Российская академия наук
Сибирское отделение
Институт систем информатики
им. А. П. Ершова

Л.В. Городняя

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 3
Основные парадигмы программирования
Языки высокого уровня

Препринт
174

Новосибирск 2015

Препринт является третьей частью серии «Парадигмы программирования», посвященной исследованию парадигм программирования. Представлены результаты анализа особенностей языков высокого уровня. В качестве иллюстрации использованы фрагменты ряда языков, поддерживающих основные парадигмы программирования. Содержание препринта представляет интерес для системных программистов, студентов и аспирантов, специализирующихся в области системного и теоретического программирования, и для всех тех, кто интересуется проблемами современной информатики, программирования и информационных технологий.

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

L.V. Gorodnyaya

PROGRAMING PARADIGMS

Part 3
Basic Paradigms of Programming

Preprint
174

Novosibirsk 2015

The work describes research and specification of basic paradigms of programming. The author analyzes and compares special features of programming languages of high level. A functional model to comparative description of implementation semantics of basic paradigms is proposed. The author proposes a scheme of describing and defining paradigm features of a programming language. The approach is illustrated with fragments of programming languages of different levels, which belong to machine-oriented, system, imperative, object-oriented, and productive programming.

ВВЕДЕНИЕ
Языки высокого уровня

Благодаря языкам высокого уровня (ЯВУ) программирование стало массовой профессией. Программирование на ЯВУ приспособлено к представлению расширяемой иерархии понятий, отражающей природу понимания человеком решаемых задач и организации процессов их решения. Переход к ЯВУ дал возможность систематически укрупнять конструкции при подготовке текстов программ. Для этого понадобились сложные структуры данных, стереотипы техники программирования, локализуемые области видимости имен объектов и процедур их обработки, подчиненные структурно-логической модели управления, допускающей сходимость пошагового процесса отладки программ [1, 11, 28, 34–36, 39, 46]. Результативны графические интерфейсы и компонентные технологии, поддерживающие перенос отлаженных результатов в разные системы. В центре внимания – интеграция с библиотеками процедур, эффективная компиляция программ, контроль типов данных, соответствие стандартам области применения программ и технологиям быстрой разработки удобно сопровождаемых программ. Ряд проблем решается включением в ЯВУ низкоуровневых средств. Практически исчезает необходимость в блок-схемах, а методика самодокументирования и реализации справочных подсистем смягчает роль документирования. Программе на ЯВУ обычно соответствует семейство допустимых процессов, определение которого представлено формальной семантикой языка. Система программирования (СП), поддерживающая ЯВУ, как правило, порождает один из процессов этого семейства. Такое сужение диктуется не только реализационной прагматикой ЯВУ, но и необходимостью воспроизведения процессов при отладке программ [20–23, 33].

Текст программы на ЯВУ обычно обретает бипланарность – императивное представление процесса обработки данных в нем совмещено с декларативным описанием типов обрабатываемых данных, спецификаций, прагм и пр.. Возникает нечто вроде пространственной аппроксимации процесса вычислений, используемой при проверке корректности программ − статический или динамический контроль типов данных. Проработка понятий ЯВУ характеризуется пропорциями между чёткой аппликативностью и недетерминизмом, пространствами константных и переменных значений, элементарных и составных данных, открытыми и замкнутыми процедурами, средствами обработки строк и файлов, возможностями активного и

«ленивого» вычисления, использованием последовательных и параллельных схем управления вычислениями. Все ЯВУ используют стек при реализации укрупненных конструкций и защите локализуемых данных. Стилистика ЯВУ тесно связана с конструированием структур данных, кодированием алгоритмов, методами синтаксического анализа и компиляции программ с опорой на критерии теории программирования [2, 4–7].

Обычно высокий уровень языка обеспечивается программными средствами, но с появлением программаторов и микропрограммирования разница между программой и аппаратурой стала условной. Lisp, Pascal, Prolog, Smalltalk, Algol и другие ЯВУ были реализованы как входные языки на правах машинного кода [9, 37, 50, 58, 66, 67]. Ассемблер Эльбрус – яркий пример отечественной реализации ЯВУ аппаратными средствами [29].

При анализе парадигм ЯВУ необходимо учитывать следующие их особенности:

- практикуются неявные формы представления отдельных понятий ради лаконизма записи программ;
- выражения чаще всего рассчитаны на схему предвычисления над скалярами конкретной длины или сложными значениями (сначала вычисляются операнды, затем вычисляется результат операций).
- разнообразны виды ветвлений и циклов, категории функций и процедур;
- типы данных конструируются по фиксированным в ЯП правилам и реализуются по принятым в СП шаблонам;
- схемы управления вычислениями нередко фиксированы в языке и конкретно реализованы в системе программирования;
- взаимодействие и соответствие средств и методов, относящихся к разным семантическим системам, при их реализации в системе программирования определено по традиции и прецедентам, причем в коде оно скрыто или рассредоточено, а не структурировано;
- эффективность программирования базируется на знании методов реализации значений и обработчиков структур данных в памяти;
- результат программы, как правило, рассредоточен по разным переменным, но во многих ЯВУ есть выражения и функции, формирующие один результат.

Таблица 1

Названия некоторых ЯВУ,
относящихся к разным парадигмам программирования

ИП	ФП	ЛП	ООП	УЯ
Fortran	Lisp	Planner	Simula-67	Pascal
Algol	Рефал	Snobol	Smalltalk-80	Basic
Альфа	ML	Prolog	C++	Grow
C	Cmucl	Conniver	Eiffel	Logo
Modula	Erlang	QLISP	Clos	Oberon
Эльбрус	Interlisp	Mercury	Java	Робик
Occam	MuLisp	Oz	C#	Рапира
BLISS	Scheme		Scala	Oz
ЯРМО	Hope			
	Clean			
YACC	Dylan			
Lex	Miranda			
	Python			
	Haskell			
	Rubi			
	F#			

(ИП – императивное (стандартное/системное) программирование, ФП – функциональное/аппликативное программирование, ЛП – логическое/декларативное программирование, ООП – объектно-ориентированное программирование, УЯ – учебные языки программирования[1]).

В сравнении с языками низкого уровня семантика ЯВУ содержит арифметику, разделенную на ряд систем обработки целых и вещественных чисел в разных диапазонах, обычно зависящих от разрядности адресов и машинных слов. Имеются системы работы с указателями, символами и строками и система конструирования типов данных. Поддерживается набор стандартных операторов управления вычислениями – ветвления, циклы и вызовы процедур/функций. А главное – кроме работы с глобальными объектами, используются разные схемы локализации хранимых в памяти объектов, что обеспечено организацией структур данных по принципу иерархии.

Учебный концентр или инструментальное ядро ЯВУ можно ограничить одной или двумя арифметическими системами. Элементарные уровни

[1] УЯ не образуют самостоятельной парадигмы. Обладая своей эксплуатационной прагматикой, они используют операционную семантику изучаемых парадигм.

опорных ЯВУ разных парадигм можно описать как расширения или конкретизации такого концентра.

При анализе парадигм ЯВУ следует отметить вехи совершенствования системной программотехники.

- Первый ЯВУ Fortran ввёл в практику раздельную компиляцию, снизившую трудозатраты на отладку программ благодаря выделению техники сборки модулей, что обеспечило формирование общих библиотек, а заодно и устойчивость позиций языка Fortran до наших дней [21].
- Универсальный язык Lisp дал жизнь машинно-независимому стилю обработки данных, рассматриваемых как символьные списки или строки, что привело к парадигме функционального программирования [9, 24, 38, 49, 52, 58, 61–65].
- Популярность языка C связана с решением проблем машинно-зависимого переноса программ путем выделения компактного машинно-ориентированного ядра и самоописания Си-компилятора, что одновременно поддержало перенос на множество архитектур и стыковку с близкими языками, компилируемыми на тот же уровень сборки модулей [3, 8, 42, 59].
- Логическое программирование на языке Prolog показало возможности накопления и наследования частных рецептов [16, 26, 51], что позволило работать с недоопределёнными постановками задач, повышая степень их изученности.
- Появление C++ и ООП отвечает расширению сферы приложения информационных систем, при конструировании которых необходимо минимизировать объем повторного программирования [14, 27, 42, 67].

Теоретически различие ПП достаточно ясно выражается на уровне операционной семантики, представляющей детали структур памяти и механизмы выполнения укрупнённых действий. Но для выработки практических рекомендаций требуется более подробная формулировка различий на всех уровнях определения ЯП, включая реализационную прагматику, задающую границы вычислимости и эффективности программируемых решений, набор синтаксически различимых понятий, показывающий удобство отображения терминологии области приложения в программные конструкции, и эксплуатационную прагматику, определяющую трудоёмкость программирования и живучесть разрабатываемых программ.

Представление специфических деталей парадигм при сравнении ЯВУ можно выразить в форме нормализованной предикатной формы интерпретатора, различающего сквозные понятия сравниваемых языков на уровне абстрактного синтаксиса (АС) и абстрактной машины (АМ), дополненной описанием реализационной прагматики и онтологической спецификации опорных ЯП, поддерживающих парадигмы программирования.

При сравнительном описании парадигм СП, ФП, ЛП и ООП в качестве опорных языков используются C, Lisp 1.5, Clisp, Prolog, C++, Clos [2] на базе Венской методики определения ЯП [57] и подходов к динамической оптимизации программ [56].

9. ИМПЕРАТИВНОЕ (СТАНДАРТНОЕ) ПРОГРАММИРОВАНИЕ

Стандартное, императивно-процедурное программирование (ИП) рассматривает процесс обработки информации как конечную последовательность локальных изменений состояния памяти – императивно-процедурный стиль. Для ИП характерно четкое разделение понятий «программа» и «данные» с учётом статических методов контроля типов данных и оптимизации программ при компиляции. Общий механизм интерпретации стандартной программы естественно представить как автомат с отдельными таблицами имен для переменных, значения которых подвержены изменениям, и для меток и процедур, определения которых неизменны.

При трансляции программ обычно планируется распределение памяти для значений переменных в зависимости от их типов данных, выполняется размещение локальных данных в памяти, частичный контроль доступа к переменным и совместимости операций и операндов по типам данных, вычисление значений выражений (констант, переменных, элементов структур данных, результатов операций и вызовов функций), манипуляции по управлению вычислениями.

СП = (Текст → {Код | Адрес}): Пам [Переменная] → Пам

[2] CLOS – Объектно-ориентированная библиотека функций для Common Lisp.

Таблица 2

Конкретизация понятий в стандартных императивно-процедурных языках на примере языка С

Понятие	*Конкретизация*
Атом (Элементарное)	Имя |Скаляр|
Структура	Вектор, Строка, Структура = Запись, Объединение
Переменная	Ид → Адр (Знач)
Значение	Адрес, |Скаляр|
Выражение	Произвольные, со скобками и приоритетами
Действие/Операция	Арифметика, Доступ к элементам структур, Присваивания, Работа со строками. Формирование ТД
Условие/истина	0
Функция/ оператор	Организация схем управления Поствычисления в операторах
Аргумент	Локальные переменные на стеке
Вызов Фн/подпр	Организация новой области имен и исполнение определения попрограммы – блок
Определение Фн/подпр	Пополнение таблицы функций
Идентификатор	Адрес метки, константы, переменной, функции

Самый популярный язык программирования С содержит низкоуровневое подмножество, что провоцирует рассматривать его рассматривать как ЯНУ. Но приспособленность С к представлению обработки иерархии структур данных с помощью программируемых функций дает основания относить его к ЯВУ.

Язык Си предполагает, что программа собирается из набора файлов, содержащих фрагменты программы и библиотеки функций, подготовленные возможно независимо. При этом собственно компилируемая программа представляет собой одноуровневую конструкцию из равноправных определений структур данных и функций. Это означает, что любая функция имеет доступ к любому элементу любой структуры данных и может изме-

нять его значение. Учитывая, что результат программы формируется как последовательность шагов изменения данных, размещённых по конкретным адресам, программист вынужден детально и тщательно изучать все тонкости побочных эффектов как своего, так и смежных фрагментов программы, что оказалось серьёзным препятствием к повышению производительности труда.

9.1. Особенности представления программ на Си

- При конструировании кода программы Си-компилятор распределяет память для глобальных и локальных данных в статической памяти или в стеке, соответственно.
- Встраиваются вызовы библиотечных функций создания-удаления для динамических структур данных и обмена данными, включая ввод-вывод.
- Данные бывают константами или переменными, идентифицируемыми с помощью идентификаторов.
- Типы данных разделены на основные и производные, созданные с помощью специальных операций, включая создание неоднородных структур данных.
- Над типами данных определён конкретный набор операций, с помощью которых строятся вычислимые выражения, и схем операторов управления и описания, используемых при конструировании функций.
- Многократно используемые функции объединяются в специализированные библиотеки, часть которых включена в системы программирования на языке Си.
- Компилятору для эффективности кодирования нужна информация о типах данных переменных и результатов функций, но допустимо умолчание в случае типа int.
- Возможна спецификация вида используемой памяти, но она носит рекомендательный характер – компилятор учитывает её в меру возможности.
- Определения функций обладают некоторой свободой в конкретизации списка параметров на уровне вызова функции, что позволяет программировать функции произвольного числа параметров.
- Реализация арифметических операций обладает вариантами, зависящими от основного типа обрабатываемых скаляров в соответствии с разнообразием аппаратной поддержке этих операций.

Абстрактная машина (АМ) языка Си может рассматриваться как развитие и обобщение АМ для ассемблера, обеспечивающее возможность использовать более сложные структуры данных в качестве регистров. Определение мало отличается от АМ подмножества языка Pascal (см. часть 1).

s e c m → s’ e’ c’ m’

Сумматор расширяется до стека промежуточных значений, и появляется дополнительный регистр «Локалы» для локализации хранимых объектов:

<Стек_ значений, Локалы, Текущая_Команда, Память >

Регистр «Локалы» может быть устроен подобно регистру «E» из SECD-машины Лендина [45], только хранит он не любые значения, а скаляры или ссылки на значения в «Памяти». Начальное состояние памяти – вектор глобальных переменных, адресуемых подпрограмм и меток. Локалы и Память образуют контекст исполнения программы.

АМ различает следующие категории команд:

- засылка значений из памяти в стек;
- вычисления над безымянными операндами в стеке при обработке выражений;
- пересылка значений из стека непосредственно в память или в регистр Локалы;
- организация переходов по метке в программе;
- организация ветвлений и циклов;
- организация вызовов функций с сохранением/восстановлением локального контекста.

Определение	*Примечание*

Main (argc, argv, envp)
 	
Int argc;
Char **argv;
Char **envp;
{
For (i=0; i < argc; i++)
Printf (”arg%i:%s\n”, i, argv [i]);
For (p=0; *p != (char*)0; p++)
Printf (”%s\n”, *p);
}
	Заголовок функции.
	Описания параметров функции:
	– число аргументов,
	– вектор аргументов,
	– вектор системных переменных.
	
	Цикл вывода аргументов командной строки.

Пример 1. Программа распечатки параметров командной строки и переменных среды на языке Си [42]

9.2 Структурное программирование

Прагматика стандартного программирования не требует подробного описания – она общеизвестна и подробно исследована во многих работах. Тем не менее, следует отметить ряд моментов, связанных со структурным и функциональным программированием [50].

Сложность разработки больших программ, функционирующих в стиле локальных изменений состояния памяти, привела к идеям структурного программирования, налагающим на стиль представления программ ряд ограничений, способствующих удобству отладки программ и приближающих технику стандартного программирования к функциональному программированию [15]:

- дисциплина логики управления с избеганием переходов по меткам (goto_less_style);
- минимизация использования глобальных переменных в пользу формальных параметров процедур (global_variable_harmful);
- полнота условий в ветвлениях, отказ от отсутствия ветви “else”;
- однотипность результатов, полученных при прохождении через разные пути.

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

9.3.Функциональная модель ИП

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

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

Prog-форма языка Lisp 1.5 [58] может рассматриваться как абстрактная модель ИП, в которой при интерпретации используются отдельные ассоциативные таблицы для хранения параметров доступа к значениям переменных и определениям функций и процедур в памяти. Это позволяет независимо задавать механизмы доступа к именам переменных и наименованиям процедур.

Применение возможностей prog-выражений позволяет писать «паскалеподобные» программы, состоящие из операторов, предназначенных для исполнения (точнее, «алголоподобные»[3], т.к. они появились лет за десять до Паскаля. Но теперь более известен Паскаль).

Для примера prog-выражения приводится императивное определение функции (*LENGTH* *), сканирующей список и вычисляющей число элементов на верхнем уровне списка. Значение функции ***LENGTH*** – целое число. Программу можно примерно описать следующими словами (Стилизация примера от МакКарти [58].):

Русский язык	*Примечание*
Это функция одного аргумента L. Она реализуется программой с двумя переменными u и v. Записать число 0 в v. Записать аргумент L в u. A: Если u содержит NIL, то функция выполнена и её значением является то, что сейчас записано в v. Записать в u cdr от того, что сейчас в u. Записать в v на единицу больше	Объявление функции. Объявление рабочих переменных. Инициирование рабочих переменных. Меткой «A» помечена проверка завершения и формирования результата. Шаг продолжения функции.

[3] Algol-like

того, что сейчас записано в v. Перейти к A"	 Переход на метку «A».

Пример 2. Описание алгоритма на естественном языке

Эту программу можно написать в виде Паскаль-программы. Строкам описанной выше программы в предположении, что существует библиотека функций над списками на Паскале, соответствуют строки определения функции:

Pascal	*Примечание*
function LENGTH (L: list) : integer;	Объявление функции LENGTH, вырабатывающей целое число.
label A; var U: list; V: integer; begin	Объявление метки и рабочих переменных.
V := 0; U := L;	Инициирование рабочих переменных.
A: if null (U) then LENGTH := V;	Меткой «A» помечена проверка завершения функции и выработка её результата.
U := cdr (U); V := V+1;	Шаг продолжения функции.
goto A; end;	Переход на метку «A».

Пример 3. Представление программы на языке Pascal

Переписывая в это определение виде лисповского S-выражения, получаем программу:

Lisp	*Примечание*
(defun LENGTH (L) (prog (U V) (setq V 0) (setq U L) A (cond ((null U) (return V)))	Объявление функции LENGTH, реализуемой в императивном стиле с двумя рабочими переменными. Инициирование рабочих переменных. Меткой «A» помечена проверка завершения функции и выработка её результата.

(setq U (cdr U)) (setq V (+ 1 V)) (go A)))	Шаг продолжения функции. Переход на метку «A».
(LENGTH '(A B C D)) (LENGTH '((X . Y) A CAR (N B) (X Y Z)))	Применение функции даёт 4. Применение функции даёт 5.

Пример 4. Представление программы на языке Lisp

Последние две формы представляют применение функции. Их значения четыре и пять, соответственно. Prog-форма имеет структуру, подобную определениям функций и процедур в Паскале: (PROG, список рабочих переменных, последовательность операторов и атомов ...) Атом в списке выполняет роль метки, локализующей оператор, расположенный вслед за ним. Метка A, как и в примерах 2 и 3, локализует оператор, начинающийся с ветвления.

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

Для присваивания рабочей переменной применяется форма *SET*. Чтобы присвоить переменной pi значение 3.14 пишется (SET (QUOTE PI)3.14). SETQ подобна *SET*, но она еще и блокирует вычисление первого аргумента. Поэтому (SETQ PI 3.14) – запись того же присваивания. SETQ обычно удобнее. *SET* и SETQ могут изменять значения любых переменных из более внешних функций.[4] Значением *SET* и SETQ является значение их второго аргумента.

