan J. *The T Manual.* Computer Science Department, Yale University, New Haven, USA, 1984.
24. Stoyan H. Common Lisp – a Survey and a Critique. *Rundbrief der FG KI in der GI,* Nr. 28, 1982.
25. Steele G. Lambda, the Ultimate Declarative. AI Memo 379, AI Laboratory, MIT, Cambridge, Massachusetts, 1976.
26. Steele G., Sussman J. Lambda, the Ultimate Imperative. AI Memo 353, MIT AI Lab, 1976.

27. Steele G. Fast Arithmetic in MacLisp. *Proceedings of the Macsyma 1977 User's Conference,* 1977.
28. Steele G. An Overview of Common Lisp. *Conference Record of the ACM Symposium on Lisp and Functional Programming,* Pittsburgh, Pennsylvania, Aug. 15–18, 1982 (a).
29. Steele G. *Common Lisp Reference Manual.* SPICE Project Doc. s061, Dept. of Computer Science, Carnegie-Mellon University, 1982 (b).
30. Steele G. *Common Lisp.* The language. Digital Press, Hanover, Massachusetts, 1984.
31. Sussman G., Steele G. SCHEME: An Interpreter for Extended Lambda Calculus. Tech. Rep. 349, AI Laboratory, MIT, 1975.
32. Teitelman W. *Interlisp Reference Manual.* Xerox PARC, Palo Alto, California, 1978, 1983.
33. Teitelman W., Masinter L. The Interlisp Programming Environment. *IEEE Computer,* April, 1981.
34. White J.L. Program is Data – A Historical Perspective on MacLisp. *Macsyma User's Conference, Berkeley,* California, 1977.
35. White J. NIL – A Perspective. *Macsyma User's Conference, Washington D.C,* 1979.



Покрывало алтаря в одной эпохе – это коврик для ног в следующей.

М. Твен

5.4 ЛИСП-МАШИНЫ

- **Бегство из систем разделения времени**
- **Первые изготовители**
- **Успехи Лисп-машин**
- **Лисп или Пролог?**
- **Литература**

Бегство из систем разделения времени

С наступлением 70-х годов большие системы искусственного интеллекта и алгебраические системы натолкнулись на ограничения памяти и эффективности, существующие и на больших универсальных ЭВМ.

Восемнадцатибитовое поле адреса широко используемых машин PDP-10/20 стало серьезным ограничением, к тому же исследователи искусственного интеллекта не могли работать в системе разделения времени в дневное время из-за большой нагрузки на машины. Из этих проблем родилась идея об отдельной Лисп-машине и о маневре, который известен под названием "бегство из разделения времени" (escape from timesharing). На это направление повлияло также и быстрое развитие микроэлектроники в 70-х годах, сделавшее возможным проектирование и производство ориентированных на язык процессоров и персональных ЭВМ.

Первый отчет, связанный с Лисп-машинами, появился в серии изданий лаборатории искусственного интеллекта MIT в 1974 г. (Greenblatt 1974), а интегральная схема LSI "Lisp on a Chip" была изготовлена в 1978 г. (Steele, Sussman 1980). Первые промышленные Лисп-машины появились на рынке несколько лет спустя.

Крупные фирмы поддерживают развитие Лисп-культуры делая существенные подарки университетам и вузам. Технический университет приобщился к Лисп-машинам, получив в подарок Лисп-машину Explorer.

Часть идей, касающихся Лисп-машин, зародилась в Исследовательском центре Palo Alto фирмы Xerox и была результатом пионерских разработок в области обработки данных на персональных ЭВМ и экранно-ориентированных человеко-машинных интерфейсов. Это были уже ранее упоминавшиеся объектно-ориентированный подход, а также специальные интегрированные в среду средства и методы программирования, созданные фирмами Xerox и BBN в ходе работы над системой Интерлисп.

symbolics™

Первые изготовители

LMI

Производство Лисп-машин началось в США в середине 80-х годов на трех предприятиях: Lisp Machine Inc. (LMI), Symbolics Inc. и Xerox. Первые две фирмы приходят из MIT, и производимые ими машины используют почти совместимые диалекты Зеталиспа. Машины же фирмы Xerox представляют направление

Интерлиспа. Начиная с середины этого десятилетия Лисп-машины начала производить первая крупная фирма-изготовитель аппаратуры Texas Instruments (Explorer). В Японии тоже создано несколько Лисп-машин (Kurokawa 1979). Первый коммерческий прототип Лисп-машины (Alpha фирмы Фудзицу) был объявлен в 1984 г.