Обычно в программе действия выполняются последовательно. Выполнение действия понимается как его вычисление и отбрасывание его значения. Действие программы обычно выполняются в большей степени ради эффекта действия, чем ради вычисленного значения.

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

[4] Clisp – действуют на любом уровне.

Условные выражения в качестве действий программы являются базовым средством представления ветвлений. Если ни одно из пропозициональных выражений не истинно, то программа продолжается действием, следующим за условным выражением.[5]

RETURN **–** нормальное завершение программы. Аргумент ***RETURN*** вычисляется, что и является значением программы. Никакие последующие действия не вычисляются.

Prog-выражение может быть рекурсивным.

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

Pascal	*Примечание*
function REV (x: list) :list;	Определение функции REV, вырабатывающей список.
label A, B;	Объявлены метки
var y, z: list;	и локальные переменные.
begin	
A: if null (x) then REV := y;	Меткой «A» помечена проверка завершения и результат.
z := car (x);	Выбор очередного элемента для реверсирования.
if atom (z) then goto B;	Для атома переход на метку «B» в обход реверсирования.
z := REV (z);	Замена элемента на его реверс.
B: y := cons (z, y);	
x := cdr (x);	Меткой «B» помечена сборка реверсируемого списка.
goto A;	Шаг продвижения по списку.
end;	Переход на метку «A» для дальнейшего реверсирования.

Пример 5. Представление программы на языке Pascal

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

[5] В Clisp все ветвления работают так.

Lisp	*Примечание*
(defun REV (X) (prog (Y Z)	Определение функции REV, реализуемая императивном с переменными.
A (cond ((null X)(return Y))) (setq Z (car X)) (cond ((atom Z)(goto B))) (setq Z (rev Z))	Меткой «A» помечено завершение и дан результат. Выбор очередного элемента для реверсирования. Для атома переход на метку «B» в обход реверсирования. Замена элемента на его реверс.
B (setq Y (cons Z Y)) (setq X (cdr X)) (go A)))	Меткой «B» помечена сборка реверсируемого списка. Шаг продвижения по списку. Переход на метку «A» для дальнейшего реверсирования.

Пример 6. Представление программы на языке Lisp

В принципе, *SET* и `SETQ` могут быть реализованы с помощью ассоциативного списка примерно так же, как и поиск значения, только с сохранением связей, расположенных ранее изменяемой переменной

```
(defun SET (X Y) (cons (cons X Y) Alist)) )
```

Введенное таким образом присваивание обеспечивает вычислимость левой части присваивания, т.е. можно в программе вычислять имена переменных, значение которых предстоит поменять.

9.4. Спецификация

Таблица 3

Парадигматическая характеристика языков, поддерживающих процедурно-императивную парадигму программирования

Параметр	*Конкретика*
1. Эксплуатационная прагматика ЯП	Программирование решений задач, обладающих точной постановкой задачи и практичным алгоритмом без претензий на тщательное исследование
2. Особенности системы понятий	Чёткое разделение данных и средств их обработки – динамика данных и статика процедур. Раздельные пространства имён для разных категорий фрагментов программы. Отдельные блоки памяти при разной дисциплине доступа к данным

3.	Перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Ключевые слова, скаляры, идентификаторы, переменные, операции, выражения, структуры данных, элементы СД, оператор, присваивание, блок, процедура, функция, указатель, метка, GOTO, ветвление, цикл, вызов процедуры, применение функции
4.	Базовые средства	0, Null Структурное программирование
5.	Семантические расширения	Метки, GOTO, разные типы скаляров и структур данных, конструирование типов данных, приведение типов данных, указатели, файлы, разные рамочные структуры, вариации завершения циклов, библиотеки
6.	Регистры абстрактной машины	S E C M S – стек операндов и промежуточных результатов. E – стек локальных переменных и аргументов. C – поток программы. M – вектор памяти
7.	Категории команд абстрактной машины	Засылка в стек. Вычисления над стеком. Пересылки в глобальную и локальную память. Передачи управления. Ветвления. Вызов процедуры. Возврат из процедуры
8.	Реализационная прагматика	Предпочтение статического распределения памяти по блокам для векторов заданного размера, контроля типов данных при выполнении операций и вызове процедур, операций над машинными словами. Программируемое освобождение памяти. Стеки реализованы как векторы
9.	Парадигматическая специфика	Процедурно-императивный стиль программирования обычно дополнен простыми средствами функционального программирования

10. ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Функциональный стиль программирования сложился в практике решения задач символьной обработки данных в предположении, что любая информация для компьютерной обработки может быть сведена к символьной (существование аналоговых методов принципиально не противоречит этой гипотезе). Слово “символ” здесь близко понятию “знак” в знаковых системах. Информация представляется символами, смысл которых может быть восстановлен по заранее известным правилам.

Функциональное программирование рассматривает процесс обработки данных как композицию их отображений с помощью универсальных функций. Программа при таком подходе – не более чем одна из разновидностей данных. Функциональное программирование сумело преодолеть синтаксический разнобой независимо развиваемых средств и методов организации информационных процессов. Это удалось благодаря нацеленности на проявление и ортогонализацию семантических подсистем в организации программируемых процессов. Не менее важна развиваемость представления унифицируемых структур данных и комплектов функциональных объектов. Все это позволяет языки функционального программирования рассматривать как средство сопряжения разнородных конструкций. На их основе можно осуществлять любые интегрированные построения, представимые в машинно-независимом стиле.

Языки функционального программирования (ЯФП) используют гибкие списки и абстрактные атомы. ЯФП нацелены на полный контроль типов значений при исполнении программ, допускающих динамический анализ и управление с откладыванием вычислений, автоматизацию повторного использования памяти – «сборки мусора».

Таблица 4

Конкретизация понятий в универсальном языке программирования Lisp

Понятие	ФП: Lisp
Атом (Элементарное)	Атом, Числа и строки без ограничений на длину
Структура	Список
Переменная	Атом → ассоциативный список
Значение	Данное
Выражение	Переменная, Префиксное в виде списка, начинающегося с функции

Действие/Операция	Обработка списков, Арифметика
Условие/истина	Не NIL
Функция/ оператор	SUBR – подпрограмма, EXPR – выражение
Аргумент	Локальные переменные в ассоциативном списке
Вызов Фн/подпр	Пополнение ассоциативного списка
Определение Фн/подпр	Пополнение ассоциативного списка
Идентификатор	Атом

Методы функционального программирования основаны на формальном математическом языке представления и преобразования формул. Поэтому можно дать точное, достаточно полное описание основ функционального программирования и специфицировать систему программирования для поддержки и разработки разных парадигм программирования, моделируемых с помощью функционального подхода к организации деятельности.

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

1. Природа данных.

Все данные представляются в форме символьных выражений. Данные реализуются как древообразные структуры. Это позволяет локализовывать любые важные подвыражения. Система программирования с такими структурами обычно использует для их хранения всю доступную память, поэтому программист может быть освобожден от распределения памяти под отдельные блоки данных.

2. Самоописание обработки символьных выражений.

Важная особенность функционального программирования состоит в том, что описание способов обработки данных представляется программами, рассматриваемыми как символьные данные. Программы строятся из рекурсивных функций. Определения и вызовы этих функций, как и любая информация, могут обрабатываться как обычные данные, получаться в процессе вычислений и преобразовываться как значения.

3. Подобие машинным языкам.

Система функционального программирования допускает, что программа может интерпретировать и/или компилировать программы, представленные в виде структур данных. Это сближает методы функционального

программирования с методами низкоуровневого программирования и отличает от традиционной методики применения языков высокого уровня. Не все языки функционального программирования в полной мере допускают эту возможность, но для языка Lisp она весьма характерна. В принципе, такая возможность достижима на любом стандартном языке, но так делать не принято.

Функциональное программирование активно применяется для генерации программ и выполнения динамически конструируемых прототипов программ, а также для систем, применяемых в областях с низкой кратностью повторения отлаженных решений (например, в учебе, проектировании, творчестве и научных исследованиях), ориентированных на оперативные изменения, уточнения, улучшения, адаптацию и т.п.

10.1. Основы

Внимание к идеям функционального программирования привлёк в 1977 году Джон Бэкус в своей Тьюринговской лекции. Руководитель разработки языка Fortran и соавтор формул Бэкуса-Наура (БНФ) призвал к преодолению узости «бутылочного горлышка» императивно-процедурного стиля программирования и привёл в качестве примера проект функционального языка программирования FP, поддерживающего лаконичные формы представления обработки многомерных векторов, содержательно напоминающие отдельные конструкции языков Lisp и APL.

Сформулированная Джоном Маккарти (1958) концепция символьной обработки информации восходит к идеям Черча и других математиков, известным как лямбда-исчисление с конца 20-х годов прошлого века. Выбирая лямбда-исчислени*е* как теоретическую модель, положенную в основу языка Lisp, Маккарти предложил рассматривать функции как общее базовое понятие, к представлению и реализации которого достаточно естественно могут быть сведены все другие понятия, возникающие в практике программирования. Такое сведение вовсе не означает, что все понятия сваливаются в одну кучу, что исчезают границы между понятиями. Сведение выполнено так, что при сохранении всех понятийных границ выстроено более общее пространство, в рамках которого эти понятия упорядочены и могут взаимодействовать согласно формальным определениям разных категорий функций.

Таблица 5

Унификация понятий концептуального минимума (Pure Lisp) для безмашинного обучения методам символьной обработки представлений функций, включая отображения, отложенные действия, и другие функции высших порядков, использующие исключительно чистые функции без побочных эффектов

Конструкция	*Примеры представления*	*Трактовка*	*Пояснение*
Встроенная константа	Nil	Представление функции без параметров, результатом которой является пустой список.	Результаты таких функций хранятся непосредственно в памяти. Их получение не требует вычислений.
Элементарное значение	List X A	Представление неопределённой функции, которая может быть в дальнейшем определена как значение переменной или представление конкретной функции.	
Идентификатор	A X ATOMIC IDENT	Представление функции, арность, категория и определение которой задаются разными средствами связывания атомов с их смыслом. (Связывание аргументов с параметрами при вызове функции или именование определения.)	
Именованная константа	A	Представление функции без параметров, в любой позиции программы выдающей ранее заданное значение.	

Переменная	X	Представление функции без параметров, результат которой может зависеть от контекста, например, от области видимости при вызове других функций.	
Составное значение	‘(A B C) ‘(A (B C) D) ‘(A . B)	Результат унарной функции QUOTE, препятствующей вычислению своего аргумента: (QUOTE (A B C)) (QUOTE (A (B C) D)) (QUOTE (A . B))	Блокировка вычислений, в частности для организации отложенных действий.
Вызов функции	(FN LIST-FRMS)	Представление выражения в виде списка из представления функции и представлений её параметров в виде выражений. При необходимости результат функции вычисляется с помощью универсальной функции APPLY.	Общая схема вычислений. При вызове функции происходит локальное связывание аргументов со значениями параметров, сохраняемое в стеке.
Выражение (форма)	X FNAME (CAR ‘(a b c))	Представление аргумента универсальной функцией EVAL, пригодное для вычисления значения этого аргумента.	Вывод значений по представлению выражений.
Ветвление	(COND ((EQ X A) Y) ((ATOM X) D) -- -- (T Z))	Специальная мультифункция над произвольным числом аргументов, каждый из которых является списком из представлений предиката и соответствующей ему ветви. При пус-	Метод организации частичных вычислений

		том списке – результат Nil.	
Определение безымянной функции	(LAMBDA (x y) (expr x y))	Результат специальной бинарной функции LAMBDA, первый аргумент которой – список аргументов определяемой функции, а второй представляет выражение, задающее её тело.	Конструирование представления функции
Именование локального определения	(LABEL NAME DEF)	Результат специальной бинарной функции LABEL, связывающей новое имя, заданное первым аргументом, с представлением функции, заданной вторым аргументом, в котором это связывание локализовано.	Поддержка многократного применения функции
Вычисление	(EVAL expr)	Результат универсальной функции EVAL, анализирующей структуру представления выражения и выбирающей метод вычисления значения выражения.	Поддержка управления ходом вычислений, включая возобновление отложенных действий.

Практическое программирование обычно выходит за круг константно определённых программ, поддерживаемых чистыми функциями. Для целей решения новых задач, отладки и оптимизации программ на компьютере возникает расширение пространства используемых категорий функций, обладающих разными побочными эффектами.

Таблица 6

Трактовка основных понятий программирования, унифицированных как функции, для практичного расширения средств отладки и оптимизации программ и решения новых задач

Конструкция	*Примеры представления*	*Трактовка*	*Пояснение*
Вывод данных	(PRINT X) ?X	Представление тождественной псевдо-функции PRINT с побочным эффектом, заключающемся в размещении эквивалентного её аргументу текста на экране или в заданном файле.	(PRINT X) = X Результат совпадает с аргументом, поэтому размещение в программе вывода данных может не влиять на ход выполнения программы.
Ввод данных	(READ) !	Представление псевдо-функции без параметров READ, побочным эффектом которой является конструирование представления данного, эквивалентного строке, набранной или размещённой в заданном файле.	Включение в программу средств ввода данных позволяет управлять ходом вычислений без редактирования текста программы, что позволяет повысить надёжность отладки.
Обработка прерываний, ошибок, исключений, неожиданностей и т.п.	(ERROR N “message” continuation)	Представление псевдо-функции, обеспечивающей для заданного номера события поясняющее сообщение и возможное продолжение процесса вычислений. По умолчанию – восстановление начального контекста.	Поддержка вычислений в стиле бэктреккинга.
Элементарное значение	123 3.14 4/5	Представление самоопределимой функции без аргументов,	Результаты такой функции хранятся непо-

	«строка»	результатом которой является её собственное представление.	средственно в памяти. Их получение не требует вычислений.
Операции над числами	(+ 1 2 3 4) (* 1 2 3 4) (- 1 2 3 4) (/ 1 2 3 4)	Представление мультифункций над произвольным числом аргументов. Аддитивные операции при пустом списке аргументов вырабатывают число 0. Мультипликативные – число 1.	(+ 1 2 3 4) = 1+2+3+4 (* 1 2 3 4) = 1*2*3*4 (- 1 2 3 4) = 1-2-3-4 (/ 1 2 3 4) = 1/2/3/4
Арифметические выражения	(+ (* 3 5 D) (- A 8) (/ 1 2 X))	Представление результатов вычислений не требует определения типов реализуемых чисел, зависящих от формата кода или длины машинного слова.	(/ 1 2) = ½ Длина целых чисел не ограничена, результат целочисленного деления может быть представлен как дробь без потери точности, точность вещественных может быть задана.
Именование глобальной функции	(DEFINE NAME DEF)	Результат специальной бинарной функции DEFINE, связывающей новое имя, заданное первым аргументом, с глобальным определением функции, представленным вторым аргументом.	Поддержка комбинаторики независимых определений функций
Конструирование представления именованной глобальной функции	(DEFUN NAME ARGS DEF)	Результат специальной функции DEFUN, конструирующей по списку аргументов и опре-	

		деляющему выражению новое глобальное определение функции и связывает его с именем.	
Область видимости рабочих переменных	(LET LIST EXPR)	Специальная функция LET строит область видимости, в которой определены значения рабочих переменных согласно списку LIST, используемых при вычислении выражения EXPR.	Поддержка иерархии наследуемых определений
Область видимости вспомогательных функций	(FLET LIST REXPR)	Специальная функция FLET строит область видимости, в которой определены вспомогательные функции согласно списку LIST, используемых при вычислении выражения EXPR.	
Конструирование специальных функций	(MACRO)	Специальная функция MACRO создает определения новых специальных функций, обрабатывающих представления своих параметров без их предварительного вычисления.	Поддержка альтернативных методов вычислений
Цикл	(LOOP …)	Специальная функция, один из параметров которой является предназначенным для многократного вычисления телом цикла, а остальные используются при управлении	Поддержка традиционных схем представления программ

		кратностью вычисления тела цикла.	
Императивные вычисления	(PROG (V ..)….)	Специальная функция, выполняющая последовательное вычисление выражений, рассматриваемых как операторы. Результат выделен функцией RETURN или определён последним оператором.	
Параллельные вычисления	(MULTIPLY …)	Специальная функция, выполняющая вычисление своих параметров в произвольном порядке, не заданном заранее. Значения всех параметров доступны объемлющим функциям.	Подготовка параллельных вычислений
Компиляция функций	(COMPILE …)	Специальная функция, побочный эффект которой заключается в создании кода функции, оттесняющего её символьное определение.	Оптимизация отлаженных функций по скорости выполнения
Структуро-разрушающие функции	(RPLACA X Y) (RPLACD X Y) (CONC AL BL) (MAPCON FN AL)	Представление функций с побочными эффектами, вызванными использованием памяти аргументов при конструировании результата, при наличии их чисто функциональных эквивалентов: (RPLACA X Y) = (CONS Y (CDR X)) (RPLACD X Y) =	Поддержка синхронизации независимых участков программы, возникающих при вычислении рекурсивных функций, независимых параметров, выполнении итераций и т.п..

		(CONS (CAR X) Y) (CONC AL BL) = (APPEND AL BL) (MAPCON FN AL) = (MAPLIST FN AL)	

Управление обработкой информации в лямбда-исчислении осуществляется в рамках иерархии свободных и связанных переменных, реализуемых с помощью таблицы соответствия символов и их смысла. Обработка представляется посредством правил интерпретации выражений, построенных из всюду определенных функций, аргументы которых могут быть упорядочены. Общностью такое построение сравнимо с аксиоматической теорией множеств.

Следует отметить некоторую разницу в понимании принципов ФП, сложившуюся в теории и практике программирования. Эта разница чётко проявилась при стандартизации языка Lisp в 1980-е годы в виде принятия двух стандартов: LISP1 – академический и LISP2 – производственный. Теоретически достаточно исследовать чисто функциональные лаконичные представления программ, поведение которых не зависит от побочных эффектов и результат программы может быть получен системой редукций её представления. Процессы применения редукций можно выбирать в зависимости от стратегии вычислений. На практике основная трудоёмкость связана с отладкой программы на базе конкретной системы программирования, поддерживающей определённую стратегию вычислений или дающей возможность явного управления ходом выполнения программы в зависимости от данных. Чисто функциональная программа при оптимизирующей компиляции может быть сведена к её формальному результату без генерации исполнимого кода.

По отношению к проблемам определения языков и систем программирования (СП) основные идеи ФП сложились при реализации языка Lisp и в работах Венской лаборатории IBM в начале 1960-ых годов. Эти идеи оказались трудными для восприятия в пионерскую эпоху программирования, но в настоящее время их популярность растет, что и обуславливает целесообразность фундаментальных исследований в сфере функционального программирования, проектирования и моделирования.

С конца 70-х годов появились Лисп-процессоры, доказавшие, что неэффективность функционального программирования обусловлена характеристиками оборудования, а не стилем программирования. Функциональные мини-языки хорошо показали себя и при решении задач аппаратного уров-

ня. Все это превращает ФП в практичный и перспективный инструментарий. Такая схема подтверждается самой историей развития диалектов языка Лисп и родственных ему языков программирования.

Механизм функций основательно развит математиками, и это позволяет программистам наследовать выверенные построения, обладающие предельно высокой моделирующей силой.

Понятие «функция» связано с понятиями аргумента функции, области ее существования и значения, соответствия между ее аргументами и результатами, а также правил применения функции к ее аргументам. Существуют различные точки зрения на природу всех этих терминов, на границы определяющих их множеств, на возможность их взаимодействия в более общих построениях.

Изучение функционального программирования начинается с овладения техникой работы с так называемыми «чистыми», строго математическими, идеальными функциями. Для реализации таких функций характерен отказ от необоснованного использования присваиваний и низкоуровневого управления вычислениями в терминах передачи управления. Такие функции удобны при отладке и тестировании благодаря независимости от контекста описания и предпочтения явно выделенного чистого результата. Трудоемкость отладки композиций из хорошо определенных функций растет аддитивно, а не мультипликативно. Кроме того, системы из таких функций могут развиваться в любом направлении: сверху вниз и снизу-вверх (а также расширяясь и сужаясь, если понадобится). Можно быстро продвинуться по сложности решаемой задачи, не отвлекаясь на синтаксическое разнообразие и коллизии при обработке общих данных. Для обучение такому стилю программирования на языке Lisp был создан язык Pure Lisp и определён его интепретатор. Концептуально близкие идеи «структурного программирования» были сформулированы лишь более чем через десять лет.

Особенно интересны рекурсивные функции и методы их реализации в системах программирования. Интуитивное понятие функции**,** в отличие от классического понятия множества, отчасти содержит концепцию времени: сначала аргументы вычисляются, затем в соответствии с заданным алгоритмом интерпретации символьных выражений строится значение функции – её результат, возможно, явно зависящий от результатов других функций или от этой же функции, но при других, ранее вычисленных, значениях аргументов. Явно подразумевается, что значения аргументов вычисляются до того, как к ним применяется функция. Но если в качестве данных допускать не только значения, но и символьные формы для вычис-

ления этих значений, то вопрос о времени вычисления аргументов можно решать не столь категорично. Кроме обычных функций, аргументы которых вычисляются предварительно, в ряде случаев можно рассматривать и реализовывать специальные функции, способные обрабатывать аргументы нестандартным способом по любой заданной схеме – отложенные действия (Lazy evaluation). Такое развитие понятия функции напоминает развитие понятия числа по мере расширения класса удобных формул над числами (в этом отношении показательна аналогия с историей математики. Эволюция понятия числа содержит много резких обобщений с сохранением основных алгебраических свойств базовых операций и удобства работы с формулами. Так от натуральных чисел перешли к отрицательным, ввели ноль, дробные, вещественные, иррациональные, комплексные и т.д.).

Математическое понятие «функция» отражает связь между элементами заданных множеств. Функция – это «закон», по которому каждому элементу одного множества ставится в соответствие некоторый элемент другого множества. В частности, некоторый задуманный заранее алгоритм по варьируемым входным данным выдаёт определённые выходные данные. Числовые функции удобно представляются на рисунках в виде графиков. Часто вместо определения функции даётся её интуитивное описание; то есть понятие функции задаётся на обычном языке, используя слова «правило» или «соответствие». Функции могут быть частично определёнными, отображающими часть множества, и многозначными, область значений которых является семейством множеств, т.е. хотя бы одному значению аргумента соответствует два или более значений функции. Во многих физических и математических задачах возникла потребность в обобщении понятия «функция». В понятии обобщённой функции находит отражение тот факт, что реально нельзя измерить значение физической величины в точке, а можно измерять лишь её средние значения в малых окрестностях данной точки. Таким образом, техника обобщённых функций служит удобным и адекватным аппаратом для описания распределений различных физических величин. Теория обобщенных функций была впервые построена Н.М. Гюнтером в 1916 году и позже развивалась и пропагандировалась С.Л. Соболевым и затем Л. Шварцем. Обобщённые функции использовались Дираком в его исследованиях по квантовой механике.

Можно сказать, что функциональное программирование базируется на «живом» понятии, развивающимся для нужд практики с сохранением техники удобной работы с математическими формулами, разрабатываемыми в соответствии с эволюцией решаемых задач. Прежде всего понятие «функция» обогащается представлением о псевдо-функциях, используемых с

целью представления аппаратных, зависимых от устройств действий (ввод/вывод, сообщения, рисование и т.п.), фактически осуществляющих известный побочный эффект в результате работы конкретного оборудования и ОС – минимального контекста исполнения любой практически полезной программы. Формально все псевдо-функции обязательно выполняют и отображение аргументов в результаты, что позволяет им равноправно участвовать в любой позиции формулы, задающей вычислительный процесс. Формальный результат сопровождается дополнительными эффектами. Этот переход обеспечивает при необходимости корректное моделирование всей традиционной программотехники, включая присваивания, передачи управления, системные вызовы, обработку файлов и доступ к любым устройствам. Все эти непредсказуемо сложные машинно-зависимые реалии при функциональном стиле программирования локализованы, наращиваются на ранее отлаженный каркас функционирования программы, их представления могут быть четко отделены от сущности решаемой задачи. Функциональное программирование снижает трудоёмкость отладки программ благодаря созданию коллекций абстрактных лаконичных универсальных функций, приспособленных к многократному применению в разных программах. Модное поветрие выделять монады якобы ради сохранения чистоты функционального стиля скорее вызвано стремлением сохранить привычный императивный стиль организации ввода-вывода, который в отличие от оператора присваивания не искажает контекст вычислений и следовательно чистоте обработки данных с помощью функций не препятствует.

Здание функционального программирования получает логическое завершение на уровне определения функций высших порядков, удобных для синтаксически управляемого конструирования программ на основе спецификаций, типов данных, визуальных диаграмм, формул и т.п. [45]. Функциональные программы могут играть роль спецификации обычных итеративно-императивных программ. Иногда такой переход не вызывает затруднений. Факториал можно определить рекурсивно как сведение к значению функционала от предыдущего числа, но столь же понятно и определение в виде цикла от одного до `N`. На языке Sisal и цикла для этого не требуется, достаточно задать границы области, элементы которой перемножаются `(* 1, ,N)`. Конечно, числа Фибоначчи легко порождать с помощью рекурсивного восходящего процесса, но и цикл с заданной границей заработает вполне практично. Однако бывают несложные задачи, для которых такой переход не столь прост. Отнюдь не любая обработка произвольной последовательности легко излагается в терминах векторов, и многие задачи

на больших графах могут весьма сложно приводиться к итеративной форме. Заметные трудности в процесс сведения *рекурсии* к итерации создает динамика данных и конструируемые функции. Даже реализация равенства для произвольных структур данных при неизвестной размерности и числе элементов – дело непростое. Известно, что лаконичность рекурсии может скрывать нелегкий путь. А. П. Ершов в предисловии к книге П. Хендерсона привел поучительный пример не поддавшегося А. Чёрчу решения задачи о рекурсивной формуле, сводящей вычитание единицы из натурального числа к прибавлению единицы `{1 −1 = 0 | ( n +1 ) −1 = n}`, полученного С. Клини лишь в 1932 году:

Алгоритм	*Примечание*
n –1 = F(n, 0, 0) где F(x, y, z) = если (x = 1) то 0 иначе если ((y +1) = x) то z иначе F(x, y +1, z +1)	Сведение к вызову вспомогательной функции. Вспомогательная функция. Объявление значения от 1. Проверка достижения предыдущего числа. Шаг рекурсии с наращиванием параметров.

Пример. 7. Выражение «-1» через «+1»

Решение получилось через введение формально усложненной функции F со вспомогательными параметрами, что противоречит интуитивному стремлению к монотонности при движении от простого к сложному. Универсальность понятия «функция» и разнообразие видов его применения позволяет унифицировать используемые при описании процессов понятия «действие», «значение», «формула», «переменная», «выбор варианта» и пр. Все это – разные категории функций с различными формами унифицированного представления (записи, изображения) в тексте программы и правилами интерпретации (выполнения, вычисления), обеспечивающими получение результата функции при исполнении программы. Аргументами функции могут быть готовые данные или результаты других функций. Возможны ограничения на типы данных, допускаемых в качестве аргументов – тогда речь идет о частичных функциях. Такие функции должны выяснять допустимость фактических параметров и сообщать о несоответствии. Удобно, если часть такой работы берет на себя компилятор в классической традиции статического контроля правильности типов данных, но динамический контроль типов данных в условиях, характерных для современных информационных сетей, может быть надежнее, чем традиционный

статический анализ, сложившийся для замкнутых, защищенных от несанкционированного доступа конфигураций, обеспечивающий гарантии сохранения скомпилированного кода программы при его использовании. (Имеется в виду вероятность искажения скомпилированного кода при его эксплуатации на компьютере в сетях.) Это приводит к компромиссу в виде объектно-ориентированного программирования, допускающего динамический контроль типов данных.

Важно отметить, что преимущества ФП обусловлены не только семантикой используемых языков программирования, но и особенностями его поддержки в системах программирования:

1. Представления данных (чисел, строк, имён, списков) не ограничены по длине.
2. Полностью автоматизировано первичное и повторное распределение памяти – «сборка мусора».
3. Выделен комплекс базовых средств для программирования без побочных эффектов в памяти программы, достаточный для представления ленивых вычислений, отображений и других функций высших порядков.
4. Встроены средства управления вычислениями, расширяющие возможности СП в направлении основных парадигм программирования.

10.2. Универсальный язык программирования Lisp

Идеи ФП достаточно полно поддержаны в проекте Lisp 1.5, выполненном Дж. МакКарти и его коллегами. В этом исключительно мощном языке не только реализованы основные средства, обеспечившие практичность и результативность ФП, но и впервые опробован целый ряд поразительно точных построений, ценных как концептуально, так и методически и конструктивно, понимание и осмысление которых слишком отстает от практики применения. Понятийно-функциональный потенциал языка Lisp 1.5 в значительной мере унаследован стандартом Common Lisp, но некоторые идеи пока не получили достойного развития. Вероятно, это дело будущего – для нового поколения системных программистов.

Языки ФП достаточно разнообразны. Существует и активно применяется более трехсот диалектов Лиспа и родственных ему языков: Interlisp, muLisp, Clisp, Scheme, ML, Cmucl, Logo, Hope, Sisal, Haskell, Miranda и др. При сравнении языков и парадигм программирования часто классифицируют функциональные языки по следующим критериям: “ленивый” или

аппликативный, последовательный и параллельный. Например, ML является аппликативным и последовательным, Erlang – аппликативным и параллельным, Haskell – “ленивым” и последовательным, а Clean – параллельным и “ленивым”. Scheme, ML, Hope, Haskell – типичные представители академического стандарта LISP1, а Common Lisp, Clisp, Sisal, Cmucl – производственного стандарта LISP2.

В рамках проекта **.Net** выполнено большое число реализаций весьма различных языков программирования, в их числе Haskell, Sml, Scheme, Mondrian, Mercury, Perl, Oberon, Component Pascal, разрабатан F# – новый язык ФП [40, 43]. Еще большее разнообразие предлагает проект DotGNU**,** пытающийся отстоять приоритет в области разработки переносимого ПО. Развиваются линии учебного и любительского программирования и методично осваивается техника выстраивания иерархии абстрактных машин при определении языков программирования.

Разработка языков функционального программирования (ЯФП) и приспособленность средств ФП к быстрой отладке, верификации, их лаконизм, гибкость, конструктивность и моделирующая сила позволяют рассматривать ФП как основу информационной среды обучения современного программирования на всех уровнях проблематики от алгоритмизации до включения в социальный контекст приложений разрабатываемых ИС.

10.3. Отображения и функционалы

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

Определение отображений – ключевая задача информатики. Построение любой информационной системы сопровождается реализацией большого числа отображений. Сначала выбираются данные, с помощью которых представляется информация. В результате по данным можно восстановить представленную ими информацию – извлечь информацию из данных (по записи числа восстановить его величину). Потом конструируется набор

структур, достаточный для размещения и обработки данных и программ в памяти компьютера (по коду команды можно выбрать хранимую в памяти подпрограмму, которая построит новые коды чисел или структур данных).

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

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

Это позволяет задать порядок перебора множества и метод передачи аргументов для вычисления отображающей функции. При обходе структуры, представляющей множество, отображающая функция будет применена к каждому элементу множества.

Проще всего выработать структуру множества результатов, подобную исходной структуре. Но возможно, что не все полученные результаты нужны, или требуется собрать их в иную структуру, поэтому целесообразно заранее прояснить еще ряд вопросов:

1) где размещается множество полученных результатов;
2) чем отличаются нужные результаты от полученных попутно;
3) как строятся итоговые данные из отобранных результатов.

При функциональном стиле программирования ответ на каждый из таких вопросов может быть дан в виде отдельной функции, причем роль каждой функции в схеме реализации отображения четко фиксирована. Схема реализации отображения может быть представлена в виде определения, формальными параметрами которого являются обозначения функций, выполняющих эти роли. Такое определение называется «функционал». Говоря более точно, функционал может оперировать представлениями функций в качестве аргументов или результатов.

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

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

Отказ от барьера между представлениями функций и значений дает возможность использовать символьные выражения как для изображения заданных значений, включая любые структуры над числами и строками, так и для представления функций, обрабатывающих любые данные. (Напоминаем, что определение функции представляется как данное.) Таким образом, функционалы в ЯФП – это функции, которые используют в качестве аргументов или результатов представления других функций, а не собственно функции как принято говорить. При построении функционалов переменные могут играть роль имен функций, определения которых находятся во внешних формулах, использующих функционалы.

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

Определение	*Примечание*
(DEFUN map-el(fn xl) (COND (xl (CONS (FUNCALL fn (CAR xl)) (map-el fn (CDR xl))))))	Поэлементное преобразование XL с помощью функции FN. Пока XL не пуст Присоединяем результат FN от головы XL к списку преобразованных остальных элементов.
(map-el #'1+ xl) [6]	Следующие числа
(map-el #'CAR xl)	«головы» элементов = CAR
(map-el #'length xl)	Длины элементов

Пример 8. Отображение элементов списка с помощью заданной функции

Числа представляются в Лиспе как специальный тип атома. Атом такого типа состоит из указателя с тэгом, специфицирующим слово как число,

[6] #'x – эквивалент (FUNCTION x), что является представлением функции в качестве аргумента.

тип этого числа и адрес собственно числа произвольной длины. В отличие от обычного атома, одинаковые числа при хранении не совмещаются.

Определить функцию покомпонентной обработки двух списков с помощью заданной функции FN:

Определение	*Примечание*
(DEFUN map-comp (fn al vl) (COND (al (CONS (FUNCALL fn (CAR al) (CAR vl)) (map-comp (CDR al) (CDR vl))))))	fn покомпонентно применить к соответственным элементам al и vl Пока AL не пуст Присоединяем результат FN от головAL и VL к списку преобразованных остальных элементов
(map-comp #'+ '(1 2 3) '(4 6 9))	= (5 8 12) Суммы
(map-comp #'* '(1 2 3) '(4 6 9))	= (4 12 27) Произведения
(map-comp #'CONS '(1 2 3) '(4 6 9))	= ((1 . 4) (2 . 6) (3 . 9)) Пары
(map-comp #'EQ '(4 2 3) '(4 6 9))	= (T NIL NIL) Сравнения

Пример 9. Покомпонентные действия над векторами, представленными с помощью списков

Определение	*Примечание*
(DEFUN mapf (fl el) (COND (fl (CONS (FUNCALL (CAR fl) el) (mapf (CDR fl) el)))))	 Пока FL не пуст, присоединяем результат очередной функции от EL к списку результатов остальных функций
(mapf '(length CAR CDR) '(a b c d))	= (4 a (b c d))

Пример 10. Применение списка функций к общему аргументу

Композициями таких функционалов можно применять серии функций к списку общих аргументов или к параллельно заданной последовательности списков их аргументов.

Естественно, и серии, и последовательности представляются списками.

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

Показанные построения достаточно разнообразны, чтобы можно было сформулировать, в чем преимущества применения техники функционального программирования:

- отображающие функционалы позволяют строить программы из крупных действий;
- функционалы обеспечивают гибкость отображений;
- определение функции может совсем не зависеть от конкретных имен;
- с помощью функционалов можно управлять выбором формы результатов;
- параметром функционала может быть представление любой функции, преобразующей элементы структуры;
- функционалы позволяют формировать серии функций от общих данных;
- встроенные в Clisp функционалы приспособлены к покомпонентной обработке произвольного числа параметров;
- любую систему взаимосвязанных функций можно преобразовать к одной функции, используя вызовы безымянных функций.

10.4. Отложенные действия

Результаты первых экспериментов с отложенными действиями на базе языка Lisp были опубликованы в 1964 году L.Lombardi, построившему частичный вычислитель арифметических выражений, поддерживающий пошаговое задание входных данных. При неполных или неподходящих данных конструируется так называемая остаточная программа, способная принять недостающие данные и в конце концов вычислить итоговый результат. Независимо с 1969 Футамура и с 1976 А.П. Ершов исследовали такую технику для решения проблем конструирования компиляторов и организации смешанных вычислений. Позднее появились ЯФП целенаправленно использующие разные схемы отложенных действий ради оптимизации общего процесса вычислений и динамической оптимизации программ [56].

Императивная организация вычислений по принципу немедленного и обязательного выполнения каждой очередной команды не всегда эффективна. Существует много неимперативных моделей управления процессами, позволяющих прерывать и откладывать процессы, а потом восстанавливать их и запускать или отменять. Организация такого управления, достаточного для оптимизации и программирования параллельных процессов, реализуется с помощью так называемых «замедленных» или «ленивых»

вычислений (lazy evaluation). Основная идея таких вычислений заключается в сведении вызовов функций к представлению рецептов их вычисления, содержащих замыкания функций в определенном контексте.

(λ () fn) – заблокировать вычисление «fn», превратив его в тело функции без аргументов.

(fn) – разблокировать выражение «fn» в форме вызова функции без параметров.

Непосредственное применение таких формул влечёт многократное вычисление «fn», поэтому в инструментальном концентре используется реализационная структура данных, названная «рецепт», хранящая варианты представления выражения.

{ [F (fn . e)] | [T x]}

Сначала рецепт представлен как «(F (fn . e))», где «(fn . e)» – это замыкание выражения «fn» в пределах контекста «e». Попытка вычисления рецепта приводит к замене его результатом. По прежнему адресу размещается структура «(T x)», в которой «x» равен результату вычисления «fn» в контексте «e».

Таблица 7

Дополнительные команды АМ для эффективной поддержки отложенных выражений

Команда	*Пояснение*
LDE	Загрузка отложенного выражения с созданием рецепта
RPL	Замена рецепта его результатом
APN	Возобновление отложенного выражения

Вычисляться такими командами рецепт может не более чем один раз, и то если его результат действительно нужен.

Таблица 8

Дополнение АМ для эффективного выполнения отложенных действий [45]

Команда	*Результат*
s e (LDE f . c) d	→ ([F (f . e)] . s) e c d
(x) e (RPL) ([F (F . e)] ee c . d)	→ (x . s) ee c d при этом ([F (F . e)] → [T x] [7]
([T x] . s) e (APN . c) d	→ (x .s) e c d
([F (f . e)] . s) ee (APN . c) d	→ Nil e f ([F (F . e)] ee c . d)

[7] Включение в квадратные скобки «[ ]» здесь символизирует размещение по тому же адресу.