Успехи Лисп-машин

Целью проектирования Лисп-машин была разработка их в виде персональных ЭВМ, которые можно было бы использовать не только для профессиональных исследований в области искусственного интеллекта, но и для различных промышленных и коммерческих приложений. Разработке и их распространению помешала необходимость переноса программного обеспечения большого объема из дорогой среды больших машин.

По производительности оборудования Лисп-машины очень эффективны, кроме того, они имеют большой объем основной памяти. Их аппаратура спроектирована специально для вычислений на Лиспе. С точки зрения эффективности одной из наиболее важных особенностей является проверка типов на уровне аппаратуры (tagged architecture), используемая в системах, происходящих из MIT.

Однако наиболее существенным преимуществом такой аппаратуры является возможность использования на Лисп-машине интегрированной программной среды. В ней наряду с самим Лиспом содержатся разнообразные программные средства, которые мы уже рассматривали в связи с Зеталиспом и Интерлиспом. Программное обеспечение использует тысячи функций. Во многих системах помимо Лиспа доступны и другие языки (Паскаль, Ада, Фортран, Си, Пролог и др.). Так, в систему можно добавить реализованные ранее на других языках части или сделать традиционное программирование более эффективным с помощью разнообразнейших средств, имеющихся на Лисп-машине.

Лисп или Пролог?

В апреле 1982 г. японское Министерство внешней торговли и промышленности MITI приступило к реализации 10-летней программы исследований и создания ЭВМ, цель которой состоит в разработке компьютерной технологии и методов программирования, необходимых для интеллектуальных компьютеров 90-х годов (Moto-Oka 1982, Feigenbaum и McCorduck 1983, Simons 1983). Компьютеры пятого поколения, подобно Лисп-машинам, поддерживают работу по разработке программ искусственного интеллекта и его приложений как в части аппаратуры, так и программного обеспечения. В качестве языка программирования японцы используют не Лисп, а расширенные логические языки на основе Пролога.

Как Лисп-культуре, так и Пролог-культуре несомненно есть что позаимствовать друг у друга. Предсказывают, что в Лиспе появятся средства логического и продукционного программирования подобно тому, как раньше были заимствованы такие новые методы программирования и формализмы, как объектно-ориентированное программирование из языков Симула и Смолток.

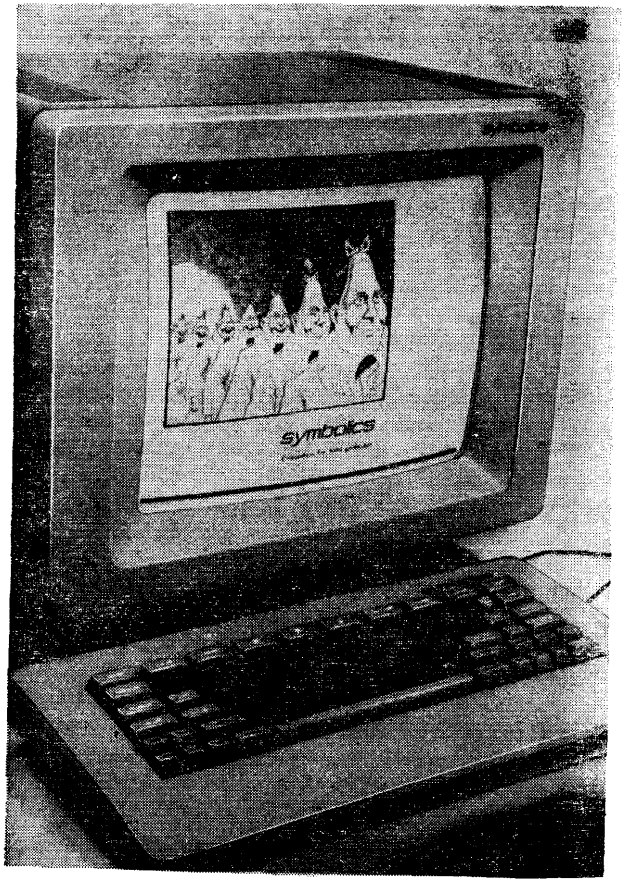
Соответственно и в логическом программировании замечено, что в некоторых случаях удобнее применять методы функционального и процедурного программирования из Лиспа, чем чисто декларативную логику. Поэтому в разрабатываемые в настоящее время расширения Пролога войдут многие лисповские средства, например развитые типы данных, процедурные структуры управления и объекты.