Эффект отложенных вычислений можно продемонстрировать на обработке бесконечных структур данных, в которых продолжение структуры представлено как вычисление следующих элементов.

Фрагмент	*Примечание*
(def beg (LAMBDA (k lst) (cond ((EQ k Nil)()) (T (cons (car lst) (beg (cdr k) (eval (cons (cdr lst) Nil)))))))) (def endless (LAMBDA (m) (cons m (lambda () (endless (cons m m)))))) (beg '(a b c) (endless 'A)) ; = (A (A . A) ((A . A) A . A))	Вырезка начального отрезка

Пример 11. Конструирование последовательности произвольной длины из бесконечного списка

10.5. Свойства атомов

Методы расширения функциональных построений могут быть применены для моделирования привычного императивно-процедурного стиля программирования и техники работы с глобальными определениями. Здесь демонстрируется расширение базовой схемы обработки символьных выражений и представленных с их помощью функциональных форм на примере механизма списков свойств атомов. В результате можно собирать функционально полное определение гибкой и расширяемой реализации языка программирования, что и показано на примере Лисп-интерпретатора, написанном на Лиспе.

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

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

Здесь достаточно принять к сведению, что реализация атомарных объектов – это сложная структура данных, в свою очередь представленная списками.

С помощью функции GET в форме `(GET x i)` можно найти для атома x свойство, индикатор которого равен `i`.

Значением `(GET 'FF 'EXPR)` будет `(LAMBDA (X) (COND ... ))`, если определение `FF` было предварительно задано с помощью `(DEFUN FF (X)( COND ... ))`.

Свойство с его индикатором может быть вычеркнуто – удалено из списка функцией REMPROP в форме `(REMPROP x i)`.

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

10.6. Гибкий интерпретатор

В качестве примера повышения гибкости определений приведено упрощенное определение Лисп-интерпретатора на Лиспе, полученное из M-выражения, приведенного Дж. Маккарти в описании Lisp 1.5 [58].

Ради удобочитаемости здесь уменьшена диагностичность, нет поствычислений и формы PROG. Лисп хорошо приспособлен к оптимизации программ. Любые совпадающие подвыражения можно локализовать и вынести за скобки, как можно заметить по передаче значения.

Определения функций хранятся в ассоциативном списке, как и значения переменных.

Функция `SUBR` – вызывает примитивы, реализованные другими, обычно низкоуровневыми, средствами.

ERROR – выдает сообщения об ошибках и сведения о контексте вычислений, способствующие поиску источника ошибки. Уточнена работа с функциональными аргументами:

Определение	*Примечание*
(DEFUN EVAL (e al)	
(COND	
((EQ e NIL) NIL)	
((ATOM e)((LAMBDA (v)	
(COND (v (CDR v))	
(T (ERROR 'undefvalue)))	Диагностика
) (ASSOC e al))	Однократность вычисление значения
)	
((EQ (CAR e) 'QUOTE) (CAR (CDR e)))	
((EQ (CAR e) 'FUNCTION)	
(LIST 'CLOSURE (CADR fn) al))	Замыкание функционального аргумента
((EQ (CAR e) 'COND) (EVCON (CDR e) al))	
(T (apply (CAR e)(evlis (CDR e) al) al))	
))	

Пример 12. Диагностика отсутствия определений при вычислении форм

Определение	*Примечание*
(DEFUN APPLY (fn args al)	
(COND	
((EQ e NIL) NIL)	
((ATOM fn)	
(COND	
((MEMBER fn '(CAR CDR CONS ATOM EQ))	Локализация списка встроенных подпрограмм
(SUBR fn agrs al))	Выполнение подпрограмм
(T (APPLY (EVAL fn al) args al))	
))	
((EQ (CAR fn) 'LABEL)	Именование локальных функций
(APPLY (CADDR fn)	
args	
(CONS (CONS (CADR fn)(CADDR fn))	
al)))	
((EQ (CAR fn) ' CLOSURE)	
(APPLY (CDR fn) args (CADDR fn)))	Применение функционального аргумента
((EQ (CAR fn) 'LAMBDA)	

(EVAL (CADDR fn) (APPEND (PAIR (CADR fn) args) al)) (T (APPLY (EVAL fn al) args al))))	

Пример 13. Применение подпрограмм и функциональных параметров

Определения ASSOC, APPEND, PAIR, LIST – стандартны.

Определение	*Примечание*
(DEFUN evcon (c a) (COND ((null c) Nil) ((evel (car c) a) (evel (cadr c) a)) (T (evel (caddr c) a))))	Возможно отсутствие ветви

Пример 14. Выбор ветви

Определение	*Примечание*
(DEFUN evlis (m a) (COND (m (cons(evel (car m) a) (evlis(cdr m) a)))))	Пока «M» не пуст, присоединяем значение его первого элемента к списку значений остальных элементов

Пример 15. Вычисление параметров

10.7. Функциональная модель взаимодействия монад

Примерно то же самое обеспечивают EVAL-P и APPLY-P, рассчитанные на использование списков свойств атома для хранения постоянных значений и функциональных определений. Индикатор MONAD указывает в списке свойств атома на правило интерпретации функций, относящихся к отдельной монаде, MACRO – на частный метод построения представления функции. Функция VALUE реализует методы поиска текущего значения переменной в зависимости от контекста и свойств атомов.

Определение	*Примечание*
(DEFUN EVAL-P (E C) (COND ((ATOM E) (VALUE E C)) ((ATOM (CAR E))(COND ((GET (CAR E) 'MONAD) ((GET (CAR E) ' MONAD) (CDR E) C)) (T (APPLY-P (CAR E)(EVLIS (CDR E) C) C)))) (T (APPLY-P (CAR E)(EVLIS (CDR E) C) C))))	До применения функции выясняем её категорию

Пример 16. Функции разных категорий

Определение	*Примечание*
(DEFUN APPLY-P (F ARGS C) (COND ((ATOM F)(APPLY-P (FUNCTION F C) ARGS C)) ((ATOM (CAR F))(COND ((GET (CAR F) 'MACRO) (APPLY-P ((GET (CAR F) 'MACRO) (CDR F) C) ARGS C)) (T (APPLY-P (EVAL F E) ARGS C)))) (T (APPLY-P (EVAL F E) ARGS C))))	Для макро-функций нет необходимости в вычислении аргументов

Пример 17. Специальные макро-функции поддерживают пост-вычисления параметров

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

Определение	*Примечание*
(DEFUN EVAL-P (E C) (COND ((ATOM E) (VALUE E C)) ((ATOM (CAR E)) ((LAMBDA (V) (COND (V (V(CDR E) C)) (T (APPLY-P (CAR E)(EVLIS (CDR E) C) C)))) (GET (CAR E) ' MONAD))) (T (APPLY-P (CAR E)(EVLIS (CDR E) C) C))))	Однократный поиск категории

Пример 18. Исключение лишнего поиска свойства MONAD

Определение	*Примечание*
(DEFUN APPLY-P (F ARGS C) (COND ((ATOM F)(APPLY-P (FUNCTION F C) ARGS C)) ((ATOM (CAR F)) ((LAMBDA (V) (COND (V (APPLY-P (V (CDR F) C) ARGS C)) (T (APPLY-P (EVAL F E) ARGS C))))(GET (CAR F) 'MACRO))) (T (APPLY-P (EVAL F E) ARGS C))))	Однократный поиск свойства

Пример 19. Исключение лишнего поиска свойства MACRO

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

Полученная схема интерпретации допускает разнообразные категории функций, реализуемые как отдельные монады, и макросредства конструирования функций, позволяет задавать различные механизмы передачи параметров функциям. Так, в языке Clisp различают, кроме обычных, обязательных и позиционных, – необязательные (факультативные), ключевые и серийные (многократные, с переменным числом значений) параметры

При разработке и отладке программ происходит развитие средств и методов обработки данных, что приводит к изменению типов представления объектов. Объекты из неопределенных становятся константами или переменными, из обработки скаляров формируется обработка структур данных, от структур данных вероятен переход к функциям и файлам и т.п. [12, 30, 41, 54].

Не менее существенное развитие происходит и на уровне постановки задачи. Задача из новой, решение которой имеет ранг пробы или демонстрации, становится направлением исследования, в котором созревают практичные или точные подзадачи, возможен и переход к поиску решений на пределе возможностей оборудования. Требования к средствам и методам решения задач на столь различных уровнях изученности привели к гибкости, присущей парадигме функционального программирования.

Определение	*Примечание*
(DEFUN eval(e a) (COND ((atom e) (cdr(assoc e a))) ((eq (car e) 'QUOTE) (cadr e)) ((eq(car e) 'COND) (evcon(cdr e) a)) (T (apply (car e) (evlis(cdr e) a) a))))	Переменная Константа Ветвление Применение функции.
(DEFUN apply (fn x a) (COND ((atom fn)(cond ((eq fn 'CAR) (caar x)) ((eq fn 'CDR) (cdar x)) ((eq fn 'CONS) (cons (car x)(cadr x))) ((eq fn 'ATOM) (atom (car x))) ((eq fn 'EQ) (eq (car x)(cadr x))) (T (apply (eval fn a) x a))))) ((eq(car fn)'LAMBDA) (eval (caddr fn) (pairlis (cadr fn) x a))) ((eq (car fn) 'LABEL) (apply (caddr fn) x (cons (cons (cadr fn)(caddr fn)) a)))))	Элементарные функции Программируемые функции Безымянная функция Именованная функция
(DEFUN evcon (c a) (COND ((eval (car c) a) (eval (cadr c) a)) (T (eval (caddr c) a))))	Выбор ветви
(DEFUN evlis (m a) (COND ((null m) Nil) (T (cons(eval (car m) a) (evlis(cdr m) a))))	Вычисление параметров

Пример 20. Самоописание Lisp-интерпретатора, предложенное Дж. Маккарти в начале 1960-х годов [58]

Наиболее очевидные следствия из выбранных принципов ФП:

- процесс разработки программ разбивается на фазы: построение базиса и его пошаговое расширение;
- рассмотрение программы как реализации алгоритма естественно дополняется табличной реализацией функций, т.е. допустимо ис-

пользование хранимых графиков функций в виде структур из аргументов и соответствующих результатов наряду с процедурами;
- прозрачность ссылок обеспечена совпадением значений одинаково выглядящих формул, вычисляемых в одинаковом контексте;
- предпочтение универсальных функций и функционально полных систем, трудоемкость первичной реализации которых компенсируется надежностью определений и простотой применения.

Язык программирования Lisp [58] и сложившееся на его основе функциональное программирование реально показали свои сильные стороны как инструментарий исследования и освоения новых областей применения вычислительной техники. Это аргумент в пользу функционального подхода к решению новых сложных задач:
- доступны преобразования программ и процессов;
- осуществима лингвистическая обработка информации;
- поддерживается гибкое управление базами данных;
- возможна оптимизация информационных систем, вплоть до полного исключения потерь информации;
- моделирование общения;
- и многое другое.

10.8. Спецификация

Таблица 9

Парадигматическая характеристика языков, поддерживающих функциональное программирование

Параметр	*Конкретика*
1) Эксплуатационная прагматика ЯП	Исследование потенциального максимума возможностей решения сравнительно новой задачи
2) Особенности системы понятий	Всё сводится к разным категориям понятия «функция», позволяющему ясно моделировать механизмы других парадигм программирования
3) Перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Атом, число, строка, список, консолидация, константа, переменная, форма, элементарная функция, мультиоперация, безымянная функция, определение функции, аргумент функции
4) Базовые средства	Nil, Pure Lisp

5) Семантические расширения	Разные виды чисел, строки, файлы, вектора, хэш-таблицы, свойства атомов, макро-определения, мемоизация, псевдо-функции, специальные категории функций, замыкания функций, multiple, пакеты, Prog, Clos, компилятор, отладчик
6) Регистры абстрактной машины	S E C D S − стек операндов и чистого результата. E – стек значений локальных переменных C – стек исполняемой программы. D – стек защиты контекста от случайных искажений.
7) Категории команд абстрактной машины	Как в Pure Lisp
8) Реализационная прагматика	Данные строятся из тэгированных указателей с автоматизацией повторного использования памяти. Система программирования использует пару интерпретатор-компилятор функций. При связывании переменных используются стек, ассоциативный список и список свойств атомов. Возможно повышение эффективности обработки данных деструктурными функциями
9) Парадигматическая специфика	Система программирования обычно поддерживает механизмы других парадигм, возможно выделенные в отдельные подсистемы (монады в языке Haskell)

11. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Исследовательская и проектная работа обычно проходит фазу поиска практичного решения. Логическое программирование (ЛП) для поддержки этой фазы предлагает важное отступление от концепции определения подпрограмм, процедур и функций. В качестве укрупненного определения действий допускаются варианты, равноправно выбираемые из конечного множества так называемых «клауз». По мере изучения задачи это множество можно пополнять в произвольном порядке. Именно эта идея составляет одну из привлекательных особенностей ЛП, выделившегося в самостоятельную парадигму. Равноправие не распространяется лишь на тупиковую

ситуацию, когда ни один предложенный вариант не приводит к целевому результату.

Парадигма логического программирования использует идею автоматического вывода информации на основе заданных фактов и правил. Логическое программирование основано на теории и аппарате формальной логики. Написанная согласно формальной логике программа является множеством логических форм, представляющих факты и правила относительно некоторой предметной области. Основным языком логического программирования признан Prolog, хотя известны и другие – Planner, ASP и Datalog. Во всех таких языках правила имеют форму клауз [13, 32]

H :- $B1_1$, …, BN_n,

понимаемую как логическое следование

if ($B1_1$ and … and BN) then H

или

B11 & … & BN → H

H называют головой правила, а $B1_1$, …, BN – телом.

Факты – это правила без тела.

Различают атомарные и составные клаузы.

Предикаты, образующие тело, могут быть выражены в стиле сопоставления с образцом.

Важный механизм – использование отрицаний в теле клауз, что приводит к немонотонной логике.

Логика программ может использовать процедурный стиль при вычислении целей.

to solve H, solve B_1, and ... and solve B_n.

Декларативный подход к пониманию программ требует от программиста систематической проверки корректности. Более того, используются преобразования логических программ в их более эффективные эквиваленты, что сближает ЛП с макро-техникой. Для повышения эффективности программ программисту следует знать особенности поведения механизма вычислений и границы вычислимости используемых выражений.

Логическое программирование сводит обработку данных к выбору произвольной композиции определений (уравнений, предикатных форм), дающей успешное получение результата. Именно обработка формул явля-

ется основой – вычисления рассматриваются как операция над формулой. При неуспехе происходит перебор других вариантов определений. В языках ЛП считают возможным прямой перебор вариантов, сопоставляемых с образцами, и организацию возвратов при неудачном выборе. Перебор вариантов выглядит как обход графа в глубину. Имеются средства управления перебором с целью исключения заведомо бесперспективного поиска.

Таблица 10

Конкретизация понятий в языках логической парадигмы на примере языка Prolog

Понятие	*ЛП: Prolog*
Атом (Элементарное)	Слово
Структура	Строка
Переменная	Именованная позиция в шаблоне
Значение	Строка
Выражение	Строка с вхождениями переменных
Действие/Операция	подстановка значения переменной
Условие/истина	
Функция/ уравнение	Вычисление или подстановка, управляемые сопоставлением шаблонов
Аргумент	Строка, сопоставленная по шаблону
Вызов Фн/подпр	Подстановка аргументов из шаблона в его определение
Определение Фн/подпр	Шаблон (имя, …) = ветвь (имя,…)
Идентификатор	Имя

Интерпретирующий автомат для выполнения недетерминированных процессов можно представить как цикл продолжения вычислений при попадании в диагностическую ситуацию. Продолжение заключается в выборе другого варианта из набора определений функционального объекта. Вычисление признается неудачным, лишь если не удалось подобрать комплект вариантов, позволяющий вычислить значение формулы.

В отличие от множества элементов набор вариантов не требует одновременного существования всех составляющих. Поэтому программирование вариантов можно освободить от необходимости формулировать все варианты сразу. В логическом программировании можно продумывать варианты отношений между образцами формул постепенно, накапливая реально встречающиеся факты и их сочетания. Содержательно такой процесс похож и на уточнение набора обработчиков прерываний на уровне оборудования. Кроме основной программы, выполняющей целевую обработку

данных, отлаживается коллекция диагностических реакций и процедур продолжения счета для разного рода неожиданных событий, препятствующих получению результата программы.

Следует иметь в виду, что варианты не образуют иерархии. Их аксиоматика подобна так называемой упрощенной теории множеств [53]. Принципиальная особенность – совпадение предикатов принадлежности и включения.

Если варианты в таком выражении рассматривать как равноправные компоненты, то неясно, как предотвратить преждевременный выбор пустого списка при непустом перечне вариантов. Чтобы решить эту задачу, в системах ЛП вводится специальная форма ESC (ТУПИК), действие которой заключается в том, что она как бы «старается» по возможности не исполняться. Иными словами, при выборе вариантов предпочитаются варианты, не приводящие к исполнению формы ESC. Такая же проблема возникает при обработке пустых цепочек в грамматиках. Аналогичная проблема решена при моделировании процессов интерпретированными сетями Петри соглашением о приоритете нагруженных переходов в сравнении с пустыми.

11.1. Операционная семантика

АС сводим к формуле:

(факт | (предикат цель) | ESC)
AML = <SCL, RL>, где RL = <S, E, C, D, R>
s e c d r → s' e' c' d' r' – переход от старого состояния к новому.
s e c d те же, что и в ФП, r – предназначен для хранения не опробованных вариантов.

В книге Хендерсона приведено обобщение абстрактной машины, поддерживающее на базовом уровне работу с вариантами с использованием дополнительного дампа, гарантирующего идентичность состояния машины при переборе вариантов [45].

s e c d r → s’ e’ c’ d’ r’

Таблица 11

Расширение абстрактной машины для выполнения альтернатив ЛП [45]

Команда	Пояснение
ALT	выбор подходящего варианта.
ESC	выход из тупиковой ситуации.

Таблица 12

Определение команд ЛП

Формат регистров	Результат
s e (ALT c1 c2 . c) d r	→ s e (c1 . c) (c . d) ((s e (c2 . c) d) . r)
s e (ESC) d Nil	→ Nil e (Esc) d Nil
s' e' (ESC) d' ((s e (c2 . c) d) . r)	→ s e (c2 . c) d r

11.2. Основы

Представление вариантов в чем-то подобно определению ветвлений, но без предикатов, управляющих выбором ветви, что по реализации напоминает вариантные записи или объединения в обычных ЯВУ. В некоторых языках, например, учебно-игрового характера, можно указать вероятность выбора варианта. В языках логического и генетического программирования считают возможным прямой перебор вариантов, сопоставляемых с образцами, и организацию возвратов при неудачном выборе.

Обычно понятие алгоритма и программы связывают с детерминированными процессами. Но эти понятия не очень усложняются, если допустить недетерминизм, ограниченный конечным числом вариантов, так что в каждый момент времени из них существует только один вариант.

В любом выражении можно выполнить разметку ветвей на нормальные и тупиковые. Тупики можно связать с различными тэгами и выставить ловушки на заданные тэги. При попадании в тупик формируется значение всей структуры, размещенной внутри ловушки.

Используя тупики и ловушки, можно организовать перебор вариантов до первого беступикового или собрать все беступиковые варианты. Второе можно сделать, используя отображения (map), а первое – первый подходящий – слегка модифицированным evcon, можно с добавочной ловушкой на прерывание при достижении успеха.

Более сложно обеспечить равновероятность выбора вариантов. Наиболее серьезно возможность такой реализации рассматривалась в проекте языка *SETL* [60]. Похожие механизмы используются в языках, ориентированных на конструирование игр, таких как Grow, в которых можно в качестве условия срабатывания команды указать вероятность.

В задачах искусственного интеллекта работа с семантическими сетями, используемыми в базах знаний и экспертных системах, часто формулируется в терминах фреймов-слотов (рамка-щель), что конструктивно очень похоже на работу со списками свойств атома в языке Lisp. Каждый объект

характеризуется набором поименованных свойств, которые, в свою очередь, могут быть любыми объектами. Анализ понятийной системы, представленной таким образом, обычно описывается в недетерминированном стиле.

11.3. Язык декларативного программирования Prolog

Prolog является наиболее известным языком логического программирования общего назначения, ассоциируемый с проблемами искусственного интеллекта и компьютерной лингвистики. Он базируется на логике первого порядка и в отличие от обычных языков программирования приоритетно использует декларативность. Логическая программа выражается в терминах отношений, представляемых как факты и правила. Применялся для автоматизации доказательств теорем и разработки экспертных систем, включая лингвистические процессоры для естественных языков. Выполнение логической программы понимается как ответ на запрос над системой отношений. Отношения и запросы конструируются из термов и клауз.

Язык программирования Prolog предложен в 1972 году А. Колмерауэром. (Alain Colmerauer), процедурную интерпретацию языка выполнил Р. Ковальский (Robert Kowalski) и дал её описание в 1973 году, опубликованное в 1974 году. Практичность языка резко возросла благодаря созданию Д. Уорреном (David Warren) компилятора в 1977 году, что приблизило скорость символьной обработки к эффективности языка Lisp.

Существует Pure Prolog в качестве семантического базиса языка, удобного для изучения основных механизмов Prolog-машины [67].

Итеративные алгоритмы можно программировать как рекурсивные функции.

Prolog-машина ищет ответ на запрос в имеющейся системе отношений и, если находит, то сообщает его.

Опубликованы алгоритмы мета-интерпретатора для языка Pure Prolog, показывающие его самоприменимость и расширяемость [67].

Определение	*Примечание*
Factorial (N, F) Factorial (0, 1) Factorial (x+1, F) ← F := Factorial (x, y) * (x + 1)	Функция реализуется двумя клаузами: – констатация, что от 0 – значение 1; – декларация отношения между значениями функции на соседних числах

Пример 21. Вычисление факториала [67]

Определение	*Примечание*
partition([], _, [], []). partition([X | Xs], Pivot, Smalls, Bigs) :- (X @< Pivot -> Smalls = [X | Rest], partition(Xs, Pivot, Rest, Bigs) ; Bigs = [X | Rest], partition(Xs, Pivot, Smalls, Rest)). quicksort([]) --> []. quicksort([X | Xs]) --> { partition(Xs, X, Smaller, Bigger) }, quicksort(Smaller), [X], quicksort(Bigger).	Разбиение на участки Запуск сортировки

Пример 22. Быстрая сортировка [67]

Отображение

Определение	*Примечание*
maplist(_P, [], []). maplist(P, [X1 | X1s], [X2 | X2s]) :- call(P, X1, X2), maplist(P, X1s, X2s).	Отображать нечего Можно выделить начало Вызов от первого Отображение остального

Пример 23. Отображение [67]

Интерпретатор Pure Prolog

Определение	*Примечание*
mi1(true). mi1((A,B)) :- mi1(A), mi1(B). mi1(Goal) :- Goal \= true, Goal \= (_,_), clause(Goal, Body), mi1(Body).	 Равноправие вариантов. Продвижение к цели

Пример 24. Расширяемый скелет Prolog-интерпретатора. [67]

Определение	*Примечание*
Доказуемость	
provable(true, _) :- !. provable((G1,G2), Defs) :- !, provable(G1, Defs), provable(G2, Defs). provable(BI, _) :- predicate_property(BI, built_in), !, call(BI). provable(Goal, Defs) :- member(Def, Defs), copy_term(Def, Goal-Body), provable(Body, Defs).	Варианты. Встроенные свойства. Движение к цели.

Пример 25. Выводимость цели. Обобщение интерпретатора [67]

provable(Goal, Defs) – истина, если цель *Goal* выводима по отношению к *Defs*, представленным списком клауз в форме *Head-Body*.

полная реализация *backtracking* требует детерминированного накопления результатов **findall**. Отдельная альтернатива представляется как список целей и ветвей, которые могут использоваться как список альтернатив.

Определение	*Примечание*
Бэктрекинг	
mi_backtrack_([[]-G|_], G). mi_backtrack_(Alts0, G) :- resstep_(Alts0, Alts1), mi_backtrack_(Alts1, G).	Других вариантов нет. Перебор других вариантов.

Пример 26. Возвраты при неудачном выборе клаузы [67]

Если ни одна цель не доказана, то выбирается решение из внутренней очереди. Вторая клауза описывает вычисление.

Существуют версии языка, приспособленные к работе с функциями высших порядков (HiLog, λProlog) и с модулями.

Стандарт ISO языка Prolog поддерживает компиляцию, хвостовую рекурсию, индексирование термов, хэширование, обработку таблиц.

Яркая реклама ЛП в рамках японского проекта компьютерных систем 5-го поколения утихла по мере прогресса элементной базы. Несмотря на широкое использование языка в научных исследованиях и образовании, логическому программированию пока не удалось внести существенный вклад в компьютерную индустрию. Весьма вероятно, что причина кроется в том, что любое производство предпочитает достаточно изученные задачи, а сильная сторона ЛП проявляется на классе недоопределённых задач.

Принято считать, что однозначное решение задачи в виде четкого алгоритма над хорошо организованными структурами и упорядоченными данными – результат аккуратной, тщательной работы, пытливого и вдумчивого изучения класса задач и требований к их решению. Эффективные и надежные программы в таких случаях – естественное вознаграждение. Но в ряде случаев природа задач требует свободного выбора одного из вариантов – выбор произвольного элемента множества, вероятности события при отсутствии известных закономерностей, псевдослучайные изменения в игровых обстановках и сценариях, поиск первого подходящего адреса для размещения блока данных в памяти, лингвистический анализ при переводе документации и художественных текстов и т.д. При отсутствии предпочтений все допустимые варианты равноправны, и технология их отладки и обработки должна обеспечивать формально равные шансы вычисления таких вариантов (похожая проблема характерна для организации обслуживания в сетях и выполнения заданий операционными системами. Все узлы и задания сети должны быть потенциально достижимы, если нет формального запрета на оперирование ими).

11.4. Функциональная модель ЛП

Первые реализации ЛП были выполнены на языке Lisp, поэтому основные механизмы можно рассмотреть как обработку списков.

По смыслу выбор варианта похож на выбор произвольного элемента множества.

{ a | b | c } = э { a, b, c }

Чтобы такое понятие промоделировать обычными средствами, нужны дополнительные примитивы.

Например, при определении функции, выбирающей произвольный элемент списка, в какой-то момент L становится пустым списком, и его разбор оказывается невозможным, тогда действует вариант ESC.

Чтобы определить выбор произвольного элемента из списка `L`, можно представить рекурсивное выражение вида:

(любой L) = { (CAR L) | (любой (CDR L)) }

По смыслу выбор варианта похож на выбор произвольного элемента множества.

{ a | b | c } = э { a, b, c }

Чтобы такое понятие промоделировать обычными средствами, нужны дополнительные примитивы с меньшей степенью организованности, чем вектора или множества.

Уточненная таким образом схема выбора произвольного элемента списка можно представить формулой вида:

Определение	*Примечание*
(любой L) = { (car L) | (любой (cdr L)) | ESC }	Множество из первого элемента списка, выбора любого из оставшихся элементов списка и тупика

Пример 27. Тупик как равноправный вариант

Более ясное определение имеет вид

Определение	*Примечание*
(любой L) = { (CAR L) | (любой (CDR L)) | (if (nl L) ESC) }	Выбор тупикового варианта возможен лишь при отсутствии других

Пример 28. В какой-то момент `L` становится пустым списком, и его разбор оказывается невозможным. Тогда действует ESC

Другие построения, характерные для теории множеств: { x | P(X) } – множество элементов, обладающих свойством `P`.

Определение	*Примечание*
(F L) = {(if (P (CAR L)) (CONS (CAR L) (F (CDR L)))) | (if (nl L) ESC) }	Соединяются в список по проверке предиката Тупик

Пример 29. Выбор элементов списка, удовлетворяющих предикату

Определение, данное в этом примере, недостаточно, т.к. порождаемые варианты элементов, удовлетворяющих заданному свойству, существуют в

разные моменты времени и могут не существовать одновременно. Чтобы иметь все варианты одновременно, требуется еще один примитив `ALL`, обеспечивающий накопление всех реально осуществимых вариантов.

Определение	*Примечание*
(F L) = (ALL {(if (P (CAR L)) (CONS (CAR L) (F (CDR L)))) | (if (nl L) ESC) })	Специальная

Пример 30. Множество всех элементов списка, удовлетворяющих предикату

Определение	*Примечание*
(ALL (LAMBDA (x y) { (if (= x y) x) | ESC }) (любой A) (любой B))	Форма выбора элементов пересечения Перебор элементов двух множеств в произвольном порядке

Пример 31. Пересечение множеств A и B

11.5. Логические связки

Логика МакКарти(компьютерная)

Определение	*Примечание*
(if (not a) NIL b)	`a & b`

Пример 32. `b` вычисляется лишь при истинном a, что результативно, но не всегда соответствует интуитивным ожиданиям (логика, предложенная в свое время Мак-Карти, позволяет добиться высокой эффективности). Математически более надежны варианты, исключающие зависимость от порядка перебора:

Определение	*Примечание*
(ALL(LAMBDA x { (if (not x) NIL) | ESC }) {a | b})	Если a и b оба истины, то получается ESC

Пример 33. Такое значение отличается от NIL тем, что работает как истина

Аналогичная проблема возникает при построении ветвлений:

Определение	*Примечание*
((LAMBDA L {(COND ((eval(caar L)AL) (eval(cadr L)AL))) | ESC }) (любой ((p1 e1) (p2 e2) ...)))	Снятие зависимости от порядка записи

Пример 34. (cond (p1 e1) (p2 e2) ...)

Поддержка вариантов, каждый из которых может понадобиться при построении окончательного результата, находит практическое применение при организации высокопроизводительных вычислений. Например, мультиоперации можно организовать с исключением зависимости от порядка отдельных операций в равносильных формулах:

Определение	*Примечание*
((LAMBDA (x y z) {(if (< (+ x y) K) (+ (+ x y) z)) | ESC}) {(a b c) | (b c a) | (c a b)})	Обеспечение максимальной вычислимости

Пример 35. a+b+c = (a+b)+c = a+(b+c) = (a+c)+b – профилактика переполнения

11.6. Реализация недетерминированных моделей

Необходимая для такого стиля работы инструментальная поддержка обеспечивается в GNU Clisp механизмом обработки событий *throw-catch*, для которого следует задать примерно такое взаимодействие:

Определение	*Примечание*
(DEFUN vars (xl)(catch 'ESC (COND ((null xl)(escape)) ((CAR xl) (CONS (CAR xl)(vars (CDR xl))))))) (DEFUN escape () (throw 'ESC NIL))	перебор вариантов до первого тупика vars not NIL сигнал о попадании в тупик
(print(vars ())) (print(vars '(a))) (print(vars '(a b c))) (print(vars (list 'a 'b (vars ()) 'c)))	Демонстрация

Пример 36. Механизм обработки событий throw-catch как модель недетерминизма вариантов

Следует отметить неисчерпаемый ряд задач, при решении которых удобно используются недетерминированные модели:

1. Обоснование упорядочений в традиционных алгоритмах – выделяется доалгоритмический уровень, на котором просто анализируются таблицы возможных решений и постепенно вырабатываются комплекты упорядочивающих условий и предикатов.
2. Переформулировка задач и переопределение алгоритмов с целью исключения необоснованных упорядочений – одна из типовых задач оптимизации, особенно при переходе от обычных программ к параллельным. Приходится выяснять допустимость независимого исполнения всех ветвей и управляющих их выбором предикатов.
3. Обобщение идеи абстрактных машин с целью теоретического исследования, экспериментального моделирования и прогнозирования недетерминированных процессов на суперкомпьютерах и многопроцессорных комплексах (многопроцессорная машина Тьюринга и т.п.).
4. Конструирование учебно-игровых программ и экспериментальных макетов, в которых скорость реализации важнее, чем производительность.
5. Описание и реализация недетерминизма в языках сверхвысокого уровня, таких как Planner, *Setl*, Sisal, Id, Haskell и др.
6. Недетерминированные определения разных математических функций и организация их обработки с учетом традиции понимания формул математиками.
7. Моделирование трудно формализуемых низкоуровневых эффектов, возникающих на стыке технических новинок и их массового применения как в научных исследованиях, так и в общедоступных приборах.
8. Обработка и исследование естественно языковых конструкций, речевого поведения, культурных и творческих стереотипов, социально-психологических аспектов и т.п.
9. Организация и разработка распределенных вычислений, измерений, Grid-технологий, развитие интероперабельных и телекоммуникационных систем и т.п.

За историю ЛП выделился ряд специфических линий:

– абдуктивное логическое программирование;

- металогическое программирование;
- логическое программирование в ограничениях;
- параллельное логическое программирование (FGCS – японский проект 5-го поколения компьютерных систем);
- индуктивное логическое программирование;
- линейное логическое программирование;
- объектно-ориентированное логическое программирование;
- транзакционное логическое программирование.

11.7. Спецификация

Таблица 13

Парадигматическая характеристика языков логического программирования

Параметр	*Конкретика*
1) эксплуатационная прагматика ЯП	Неопределенные постановки задач. Эмпирическое накопление рецептов, достаточное для решения некоторых практических задач.
2) особенности системы понятий	Недетерминированный выбор вариантов. Клаузы и факты. Логический вывод цели. Предикатные формы в виде сопоставления с образцом
3) перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Логическая форма. Факт, правило, клауза, атом, слово, предикат, выражение, процедура, вариант, тупик, строка, переменная, функция
4) базовые средства	Pure Prolog
5) семантические расширения	Бэктреккинг. Сечение. Числа. Хэш-таблица. Компиляция. Функции высших порядков. SDL-резолюция
6) регистры абстрактной машины	S E C D R S – стек промежуточных результатов. E – стек локальных переменных. C – стек основной программы. D – дамп для защиты и восстановления данных. R – стек для перебора вариантов. Результат – заключение об успехе-неудаче вывода цели
7) категории команд абстрактной машины	Кроме типов команд, характерных для ФП: - выбор варианта; - восстановление контекста при переборе вариантов
8) реализационная прагматика	В дополнение к ФП обработка разностных списков, отслеживание низкого приоритета тупиков и сечений. Вместо произвольного выбора варианта реально варианты перебираются в порядке представления

9) парадигматическая специфика	Расширение класса решаемых задач использованием недетерминированных моделей

12. ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

При переходе к практическому программированию обычно возникают проблемы, связанные с изменением отношения к постановке задачи и оценке методов её решения в процессе подготовки, отладки и эксплуатации программы.

1. Прежде всего, необходимость в целостном решении полной задачи «с нуля» теперь встречается достаточно редко.
2. Как правило, необходимо учитывать, что отчасти задача или похожие задачи уже имеют готовые решения. Их надо найти, изучить и выбрать фрагменты, наиболее подходящие для конструирования практичного решения нужной задачи.
3. Обычно сложные задачи декомпозируются на модули, разрабатываемые отдельно, но функционировать модулям предстоит совместно.
4. Кроме того, снижение трудозатрат на создание программ достигается программированием многократно используемых компонент, условия применения которых заранее не известны.

Решение таких проблем требует углубленного анализа структур данных и логики вычислений, обоснованного выбора методов обработки данных и декомпозиции программы на удобно комплексируемые фрагменты. Объектно-ориентированное программирование (ООП) в настоящее время – самый популярный подход к решению этих проблем для широкого класса задач, не требующих предельно жестких эксплуатационных характеристик.

ООП рассматривает информационный процесс как частичную обработку объектов посредством реагирования на события с помощью методов, выбираемых в зависимости от типа обрабатываемых данных. Центральный момент – структурирование множества частных методов, используемых в программе, в соответствии с иерархией классов объектов, обрабатываемых этими методами, в предположении, что определяемые в программе построения могут локально видоизменяться при сохранении основных общих схем информационного процесса. Это позволяет выполнять модификации программы объявлением новых подклассов и дописыванием методов обра-

ботки объектов отдельных классов без радикальных изменений в ранее отлаженном тексте программы.

Связь методов с классами объектов позволяет вводить одноименные методы над разными классами объектов (полиморфизм), что упрощает логику управления: на уровне текста программы можно не распознавать принадлежность классу – это сделает система программирования. Именно так обычно реализовано сложение и другие операции, одинаково изображаемые для чисел, строк, векторов, множеств и т.п. Основной принцип – выбор метода в зависимости от типа данных.

Сборка программы из автономно развивающихся компонентов опирается на формулировку достигаемой ими цели, понимание которой гарантирует не только корректность полученного результата, но и рациональность его использования. Формулировать цели частей программы – процесс нетривиальный. В его основе лежат весьма различные подходы к классификации понятий. Для демонстрации преимуществ ООП следует рассматривать задачи, решение которых требует более одного шага. Первый шаг – программирование ядра программы или его комплектация из готовых модулей с оформлением иерархии классов объектов, содержащих первичный комплект методов. Второй и последующие шаги – развитие иерархии классов и/или перегрузка методов. Задачи, постановки которых допускают более одного сценария развития, потребуют ещё одного специального шага: определение абстрактных/виртуальных классов в качестве точек варьирования возможного развития постановки задачи.

Таблица 14

Конкретизация понятий в языках ООП на примере языка C++ и системы CLOS

Понятие	ООП: C++ Clos
Атом (Элементарное)	Имя объекта, поля, метода, Скаляр
Структура	Класс с разметкой доступа
Переменная	Имя поля, Имя → Ид (Знач)
Значение	|скаляр|, Объект
Выражение	Объект.Метод
Действие/Операция	Над скалярами, Доступ к полю, Применение метода

Условие/истина	
Функция/ метод/ оператор	Имя → Ид (подпр), Локализованы по иерархи классов
Аргумент	Локальные встеке, Поле объекта
Вызов Фн/подпр	Выбор метода по ТД, принадлежности к классу
Определение Фн/подпр	Пополнение списка методов для класса
Идентификатор	Адрес объекта, поля, переменной

12.1. Общее представление

ООП объединяет в рамках единой методики организации программ классификацию на базе таких понятий как класс объектов, структура данных и тип значений. Тип значений обычно отражает спектр основных низкоуровневых реализационных средств, учет которых обеспечивает эффективность кода программы, получаемого при компиляции. Структура данных обеспечивает конструктивность построений, гарантирует ясный доступ к частям, из которых выстроено данное любой сложности. Класс объектов характеризуется понятным контекстом, в котором предполагается их корректная обработка. Обычно контекст содержит определения структуры объектов и их свойства.