ICOT
新世代コンピュータ
New Generation Computer

Литература

1. Bawden A., Greenblatt R., Holoway J. et al. The Lisp Machine Lisp. В кн.: *Artificial Intelligence:*

An MIT Perspective, Vol. 2, MIT Press, Cambridge, Massachusetts, 1979.
2. Burton R., Kaplan R., Masinter L. et al. Papers on Interlisp-D. Xerox Palo Alto Research Center, Palo Alto, California, 1981.
3. Feigenbaum E.A., McCorduck P. *The Fifth Generation – Artificial Intelligence and Japan's Computer Challenge to the World.* Addison-Wesley, 1983.
4. Greenblatt R. The Lisp Machine. Working Paper 79, AI Laboratory, MIT, Cambridge, Massachusetts, 1974.
5. ICOT: Outline of Research and Development Plans for Fifth Generation Computer Systems. Institute for New Generation Computers (ICOT), Tokio, 1982.
6. Moto-Oka T. (ed.) Fifth Generation Computer Systems: *Proceedings of the International Conference of Fifth Generation Computer Systems.* Tokyo, Oct. 9–22, 1981, North-Holland, 1982.
7. Simons G. *Towards Fifth-Generation Computers.* NCC Publications, Manchester, England, 1983.
8. Steele G., Sussman G. Design of a Lisp-based Microprocessor. *CACM*, Vol. 23, No. 11, 1980.
9. Sussman G., Holoway J., Steele G. and Bell A. SCHEME-79 – Lisp on a Chip. *IEEE Computer*, July, 1981.
10. Weinreb P., Moon D. *Lisp Machine Manual.* AI Laboratory, MIT, Cambridge, Massachusetts, 1978.



Приключения талисмана разработчиков Зеталиспа, героя нью-йоркского темного мира "Zippy the Pinhead" в среде Зеталиспа.

ПРИЛОЖЕНИЕ 1

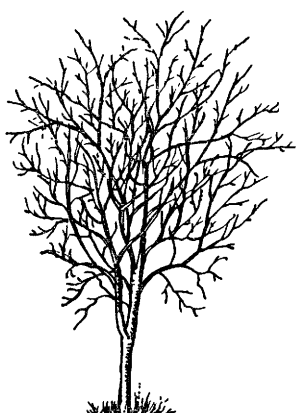
УКАЗАТЕЛЬ ФУНКЦИЙ И СИМВОЛОВ

Указатель содержит упорядоченные в алфавитном порядке имена функций, символов и других системных имен Коммон Лиспа содержащихся во втором томе. Приведены также и имена определяемых в примерах функций переменных и других лисповских объектов и обозначений. Сводка всего Коммон Лиспа приведена в приложении 1 к первому тому.

ПРИЛОЖЕНИЕ 2

УКАЗАТЕЛЬ ИМЕН И СОКРАЩЕНИЙ

В этом приложении собраны встречающиеся в тексте имена и сокращения. Одновременно оно может служить указателем авторов по перечням литературы. С его помощью можно найти литературу, которая вследствие разбиения по темам приведена в разных разделах. Символы и зарезервированные слова Коммон Лиспа образуют отдельный указатель (приложение 1).



ПРИЛОЖЕНИЕ 3

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Указатель составлен из встречающихся в тексте понятий и терминов, а также из специальных терминов языка Лисп. Имена и сокращения собраны в свой указатель (приложение 2) также, как символы Коммон Лиспа и имена, определяемые в примерах программирования (приложение 1).

ОГЛАВЛЕНИЕ

СОДЕРЖАНИЕ 1-ГО ТОМА