Текст программы одновременно представляет и динамику управления процессами, и схему информационных потоков, порождаемых при исполнении программы. Кроме того, последовательность написания программы и ее модификации по мере уточнения решаемой задачи могут соответствовать логике, существенно отличающейся и от логики процесса выбора системных и реализационных решений, и от логики применения реализованных решений. Обычно программирование скрывает сложность таких деталей управления процессами путем сведения его к общим функциям, преобразующим любые аргументы в определенные результаты. Модификации программы при развитии решаемой задачи нередко осуществляются непосредственно ручной переработкой текста программы и определений входящих в нее функций.

При анализе задач, решаемых в терминах объектов, некоторая деятельность описывается так, что постепенно продумывается все, что можно делать с объектом данного класса. Потом в программе достаточно лишь упоминать методы обработки объекта. Если методов много, то они структурированы по иерархии классов, что позволяет автоматизировать выбор конкретного метода. На каждом уровне иерархии можно немного варьировать

набор методов и структуру объектов. Таким образом, описание программы декомпозируется на интерфейс и реализацию, причем интерфейс скрывает сложность реализации так, что можно обозревать лишь необходимый для использования минимум средств работы с объектом.

Исходная гипотеза при программировании работы с объектами:

Объект не изменен, если на него не было воздействий из программы.

Но реальность зачастую требует понимания и учета более сложных обстоятельств, что может существенно продлить время жизни программы или ее компонентов. В таком случае удобно предоставлять объектам большую свободу, сближающую их с понятием субъекта, описание которого содержит все, что он может делать. Программа может давать ему команды-сообщения и получать ответы-результаты.

Фактически субъектом является суперкласс, объединяющий классы объектов, обрабатываемые одноименными методами, т.е. функциями одного семейства. Так, при организации сложения можно считать, что существует суперкласс «слагаемые», которое умеют складываться с другими слагаемыми. При этом они используют семейство функций – методов, реализующих сложение. В зависимости от полноты семейства результат может быть получен или не получен. Семейство легко пополняется добавлением одноименных функций с новыми комбинациями типов параметров.

Дальнейшее развитие подходов к декомпозиции программ связано с выделением отдельных аспектов и шагов при решении сложных задач. Понятие аспекта связано с различием точек зрения, позволяющим описывать решение всей задачи, но отражать в описании только видимые детали. По мере изменения точек зрения могут проступать новые детали, до тех пор пока дальнейшая детализация не утрачивает смысл, т.е. улучшение трудно заметить, или цена его слишком высока. Так, представление символьной информации в Лиспе выделено в отдельный аспект, независимый от распределения памяти, вычисление значений четко отделено от компиляции программ, понятие связывания имен с их определениями и свойствами не зависит от выбора механизмов реализации контекстов исполнения конструкций программы.

Разработка сложной программы может рассматриваться как последовательность шагов процесса раскрутки программ, оправданным в тех случаях, когда целостное решение задачи не может гарантировать получение приемлемого результата в нужный срок – это влечет за собой непредсказуемо большие трудозатраты.

Удобный подход к организации программ «отдельная работа отдельно программируется и отдельно выполняется» успешно показал себя при раз-

витии операционной системы UNIX как работоспособный принцип декомпозиции программ. Но существуют задачи, например, реализация систем программирования, в которых прямое следование такому принципу может противоречить требованиям к производительности. Возможен компромисс «отдельная работа программируется отдельно, а выполняется взаимосвязано с другими работами», что требует совмещения декомпозиции программ с методами сборки – комплексации или интеграции программ из компонентов. Рассматривая комплексацию как еще одну «отдельную» работу, описываемую, например, в терминах управления процессами, можно констатировать, что эта работа больше определяет требования к уровню квалификации программиста, чем объем программирования. При достаточно объективной типизации данных и процессов, возникающих при декомпозиции и сборке программ определенного класса, строят библиотеки типовых компонентов и разрабатывают компонентные технологии разработки программных продуктов – Corba, COM/DCOM, UML и т.п.. Одна из проблем применения таких компонентов – их обширность.

Таким образом, ООП может отражать эволюцию подходов к организации структур данных на уровне задач и программ их решения, исходя из парадигмы императивно-процедурного программирования. От попыток реализации математически корректных абстрактных типов данных произошел практичный переход к технически простому статическому контролю типов данных при разработке и применении расширяемых программ. Расширение программы выполняется декларативно, а выбор нужного варианта при исполнении функций, обладающих неединственным определением, – в зависимости от типа данных. Введены дополнительные механизмы – инкапсуляция, уточнение типов данных при компиляции и выбор обработчиков данных, управляемый типами данных.

Механизмы ООП обеспечивают наследование свойств по иерархии классов объектов и так называемый «дружественный» доступ к произвольным классам. Расширение программ при объектно-ориентированном подходе к программированию выглядит как простое дописывание новых определений. Библиотеки типов данных и методов их обработки легко вписываются в более общие системы. Спецификация интерфейсов в принципе может быть сопровождена верификацией реализации компонент. Возможна факторизация программ на компоненты и рефакторизация программных компонент в стиле экстремального программирования.

При реализации экспериментальных языков и систем программирования часто применяется методика раскрутки – минимизация трудозатрат, основанная на учете формальной избыточности средств языков програм-

мирования. Выделяется небольшое ядро, на основе которого методично программируется все остальное. Каждый шаг реализации по схеме раскрутки должен обеспечивать

- уменьшение трудоемкости последующих шагов,
- отладку прототипов сложных компонентов,
- подготовку демонстрационного материала.

Выбор конкретных шагов можно соотнести с декомпозицией определения языка программирования на синтаксические и семантические, функциональные и машинно-ориентированные, языково-ориентированные и системные аспекты. При такой декомпозиции можно на первых шагах как бы «снять» синтаксическое и семантическое разнообразие языка, как имеющее чисто технический характер. Именно в этом смысл выделения элементарного ядра ЯП (Pure Lisp, Pure Prolog, структурное программирование и др.). Такая методика может быть успешна при освоении любого класса, информацию о котором можно представить в виде частично формализуемых текстовых и графовых форм.

Дальнейшие шаги раскрутки можно упорядочить по актуальности реализации компонентов, обеспечивающих положительную оценку системы пользователем. Это позволяет развить представление о принципах декомпозиции программ более созвучно ООП: «отдельная работа обнаруживается независимо от остальных работ». Наиболее устойчивая и значимая классификация работ по реализации системы программирования может быть установлена как обеспечение механизмов надежного функционирования информационных систем – «отдельная работа – это отдельное средство повышения надежности программирования». Ряд таких средств можно выделить в любом языке программирования: вычисления, статическое и динамическое управление процессами, логика выбора хода обработки информации, дисциплина именования памяти и доступа к расположенным в ней данным, правила укрупненных воздействий на блоки данных и иерархию процессов, диагностика и обработка событий – перечень открытый.

ООП структурирует множество частных методов, используемых в программе, в соответствии с иерархией классов объектов, обрабатываемых этими методами, реализуемыми с помощью функций и процедур, в предположении, что определяемые в программе построения могут локально видоизменяться при сохранении основных общих схем информационной обработки. Это позволяет выполнять модификации объявлением новых подклассов и дописыванием методов обработки объектов отдельных классов без радикальных изменений в ранее отлаженном тексте программы.

12.2. Абстрактная машина

АС включает в себя аналоги ИП, ФП и ЛП с добавлением выбора метода в зависимости от класса объектов.

АМ для языков ООП можно базировать на АМ для процедурно-императивных языков стандартного прикладного программирования.

Абстрактная машина для ОО-языка по структуре похожа на объединение машин для ФП и ИОП.

AMO = < RO, SCO>, где SCO= <S, E, C, D, M>

S – стек операндов и результатов вычислений.
E – значения локальных переменных при вызовах функций.
C – текущий стек программы.
D – дамп, обеспечивающий восстановление контекста программы при выходе из метода.
M – общая память, хранящая константы, методы и объекты с их сигнатурами.

<[cm], ts> – код метода, таблица символов
<tsi (@cm, v), ret (#data)> – таблица метода, v – символьные переменные, сигнатура возврата

Представление класса содержит сведения о числе констант и информацию о них, флаги доступа к полям объектов класса, ссылки на объект и суперкласс, число полей и информацию о них, число методов и информацию о них.

Сигнатура представляется символьной характеристикой полей объекта или параметров метода.

Значения реализованы как структура данных с тегами, задающими тип элемента данных.

Классы, поля и методы не являются значениями и хранятся без тэга.

Таблица 15

Типичные команды виртуальной машины для языка ООП

Команда	*Пояснение*
NOP	Ничего не делает
RETURN	Возврат из функции
PUT	Установка поля в объекте
GET	Доступ к полю в объекте
INVOKE	Вызов метода
CHECK	Проверка соответствия типа объекта

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

Таблица 16

Дополнительная спецификация команд виртуальной машины для языка ООП. t – логическое значение

Исходной состояние	*Результат*
s e (NOP . c) d m	s e c d m
s e (RETURN . c) d m	s e c d m
(f v . s) e (PUT i . c) d m	s e c d (m | m[f,i] = v)
(f . s) e (GET i . c) d m	(m[f,i] . s) e c d m
((a1 a2 ... aK) f .s) e (INVOKE sig . c) d m	s ((a1 a2 ... aK) . e) (f[sig] . c) d m
(Obj . s) e (CHECK type . c) d m	(t . s) e c d m

Главный путь к снижению трудоёмкости программирования лежит через упрощение процесса отладки, что зависит от искусства декомпозиции постановок задач и программ их решения на такие комплекты компонент, часть которых можно найти в библиотеках готовых модулей, а часть можно при программировании довести до уровня многократно используемых компонент.

Современное состояние имеющихся технических решений в данной области характеризуется доминированием компонентных технологий, ориентированных на ООП, обеспечивающих классификацию конструктива на уровне понятий пользователя и его отображение на уровень целевых архитектур, представимых в терминах абстрактных машин. При таком подходе не получают полного выражения функциональная декомпозиция и системные решения промежуточного уровня, что отчасти компенсируется развитием аспектно-ориентированного подхода, выглядящего как метанадстройка над ООП [48]. Отдельный ряд трудностей вызывают приаппа-

ратные оптимизации, требующие более тонкой детализации, ниже традиционного уровня абстрактных машин.

Более реальна перспектива снижения трудоёмкости и повышения надёжности практического программирования повышением кратности использования библиотечных модулей в рамках многоязыкового программирования на базе систем программирования, создаваемых из общего, а потому более отлаженного конструктива. Важный импульс создан проектом **.**Net, оживившим интерес к развитию технологий реализации ЯП, а также к разработке новых ЯП. Цель проекта – получение преимуществ ООП в области реализации СП. Концептуально идеи **.**Net-технологии весьма близки к VDM – Венской методике определения семантики языков программирования и научным теоретическим и экспериментальным исследованиям, выполненным в Новосибирске под руководством академика А. П. Ершова [18, 19], но проработка проблем декомпозиции программ не дотягивает до предложенного в 70-е годы А. Л. Фуксманом расслоенного программирования [44], подобного аспектно ориентированному программированию.

12.3. C++

С концепцией ООП связано представление о возможности сокрытия информации, наследования определений по иерархии классов и полиморфизма реализации операций и функций. Переход к ООП в языке C++ привел к пересмотру некоторых решений на уровне языка и компилятора. Рассмотрим особенности C++ как наиболее популярного языка ООП, в котором достижима схема, обобщающая комплекс решений задачи в виде ациклического графа с возможными горизонтальными связями. Такие решения направлены на программирование ряда версий решения задачи без отмены ранее отлаженных решений, но с формированием новых областей видимости, в которых устаревшая часть программы может быть просто оттеснена.

- компилируемая программа на C++ представляет собой иерархию областей видимости определений элементов классов, доступ к которым регламентирован;
- компиляция методов, конструкторов, функций и перегруженных операций управляется форматом списка фактических параметров, что привело к более жёстким правилам объявления типов данных и ограничивает свободу конкретизации списка параметров при вызове функций;

– возникают рекомендательные средства повышать эффективность кодирования вызовов функций указанием на inline-включение;
– появляется уровень программирования шаблонов для представления общих схем обработки контейнерных типов разнотипных данных.

Программа	*Пояснение*

//HELLO.CPP
#include <iostream.h>
Void main ()
{ cont << ”\nHello, World!\n” ;
}
	Комментарий с именем файла программы
	Препроцессор с вызовом библиотеки
	Головная функция
	Вывод приветствия на стандартное устройство

Пример 37. Программа на языке C++

Практический выигрыш от ООП можно показать на технике применения средств вывода данных.

Фрагмент	*Пояснение*
printf (“x = %d, y = % s”, x, y);	Вывод целого и строки. «x» должен быть целым, а «y» – строкой

Пример 38. Вывод по библиотеке функций *stdio.h*

Программист должен знать, что первый параметр задает формат вывода и отследить согласование его с числом и типами выводимых данных и обозначениями форматов для функции *printf.*

Фрагмент	*Пояснение*
cout << “x = ” << x << “, y = ” << y ;	Поток вывода управляется фактическим типом данных.

Пример 39. Вывод по библиотеке классов *iostream.h*

Программисту достаточно перечислить элементы вывода в естественном порядке. Библиотека классов *iostream.h* содержит перегрузку операции «<<» для всех основных типов данных, что позволяет компилятору выбрать нужный шаблон кода программы, и программист освобожден от необходимости представлять в программе сведения о формате выводимых данных.

Учитывая примеры описания и определения классов, можно сделать вывод, что внешние изменения в тексте программы на C++ в сравнении с текстом на Си выглядят как появление ряда новых спецификаторов, с по-

мощью которых как бы задается разметка программы, на области видимости, связанные с иерархией классов и дисциплиной доступа к элементам структурированных объектов класса. В результате вместо анализа последовательностей изменения состояний памяти по всей программе достаточно проанализировать изменения в пределах синтаксически выделенных путей по иерархии наследования методов объектов.

Доступность вложенного класса ограничивается областью видимости лексически объемлющего класса.

Если у класса есть конструктор, он вызывается всякий раз при создании объекта этого класса. Если у класса есть деструктор, он вызывается всякий раз, когда уничтожается объект этого класса.

Чтобы можно было описать массив объектов класса с конструктором, этот класс должен иметь стандартный конструктор, вызываемый без параметров. В описании массива объектов не предусмотрено возможности указать параметры для конструктора. Когда уничтожается массив, деструктор должен вызываться для каждого элемента массива.

Производный класс наследует базовый класс, он больше своего базового класса в том смысле, что в нем содержится больше данных, и определено больше функций. Производный класс сам, в свою очередь, может быть базовым классом: такое множество связанных между собой классов обычно называют иерархией классов. Обычно она представляется деревом, но бывают иерархии с более общей структурой в виде ориентированного графа. У класса может быть несколько прямых базовых классов. Возможность иметь более одного базового класса влечет за собой возможность неоднократного вхождения класса как базового.

С помощью виртуальных функций можно иметь разные версии в разных производных классах, а выбор нужной версии при вызове – это задача транслятора. Тип функции указывается в базовом классе и не может быть переопределен в производном классе. Класс, в котором есть виртуальные функции, называется абстрактным. Абстрактный класс можно использовать только в качестве базового для другого класса.

Члены класса создаются в порядке их описания, а уничтожаются они в обратном порядке. Член класса может быть частным (private), защищенным (protected) или общим (public):

Фрагмент	*Пояснение*
class complex { double re, im; public:	Объявление класса объектов с закрытыми полями. и общедоступными

complex(double r, double i) { re=r; im=i; } **friend complex operator+(complex, complex);** friend complex operator*(complex, complex); };	конструкторами и перегрузкой операций.

Пример 40. Перегрузка арифметических операций для их использования в выражениях над комплексными числами

Фрагмент	*Пояснение*

void f()
{
complex a = complex(1,3.1);
complex b = complex(1.2,2);
complex c = b;
 	
a = b+c;
b = b+c*a;
c = a*b+complex(1,2);
}
	Объявление функции, выполняющей процедуру
	
	конструирования 3-х комплексных чисел и их инициирования,
	
	и обработки чисел с помощью перегруженных операций.

Пример 41. Интерпретация этих операций задана определениями функций с именами operator+ и operator*

Если `b` и `c` имеют тип `complex`, то `b+c` означает (по определению) `operator+(b,c)`. Сохраняются обычные приоритеты операций, поэтому второе выражение выполняется как `b=b+(c*a)`, а не как `b=(b+c)*a`. При перегрузке операций нельзя изменить их приоритеты, равно как и синтаксические правила для выражений.

Для операций преобразования ТД выбран подход, при котором проверка соответствия ТД является строго восходящим процессом, когда в каждый момент рассматривается только одна операция с операндами, типы которых уже прошли проверку.

Вызов функции, т.е. конструкцию выражение(список-выражений), можно рассматривать как бинарную операцию, в которой выражение является левым операндом, а список-выражений – правым. Операцию вызова можно перегружать, как и другие операции.

Одним из самых полезных видов классов является контейнерный класс, т.е. такой класс, который хранит объекты каких-то других типов. Списки, массивы, ассоциативные массивы и множества – все это контейнерные классы.

Фрагмент	*Пояснение*

template<class T>
 	
class stack {
  T* v;
  T* p;
  int sz;
 	
public:
  stack(int s) { v = p = new T[sz=s]; }
  ~stack() { delete[] v; }
 	
  void push(T a) { *p++ = a; }
  T pop() { return *--p; }
 	
  int size() const { return p-v; }
};
	Объявление параметризованного шаблона.
	
	Для создания классов стеков в зависимости от задаваемого типа хранимых элементов при известном объёме стека.
	
	Общедоступны:
	конструкторы и деструкторы стека,
	
	функции работы со стеком

Пример 42. Шаблон типа для класса. Стек, содержащий элементы произвольного типа

Префикс `template<class T>` указывает, что описывается шаблон типа с параметром `T`, обозначающим тип, и что это обозначение будет использоваться в последующем описании. После того, как идентификатор `T` указан в префиксе, его можно использовать как любое другое имя типа.

Область видимости `T` продолжается до конца описания, начавшегося префиксом `template<class T>`.

Имя шаблонного класса, за которым следует тип, заключенный в угловые скобки <>, является именем класса (определяемым шаблоном типа), и его можно использовать как все имена класса.

Поскольку все функции-члены класса `stack` являются подстановками, то и в этом примере транслятор создает вызовы функций только для размещения в свободной памяти и освобождения.

Функции в шаблоне типа могут и не быть подстановками.

В программе может быть только одно определение функции-члена класса и только одно определение шаблона типа для функции-члена шаблонного класса. Если требуется определение функции-члена шаблонного класса для конкретного типа, то задача системы программирования найти шаблон типа для этой функции-члена и создать нужную версию функции. В общем случае система программирования может рассчитывать на указания от программиста, которые помогут найти нужный шаблон типа.

Возможна передача операций как параметров функций

Рассмотренные средства представления иерархии классов объектов с возможностью множественного наследования, использования программируемых и встроенных конструкторов и деструкторов объектов, создания массивов объектов класса с контролем доступа к элементам, перегрузки операций и задания виртуальных функций, а также, операций преобразования ТД и шаблонов типа для обработки контейнерных структур данных суммарно образуют достаточно богатый арсенал для поддержки процесса практичной разработки программ при решении расширяющейся задачи, совмещённого с процессом декомпозиции программы на многократно используемые компоненты разного уровня абстрагирования от конкретики решаемой задачи и специфики системных реализационных решений.

При организации наследования в отличие от обобщенных функций работает модель обмена сообщениями*:*

- объекты обладают свойствами,
- посылают сообщения*,*
- наследуют свойства *и* методы от предков.

При переходе от обычного стандартного программирования с ООП связывают радикальное изменение способа организации программ. Это изменение произошло под давлением роста мощности оборудования. ООП взламывает традиционное программирование по многим направлениям. Вместо создания отдельной программы, оперирующей массой данных, приходится разбираться с данными, которые сами обладают поведением, а программа сводится к простому взаимодействию новой категории данных – «объекты».

12.4. Функциональные модели ООП

Рассмотрим результаты анализа схемы реализации принципов ООП в рамках функционального программирования на базе ряда структур данных на примере простой модели ОО-языка, встраиваемого в Лисп. Реализация методов обработки объектов заданного класса сводится к отдельной категории функций, вызов которых управляется анализом принадлежности аргумента классу

Чтобы сравнить дистанцию ООП с ФП П. Грем (Paul Graham) в описании стандарта языка Common Lisp предлагает рассмотреть модель встроенного в Lisp объектно-ориентированного языка (ОО-язык), обеспечивающего основы ООП [55]. Встраивание ОО-языка показывает характерное

применение ФП – моделирование разных стилей программирования (начиная со стандартного программирования в виде prog-формы, предложенной Дж. Маккарти). В языке Lisp есть разные способы размещать коллекции свойств. Один из них – представлять объекты как хэш-таблицы и размещать свойства как входы в нее. В [55] приведен пример (8 строк) реализации ООП на базе хэш-таблиц. Фактически наследование обеспечивает единственная особенность языка Lisp: все это работает благодаря реализации рекурсивной версии GETHASH. Впрочем, Lisp по своей природе изначально был ОО-языком. Определение методов может достичь предельной гибкости благодаря возможности генерировать определения функциональных объектов с помощью DEFMACRO [55] или функцией категории FSUBR [58].

Реализация ООП с помощью хэш-таблиц обладает слегка парадоксальной окраской: гибкость у нее больше, чем надо, и за большую цену, чем можно позволить. Уравновесить это может подобная реализация на базе простых векторов. Этот переход показывает, как функциональное программирование дает новое качество «на лету». В опорной реализации фактически не было реализационного разделения объектов на экземпляры и классы. Экземпляр – это был просто класс с одним-единственным предком. При переходе к векторной реализации разделение на классы и экземпляры становится реальным. В ней становится невозможным превращать экземпляры в классы простым изменением свойства.

Более прозрачная модель ООП получается на базе обычных списков, заодно иллюстрирующая глубинное родство ФП и ООП реализована в системе CLOS.

Показанный в [55] пример работает по первому аргументу (выбор подходящего метода рассчитан на то, что достаточно разобраться с одним аргументом), `CLOS` делает это на всех аргументах, причем с рядом вспомогательных средств, обеспечивающих гибкий перебор методов и анализ классов объектов.

Классы и экземпляры объектов

```
(defclass ob () (f1 f2 ...))
```

Это означает, что каждое вхождение объекта будет иметь поля-слоты `f1 f2 ...` (Слот – это поле записи или списка свойств). Чтобы сделать представителя класса, мы вызываем общую функцию:

```
(SETF c (make-instance 'ob))
```

Чтобы задать значение поля, используем специальную функцию:

(SETF (slot-value c) 1223)

До этого значения полей были не определены.

Простейшее определение слота – это его имя. Но в общем случае слот может содержать список *свойств*. Внешне свойства слота специфицируются как ключевые параметры функции. Это позволяет задавать начальные значения. Можно объявить слот совместно используемым.

:allocation :class

Изменение такого слота будет доступно всем экземплярам объектов класса. Можно задать тип элементов, заполняющих слот, и сопроводить их строками, выполняющими роль документации.

Нет необходимости все новые слоты создавать в каждом классе, поскольку можно наследовать их из суперклассов.

Определение	*Пояснение*
(defclass expr () ((type :accessor td) (sd :accessor ft)) (:documentation "C-expression"))	Суперкласс для всех структур Lisp-выражений. Заданы ключи доступа к общим полям объектов: тип и операнд
(defclass un (expr) ((type :accessor td) (sd :accessor ft)) (:documentation "quote car *other *adr"))	Класс для унарных форм. Доступ можно унаследовать от суперкласса, а здесь не дублировать
(defclass bin (expr) ((type :accessor td) (sd :accessor ft) (sdd :accessor sd)) (:documentation "cons + lambda let"))	Класс бинарных форм Третье поле для второго операнда
(defclass trio (expr) ((type :accessor td) (sd :accessor ft) (sdd :accessor sd) (sddd :accessor td)) (:documentation "if label"))	Если взять суперкласс (bin), то можно не объявлять первые 3 поля Четвёртое поля для третьего операнда
(defmethod texrp ((x expr) (nt atom)) (SETF (slot-value x 'type) nt) (SETF (td x) nt) ;;--;; variant (:documentation "объявляем тип выражения"))	Метод представления выражений для компиляции

(defmethod spread ((hd (eql 'QUOTE)) (tl expr)) (let ((x (make-instance 'un))) (SETF (ft x) (car tl)) (SETF (td x) hd)) (:documentation "распаковка выражения"))	Метод разбора констант и выражений с унарными операциями.
(defmethod compl ((hd (eql 'QUOTE)) (tl expr)) (list 'LDC tl) (:documentation "сборка кода"))	Метод компиляции констант
(defmethod compl ((hd (eql 'CAR)) (tl expr) N) (append (compl(ft tl) N) '(CAR)) (:documentation "сборка кода"))	Метод компиляции выражений с унарными операциями
(defmethod spread ((hd (eql 'CONS)) (tl expr)) (let ((x (make-instance 'bin))) (SETF (ft x) (CAR tl)) (SETF (sd x) (cadr tl)) (SETF (td x) hd)) (:documentation "распаковка выражения"))	Метод разбора выражений с бинарными операциями
(defmethod compl ((hd (eql 'CONS)) (tl bin) N) (append (compl(sd tl) N) (compl(ft tl) N) '(CONS)) (:documentation "сборка кода"))	Метод компиляции выражений с бинарными операциями с прямым порядком компиляции операндов
(defmethod compl ((hd (eql '+)) (tl bin) N) (append (compl(ft tl) N) (compl(sd tl) N) '(ADD)) (:documentation "сборка кода"))	Метод компиляции выражений с бинарными операциями с обратным порядком компиляции операндов
(defmethod spread ((hd (eql 'IF)) (tl expr)) (let ((x (make-instance 'trio))) (SETF (ft x) (CAR tl)) (SETF (sd x) (cadr tl)) (SETF (td x) (caddr tl)) (SETF (td x) hd)) (:documentation "распаковка выражения"))	Метод разбора выражений с триадными операциями
(defmethod compl ((hd (eql 'IF)) (tl expr) N) (let ((then (list (compl(sd tl)N) '(JOIN))) (else (list (compl(td tl)N) '(JOIN))))	Метод компиляции выражений с триадными операциями

(append (compl(ft tl)N) (list 'SEL then else)))(:documentation "сборка кода"))	
(defmethod parh ((x expt)) (let (ftx (ft x)) (COND ((ATOM ftx) (spread 'ADR ftx)) ((member (CAR ftx) '(QUOTE CAR CONS + IF LAMBDA LA- BEL LET)) (spread (CAR ftx) (CDR ftx)) (T (spread 'OTHER ftx))))(:documentation "шаг разбора"))	Метод разбора произвольных выражений

Пример 43. ОО-определение Лисп-компилятора

CLOS, естественно, использует модель обобщенных функций, но мы написали независимую модель, используя более старые представления, тем самым показав, что концептуально ООП – это не более чем перефразировка идей Лиспа. ООП – это одна из вещей, которую Лисп изначально умеет делать. Для функционального стиля программирования в переходе к ООП нет ничего революционного. Такой переход практически не расширяет пространство предстоящих решений. Это просто небольшая конкретизация механизмов перебора ветвей функциональных объектов. Императивному стилю ООП компенсирует избыточную целостность представления отлаживаемых программ, смягчает жесткость зависимости компонентов программы от потока информационных процессов.

Более интересный вопрос, что же нам еще может дать функциональный стиль и лисповская традиция реализации систем программирования?

Другая модель ООП, полученная на базе обычных списков свойств (атрибутов), также иллюстрирует глубинное родство ФП и ООП [10]. Нужно лишь уточнить определение Lisp-интерпретатора, чтобы методы рассматривались как особая категория функций, обрабатываемая специальным образом.

12.5. Спецификация

Таблица 17

Парадигматическая характеристика парадигмы ООП

Параметр	*Конкретика*
1. Эксплуатационная прагматика ЯП	Практичное программирование, нацеленное на разумный компромисс в пространстве противоречивых критериев с приоритетом критериям сферы приложения программ
2. Особенности системы понятий	Провозглашены принципы наследования методов, защиты информации от бесконтрольного доступа, полиморфизма функций и методов. Статическая иерархия классов объектов. Объект – это адрес на блок в памяти, содержащий ссылку на допустимые методы его обработки
3. Перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Класс, объект, поле объекта, метод, скаляр, ключевые слова, константы, переменные, доступ к полю, вызов метода, операция, выражение, функции, процедуры, аргументы, указатель, ссылка, перегрузка операций
4. Базовые средства	В дополнение к средствам ИОП и ФП средства вызова метода и учета дисциплины доступа к полям объекта
5. Семантические расширения	Разнообразие скалярных типов данных. Перегрузка операций, функций и методов. Открытые подстановки и шаблоны. Абстрактные и виртуальные кострукции. Сторонний доступ. Умолчания. Асинхронность
6. Регистры абстрактной машины	S E C D M S – стек промежуточных результатов E – стек локальных переменных C – текущая программа D – дамп для восстановления контекста при рекурсии.

	M – общая память хранимых объектов. Результат рассредоточен по именованным состояниям памяти
7. Категории команд абстрактной машины	Загрузка в стек Сохранение значений Манипулирование стеком Арифметические и логические операции Передачи управления Выбор определения метода, функции, операции Вызов метода Возврат из метода Обработка исключений Работа под монитором (параллелизм)
8. Реализационная прагматика	Сочетание статического представления методов с динамикой размещения объектов, включая автоматизацию повторного использования памяти. Представление сигнатуры для динамического выбора конкретного метода обработки объект
9. Парадигматическая специфика	Процесс программирования сводится к последовательности расширяемых по мере целесообразности программ. Использование предметной типизации классов объектов как ведущего параметра выбора конкретной техники обработки данных

13. МУЛЬТИПАРАДИГМАТИЧЕСКИЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Достаточно чётко границы между областями практичного проявления разных парадигм программирования можно выразить типичными формами постановок задач на программирование.

Стандартное императивно-процедурное программирование: «Существует алгоритм решения актуальной задачи. Необходимо подготовить программу реализации алгоритма с практичными пространственно-временными характеристиками на доступном оборудовании».

Функциональное программирование: «Известна предметная область. Следует выбрать символьное представление данных для этой области и отладить систему универсальных функций, пригодных для использования в

разных программах обработки данных при решении актуальных задач из этой области».

Логическое программирование: «Дана коллекция фактов и отношений, показывающая актуальную задачу. Надо привести эту коллекцию к форме, достаточной для получения ответов на практичные запросы относительно данной задачи».

Объектно-ориентированное программирование: «Доступна иерархия классов объектов, поддерживающая работоспособные методы решения ряда задач некоторой предметной области. Нужно без лишних трудозатрат уточнить эту иерархию, чтобы приспособить её к решению новых востребованных задач этой области, её расширения или ей подобной».

Практические задачи нередко включают такие формулировки в качестве подзадач, что приводит при создании ЯП и разработке СП к поддержке разных парадигм одновременно. Так, например, при целенаправленной разработке моно-парадигматического языка Haskell, позиционируемого как чисто функциональный язык, авторы пришли к концепции монад, позволяющей привлекать механизмы других парадигм. Потребность в поддержке парадигм, отсутствующих в реализуемом ЯП, может встраиваться в СП в виде библиотечных процедур, ассемблерных вставок, макро-генераторов или организации выхода на уровень операционной системы.

Таблица 18

Сравнение прагматических особенностей ПП

ПП	*Иерархическая декомпозиция программы*	*Укрупнение конструкций*	*Память и результат*	*Реализационная прагматика*
ИОП	Структуры данных. Динамика вызовов процедур. Области видимости идентификаторов	Структуры и типы данных. Функции над значениями и указателями. Процедуры изменения состояний памяти	S E C M Стеки промежуточных результатов и локальных переменных. Вектор общей памяти. Результат рассредоточен по именованным блокам памяти M	Раздельное хранение программы и данных. «Забывание» идентификаторов и типов данных на период исполнения. Распределение памяти по принципу соседства

ФП	Структуры данных. Динамика вызовов функций. Статика и динамика определений. Сохраняемые состояния системы программирования	Функции, отображающие аргументы в результаты произвольной сложности. Символьные выражения	S E C D Стеки для операндов и результатов, локальных определений, программы и восстановления состояний памяти. Результат в стеке S	Тегированные указатели. Динамическое распределение памяти с автоматизацией повторного использования. Программа имеет представление в виде данных
ЛП	Структуры данных. Динамика вызовов процедур	Шаблоны как лаконичная форма представления предикатов. Клаузы частичного определения логики вывода цели	S E C D R Стеки как для ФП с добавлением регистра R для хранения вариантов. Результат – оценка выводимости цели	Разностные списки. Перебор вариантов в порядке представления. Минимальность приоритета тупиковых вариантов.
ООП	Распределение методов обработки объектов по иерархии классов. Наследование дисциплины доступа к полям объекта. Области видимости методов	Классы объектов. Методы обработки объектов определённого класса. Структуры данных. Полиморфные определения	S E C D M Стеки как для ФП с добавлением вектора M для хранения полиморфных определений методов	Представление сигнатуры методов и их вызовов на период исполнения. Выбор метода по числу аргументов и типам данных

Заметные различия ПП видны и на уровне базовых семантических систем ЯП.

Таблица 19

Специфика базовых семантических систем
в разных парадигмах программирования на ЯВУ

	Выч	Пам	Упр	Структ
ИОП	Скаляры = слова СД-ТД предвычисления	Стат.распред := Области видимости - дин вызовов - статика вложенности текста - глоб Описания Категории имен составные имена exprt-import	Goto If Call Case Select Switch For While Until Do Trap Catch Ситуации ОС	Array Record Union Pointer
ФП	Длинные скаляры Списки Программы Пред и пост lazy	GC – FS Без описаний До исчерпания Дин вызовов – БД (атомы) Списки свойств Let Declare	Quote-eval Lazy Cond Lambda Defun Evalquote Ловушки Мультизначения Параллелизм Необязательные параметры	Списки Файлы Строки Потоки Модели массивов, таблиц, вариантов
ЛП	Варианты с возвратами Шаблоны в строке Подстановка	Накопление рецептов	Обход лабиринта Параллелизм – обход графа в ширину	Иерархия принятия решений
ООП	Обработка полей записи	Разметка доступа Дружественный доступ	Выбор метода по ТД или по ситуации Привязка к интерфейсу	Иерархия классов

13.1. Долгоживущие ЯП

Следует обратить внимание, что наиболее успешные, долго живущие языки и системы программирования являются мультипарадигматическими. Fortran включает в себя [21]:

- средства представления выражений с функциями;
- арифметические функции;
- неявные циклы форматного ввода-вывода векторов и массивов;
- вычисляемые передачи управления;
- сопрограммы – многовходовые подпрограммы.

Lisp 1.5 поддерживает работу [58]:

- с таблицей свойств атомов, фактически поддерживающей статическую память с побочным эффектом;
- императивно-процедурное программирование в форме Prog.
- глобальные определения атомов в терминах свойств атомов.
- псевдо-атомы для работы со скалярами разных типов и форматов.
- мультиоперации над числами и строками.
- псевдо-функции для взаимодействия с операционной системой.
- расширение системы программирования определениями на входном языке и средствами ассемблера.

Статически типизированные ЯП Pascal, Си и др. поддерживают вариантные структуры данных, позволяющие в динамике обойти контроль ТД.

Такие комментарии можно привести для всех ЯП, подтвердивших длительностью своего существования разумность предложенных в них обобщенных решений.

13.2. Новое поколение

В рамках проекта .Net появились новые ЯП F# и C#.

Изначально императивно-процедурный Си и язык ООП C++ при переходе к C# обогащаются рядом новых возможностей:

- выражения могут содержать безымянные функции, представленные как структуры данных;
- поддержаны динамические типы данных – стеки, очереди, списки, деревья;
- доступны библиотеки классов, библиотеки элементов управления и библиотеки Web-элементов управления;
- формируется код программы, способный учитывать особенности текущей платформы;

– задача сборки мусора в памяти снята не только с программистов, но и с разработчиков трансляторов; она решается в нужное время и в нужном месте – исполнительной средой, ответственной за выполнение вычислений;
– каждый тип, помимо полей, методов и свойств, может содержать и события;
– при компиляции программы создаётся манифест, полностью описывающий её сборку.

ЯП F#, наследующий идеи функциональных ЯП ML, CAML, OCAML, поддерживает механизмы ЛП и ООП [40]:

– полный свод обычных механизмов функционального программирования, включая функции высших порядков; каррирование и сопоставление с образцом;
– динамическое связывание;
– ленивые и энергичные вычисления;
– замыкания функций и меморизация;
– квотирование выражений;
– конструирование выражений, частичное применение функции и мета-компиляция;
– разреженные матрицы;
– хеш-таблицы;
– mutable-переменные;
– монада недетерминированных вычислений;
– асинхронные выражения и параллельное программирование;
– интероперабельность с .NET.

Язык Scala объединяет механизмы ФП и ООП с резким акцентом на контроль ТД, удобный для разработки компиляторов, обеспечивающих надёжность программ [47]:

– объекты и их поведение определяется классами с возможностью множественного наследования;
– функции являются значениями и могут быть высших порядков;
– статическая типизация поддерживает обобщённые классы, встроенные классы и абстрактные типы, составные типы, полиморфные методы и др.;
– допускает включение новых языковых конструкций;
– взаимодействует с Java и .Net;
– допускает анонимные функции;
– автоматическое конструирование типо-зависимых замыканий функций;

– использует механизм сопоставления с образцом.

Объединение разных ПП в рамках одного ЯП или их поддержка при реализации СП повышает их сферу применения и способствует производительности труда программистов, смягчая избыточное разнообразие теорий и моделей, сложившихся на разных фазах и этапах ЖЦП.

13.3. Образовательные проблемы

Основная трудность перехода к новым парадигмам программирования – соблазн легкого пути, т.е. стремление быстро смоделировать привычные средства и методы программирования. Более надежный путь – исследовать их как незнакомые миры. Непривычные идеи легче воспринять как самостоятельную теорию или интеллектуальную игру, которая не только приведет к знакомым и интересным задачам, но и обеспечит преимущество – изящные решения и глубину понимания.

Так, например, функциональный подход исторически является основой для исследования средств и методов программирования, прототипирования и декомпозиции информационных систем, постановки новых задач искусственного интеллекта и развития современных методов параллельного и многоязыкового программирования.

В нашей стране популяризация идей ФП связана с именем С. С. Лаврова, руководившего реализацией первой в нашей стране Лисп-системы на БЭСМ-6 и лично участвовавшего в разработке этой системы. Идеи ФП отражены в учебнике С. С. Лаврова по теории программирования. В журнале КИвО опубликованы статьи по ФП, адресованные школьникам и школьным учителям [24]. В конце XX в. С. С. Лавров подготовил статью о ФП для пока не развернутого проекта Лексикона программирования и написал учебный Лисп-интерпретатор.

Традиционно языки и системы функционального программирования обогащены типовыми средствами для вычислений с повышенной точностью. Возможна визуализация данных и программ. На сегодняшний день утратили актуальность опасения относительно ресурсно-эксплуатационных трудностей функционального программирования.

В задачах информатики ФП определяет грань между методами теоретического и системного программирования и методами разработки информационных систем (ИС), наследующих проверенные свойства многократно используемых готовых компонентов. Рост сложности ИС требует передвижения этой грани с обработки строк, слов и структур данных на оперирование информационными ресурсами, файловыми системами, архивами,

сетями, многоядерными архитектурами, что повышает интерес к гибкому использованию мощных средств функционального и параллельного программирования, а также верификации и автоматическому тестированию компонентов.

Методы ФП позволяют радикально снижать трудоемкость отладки компонентов и сложность тестирования ИС благодаря предпочтению достаточно универсальных определений и динамическому контролю корректности исполнения программ. Средства ФП кроме проверки и удостоверения формализуемых свойств типовых компонентов в процессе разработки и создания новых языков программирования (ЯП) гарантируют сохранение этих свойств и обеспечивают их накопление при развитии компонентов, а также дают естественность и простоту процессу интеграции СП.

Сложность решения задач с помощью функциональных определений преодолевается чисто алгебраически: нацеленностью на формализацию основного множества объектов и определения полной семантической системы операций над ними. Это позволяет представлять классы задач и их решений строгими формулами, для наглядности упрощаемыми введением дополнительных функциональных символов. При необходимости такие символы вносятся в определение алгебраической системы, что приводит к ее расширению. Вводятся новые функции, подобные леммам и другим вспомогательным построениям в математике. Активно используется рекурсия и символьные обозначения как данных, так и действий и любых формул, удобных при определении функций.

Во многих системах функционального программирования обеспечена организация параллельных процессов, имеются средства объектно-ориентированного программирования [25, 47]. Поддержано управление компиляцией и конструирование компиляторов. Существенно, что операции функционального языка обладают машинно-независимой семантикой. Поэтому для функциональных программ проблема переноса стоит намного мягче, чем для стандартных языков, таких как C, Pascal, Fortran. Это делает функциональные ЯП перспективными для образовательных применений. Успех Logo в детском и любительском программировании не случаен [17].

Языки XXI века, начиная с учебных ЯП, как правило поддерживают все основные ПП (A++, Oz), что можно рассматривать как перспективы формирования единой парадигмы программирования. Впрочем, шутливый этюд «Настоящие программисты не программируют на Паскале» заключается словами «Настоящий программист всегда сумеет написать на любом языке программу на Фортране». Это может означать, что настоящий про-

граммист сумеет в рамках любой парадигмы написать императивно-процедурную программу.

ЗАКЛЮЧЕНИЕ

Более тонкий анализ реализационной прагматики при определении парадигм программирования можно выразить в терминах отношений на основных множествах сигнатур семантических систем, которые проще представлять в денотационной форме. При определении языка Lisp Дж. МакКарти уделил внимание аксиоматическим определениям специфики базовых операций над символьными выражениями. Возможно, элегантная строгость таких определений повлияла на надежность и гибкость ФП.

Новые и долгоживущие ЯП как правило имеют мультипарадигматический характер. Следует обратить внимание на появление учебных языков программирования (A++, Oz), поддерживающих все основные парадигмы программирования, что показывает их взаимную дополнительность и образовательное значение.

Можно выделять ведущую парадигму ЯП и ряд дополнительных парадигм. В «опорных» ЯП выделяются фрагменты, полностью соответствующие одной парадигме. Описания основных парадигм ЯВУ несложно моделируются средствами ФП. Основные различия связаны с вариациями дисциплины доступа к отдельным категориям имен. Фрагменты, соответствующие дополнительной парадигме ЯП, могут выполнять роль модели этой парадигмы при сравнении с другими языками.

Приведенные функциональные модели основных ПП показывают, что функциональный стиль и традиция реализации функциональных систем программирования могут дать систему мер для сравнения языков программирования.

СПИСОК ЛИТЕРАТУРЫ

1. Андреева Т.А., Ануреев И.С., Бодин Е.В., Городняя Л.В., Марчук А.Г., Мурзин Ф.А., Шилов Н.В.. Компьютерные языки как форма и средство представления, порождения и анализа научных и профессиональных знаний // Тр. XV Всерос. научно-методической конф. «Телематика-2008». – Санкт-Петербург, 2008. – С. 77–78.
2. Безбородов Ю.М. Сравнительный курс языка PL/1. – М.: Наука, 1980. – 191 с.
3. Болски М.И. Язык программирования Си. – М.: Радио и связь, 1988. – 96 с.
4. Вегнер П. Программирование на языке Ада. – М.: Мир. 1983. – 239 с.

5. Вдовкин С.В., Кубенский А.А., Сафонов В.О. Реализация языка CLU // Системная информатика. Вып. 1: Проблемы современного программирования. – Новосибирск: Наука. Сиб. отд-ние, 1991. – С. 127–130.
6. Вирт Н. От Модулы к Оберону // Системная информатика. Вып. 1: Проблемы современного программирования. − Новосибирск: Наука. Сиб. отд-ние, 1991. – С. 63–75
7. Гололобов В.И., Чеблаков Б.Г., Чинин Г.Д. Описание языка ЯРМО. – Новосибирск, 1980. – (Препр. / ВЦ АН СССР. Сибирское отд-ние; № 247, 248).
8. Голуб А.И. С и С++. Правила программирования. – М.: БИНОМ, 1996. – 271 с.
9. Городняя Л.В. Некоторые диалекты Лиспа и сферы их приложения // Трансляция и преобразования программ. – Новосибирск, 1984. – С. 60–71.
10. Городняя Л.В. Основы функционального программирования. – М.: Интернет-Университет Информационных технологий. – http://www.intuit.ru, 2004. – 272 с. *(проверено 1.12.2014)*
11. Городняя Л.В. Функциональный подход к описанию парадигм программированияю – Новосибирск, 2009. – 66 с. – (Препр. / ИСИ СО РАН; № 152).
12. Городняя Л.В. Реализация Лисп-интерпретатора // Вычислительная математика и программирование. – Новосибирск, 1974. – С. 24–35.
13. Грисуолд, Р., Поудж Дж., Полонски И. Язык программирвоания Снобол-4. – М.: Мир, 1980. – 268 с.
14. Дал У., Мюрхауг Б., Нюгорд К. Симула -67 универсальный язык программирования. – М.: Мир, 1969. – 99 с.
15. Дейкстра Э. Дисциплина программирования. – М.: Мир, 1978. – 275 с.
16. Дехтяренко И.А. Декларативное программирование. – http://www.softcraft.ru/paradigm/dp/dp02.shtml *(проверено 1.12.2014)*
17. Дьяконов В.П. Язык программирования Лого. – М.: Радио и связь, 1991. – 145 с.
18. Ершов А.П., Кожухин Г.И., Поттосин И.В. Руководство к пользованию системой АЛЬФА. – Новосибирск: Наука, 1968. – 179 с.
19. Захаров Л.А., Покровский С.Б., Степанов Г.Г., Тен С.В. Многоязыковая транслирующая система. – Новосибирск, 1987. – 151 с.
20. Камынин С.С., Любимский Э.З. Алгоритмический машинно-ориентированный язык АЛМО // Алгоритмы и алгоритмические языки. – ВЦ АН СССР, 1967. – Вып. 1.
21. Катцан Г. Язык Фортран 77. – М.: Мир. 1982. – 208 с.
22. Коддингтон Л. Ускоренный курс Кобола. – М.: Мир. 1974. – 270 с.
23. Лавров С.С. Методы задания семантики языков программирования // Программирование. – 1978. – № 6. – С. 3–10.
24. Лавров С.С., Городняя Л.В. Функциональное программирование. Интерпретатор языка Лисп // Компьютерные инструменты в образовании. – С-Пб, 2002. – № 5.
25. Лейнингем И. Освой самостоятельно Python. – М. Вильямс. – 444 с.

26. Малпас Дж. Реляционный язык Пролог и его применение. – М.: Наука, 1990. – 463 с.
27. Мейер Б. Основы объектно-ориентированного проектирования. – М.: Интернет-Университет Информационных технологий. : http://www.intuit.ru/department/se/oopbases/, 2007 *(проверено 1.12.2014)*
28. Непейвода Н.Н. Стили и методы программирования. – М.: Интернет-Университет Информационных технологий. – http://www.intuit.ru/department/se/progstyles/ , 2004 *(проверено 1.12.2014)*
29. Пентковский В.М. Автокод Эльбрус. – М.: Наука. 1982. – 350 с.
30. Пеппер П., Экснер Ю., Зюдхольд М. Функциональный поход к разработке программ с развитым параллелизмом // Системная информатика. Вып. 4: Методы теоретического и системного программирования. – Новосибирск: Наука. Сиб. изд. фирма, 1995. – С. 334–360.
31. Пересмотренное сообщение об АЛГОЛЕ 68 / Под ред. А. П. Ершова. – М.: Мир, 1979
32. Пильщиков В.Н. Язык Плэнер. – М.: Наука. 1983. – 207 с.
33. Поттосин И.В. Система СОКРАТ: Окружение программирования для встроенных систем. – Новосибирск, 1992. – 20с. – (Препр ./ РАН. Сиб. отд-ние. ИСИ; № 11).
34. Пратт Т. Языки программирования: разработка и реализация. – М.: Мир, 1979. – С. 20.
35. Пратт Т., Зелковиц М. Языки программирования. Разработка и реализация / Под общей редакцией А.Матросова. – СПб.: Питер, 2002. – 688 с.
36. Прехельт Л. Эмпирическое сравнение семи языков программирования // Открытые системы. – М., 2000. – 12(56). – С. 45–52.
37. Рендел Б., Рассел Л. Реализация Алгола-60. – М.: Мир, 1967. – 475 с.
38. Романенко Г. Л. Рефал-4 – расширение Рефала-2, обеспечивающее выразительность результатов прогонки. – М., 1987. – 27 с. – (Препр. / ИПМ АН СССР; № 147).
39. Савенков К. Верификация программ на моделях. / Курс ВМК МГУ имени М.В.Ломоносова. http://savenkov.lvk.cs.msu.su/mc/lect02.pdf
40. Д. В. Сошников Функциональное программирование на языке F#. – М.: ДМК Пресс, 2011.
41. Стивен Р. Палмер, Джон М.Фелсинг. Практическое руководство по функционально-ориентированной разработке ПО. – М.: Вильямс, 2002. – 299 с.
42. Страуструп Б. Язык программирования Си++. – М.: Радио и связь, 1992. – 348 с.
43. Уоткинс Д., Хаммонд М., Эйбрамз Б. – Программирование на платформе .Net. – М. Вильямс, 2003. – С. 367.
44. Фуксман А.П. Технические аспекты создания программных систем. – М.: Статистика, 1979. – 180 с.
45. Хендерсон П. Функциональное программирование. – М.: Мир, 1983. – 349 с.
46. Хигман Б. Сравнительное изучение ЯП. – М.: Мир, 1974. – 204 с.

47. Хорстман К. Scala для нетерпеливых. – ДМК пресс, 2013. – 408 с. – 300 экз. – ISBN 978-5-94074-920-2, 978-0-321-77409-5.
48. Кей С. Хорстманн Java SE 8. Вводный курс = Java SE 8 for the Really Impatient. – М.: «Вильямс», 2014. – 208 с. – ISBN 978-5-8459-1900-7.
49. Хьювенен Э., Сеппанен Й. Мир Лиспа. – М.: Наука, 1994. Т. 1, 2.
50. Хьюз Дж., Мичтом Дж. Структурный подход к программированию. – М.: Мир, 1985.
51. Шнейдер П.А.Основы программирования на языке Пролог. – М.: Интернет-Университет Информационных технологий. – 2004. – 196 с. http://www.intuit.ru/studies/courses/44/44/lecture/ *(проверено 1.12.2014)*
52. Backus J. Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs // Communs. ACM. – 1978. – Vol. 21, 8. – P. 613–641.
53. Henner C.R. A Simple Set Theory for Computer Science – Toronto, 1979. – TR # 102. – 12 p.
54. Hudak P. Conseption, Evolution and Application of Functional Languages // ACM Computing Servys. – 1989. – Vol .21, N 3. – P. 359-411.
55. Graham P. ANSI Common Lisp. – Prentice Hall,1996. – 432 p.
56. Knoop J. Compiler Construction // 20th Intern. Conf., CC 2011. Held as Part of the Joint European Conf. on Theory and Practice of Software, ETAPS 2011. – Lect. Notes Comput. Sci. – 2011. – Vol. 6601.
57. Lucas P., Lauer P., Stigleitner H. Method and Notation for the Formal Definition of Programming Languges. IBM Laboratory – Vienna, 1968. – TR 25.087.
58. McCarthy J. LISP 1.5 Programming Mannual. – The MIT Press, Cambridge, 1963. – 106 p.
59. Ritchie D.M., Tompson K. The UNIX Time-Sharing System // Bell System Technical J. – 1978. – Vol. 57, N 6. – P. 1905–1929.
60. Schwartz, Jacob T. Set Theory as a Language for Program Specification and Programming. – Courant Institute of Mathematical Sciences, New York University, 1970.
61. http://haskell.org/aboutHaskell.html – Материалы по языку Haskell. *(проверено 1.12.2014)*
62. http://refal.org/rf5_frm.htm – Материалы по языку Рефал. *(проверено 1.12.2014)*
63. http://www.cons.org/cmucl/ – Сайт с материалами по особо эффективной реализации Lisp-а – CMUCL. *(проверено 1.12.2014)*
64. http://www.erlang.org – Официальный сайт языка Erlang. *(проверено 1.12.2014)*
65. http://www.gwydiondylan.org/drm/drm_7.htm – Официальный сайт языка Dylan . *(проверено 1.12.2014)*
66. http://www.smalltalk.ru – сайт с материалами по языку SmallTalk. *(проверено 1.12.2014)*
67. http://www.cs.bham.ac.uk/~pjh/modules/current/25433/examples/l15_example3.html

ТЕРМИНЫ И ОБОЗНАЧЕНИЯ

Ассоциативный список – список пар, предназначенный для хранения соответствия между именами и их определениями (значения переменных, констант, определения функций).

Атом – данное, не разделяемое на части средствами ЯП.

Дамп – резервная память для хранения промежуточных результатов, которые могут понадобиться при дальнейших вычислениях.

Деструктивные операции – выполняются на памяти операндов, что может повлечь потерю данных.

Замыкание функции – конструкция из определения функции и таблицы значений используемых в ней свободных переменных.

Клауза – предикат и соответствующая ему ветвь для вывода цели вычисления.

Ленивое вычисление – отложенное действие, выполняемое лишь если оно необходимо для получения результата.

Мера организованности программы – зависимость объёма отладки модифицируемой программы от объёма вносимых изменений.

Монада – вспомогательная семантическая система ЯП со своими правилами выполнения действий и вычисления функций.

Область видимости имён – участки программы, на которых имена имеют определение.

Псевдо-функции – кроме значения производят дополнительные действия.

Рабочие переменные – видимы в пределах блока.

Раздельная компиляция – методика создания кода частей программы для их многократного использования в других программах.

Ранг работоспособности программы – полнота множества возможных данных, на которые программа реагирует разумно.

Реализационное замыкание ЯП – расширение определения ЯП, создаваемое для его эффективной реализации.

Свободные переменные – получают значение или определение вне задаваемого фрагмента.

Семантическая декомпозиция – разложение определения на части, смысл каждой из которых может быть выражен на естественном языке.

Степень изученности задачи – оценка готовности к программированию постановки задачи и методов её решения.

Структурное программирование – методика представления императивно-процедурных программ, упрощающая их отладку.

Универсальная семантическая функция ЯП – функция вычисления результата любой правильно представленной на ЯП функции от допустимых данных.

Уровень абстрагирования понятий – характеристика независимости понятия от малосущественных свойств конкретных примеров.

Хэш-функции – методика хранения динамически изменяемого конечного множества из элементов бесконечного множества.

Аббревиатуры

Обозначение	*Расшифровка*
АМ	Абстрактная машина
АС	Абстрактный синтаксис
ЖЦП	Жизненный цикл программ
ИП	Императивное программирование
ЛП	Логическое программирование
ООП	Объектно-ориентированное программирование
ОС	Операционная семантика
ПП	Парадигма программирования
РП	Реализационная прагматика
СД	Структуры данных
СП	Система программирования
ТД	Типы данных
УЯ	Учебный язык программирования
ФП	Функциональное программирование
ЭП	Эксплуатационная прагматика
ЯВУ	Язык высокого уровня
ЯП	Язык программирования
ЯСП	Язык и система программирования
ЯФП	Язык функционального программирования
SECD	Абстрактная машина языка Lisp

Обозначения

(X . Y) – работает как (cons X Y) – X становится «головой» списка Y.

`(x . l )` – это значит, что первый элемент списка – `x`, а остальные находятся в списке `l`.

(x y . l) – первый элемент списка – x, второй элемент списка – y, остальные находятся в списке l и т.д.

([XL . YL] . AL) – работает как (pairlis XL YL AL) – функция аргументов XL, YL, AL строит список пар-консолидаций соответствующих элементов из списков XL, YL и присоединяет их к списку AL. Полученный список пар, похожий на таблицу с двумя столбцами, называется ассоциативным списком или таблицей атомов. Такой список может использоваться для связывания имен переменных и функций при организации вычислений интерпретатором.

(X | Y) – работает как (append X Y) – сцепляет списки в один общий список.

AL[X] – работает как (assoc X AL) – функция двух аргументов, X и AL. Если AL – таблица атомов, подобная тому, что формирует функция pairlis, то assoc выбирает из него первую пару, начинающуюся с X. Таким образом, это функция поиска определения или значения в таблице атомов.

[x] – содержимое памяти по адресу x

e[n] – содержимое n-го элемента контекста

{A | B | … | Z} – множество вариантов.

A(Pr) – число аргументов процедуры Pr

L(Pr) – число локальных переменных процедуры Pr

@F – адрес подпрограммы, выполняющей функцию F.

@c – адрес позиции «c» в программе

_ – Произвольное значение (_ подчерк)

СОДЕРЖАНИЕ

[8] Части 1,2 и 4,5 – отдельные препринты.

Л.В. Городняя

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 3
Основные парадигмы программирования
Языки высокого уровня

Препринт
174

Рукопись поступила в редакцию 13.02.2015
Редактор Т. М. Бульонкова
Рецензент Ф.А. Мурзин

Подписано в печать 11.03.2015
Формат бумаги 60 × 84 1/16 Объем 5.7 уч.-изд.л., 6.25 п.л.
Тираж 60 экз.

Типография Оригинал-2, г. Бердск, ул. Олега Кошевого, 6, оф. 2
тел./факс: 8 (383) 328-32-38, (38341) 2-12-42, сот.: 8 913 987 77 67