МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

КАФЕДРА СИСТЕМ ИНФОРМАТИКИ

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

КУРС ЛЕКЦИЙ

НОВОСИБИРСК

2015

УДК 004.43 (042.4)
ББК 32.973-018
Г

Рецензент
канд. физ.-мат. наук, Ф. А. Мурзин

Издание подготовлено в рамках реализации *Программы развития государственного образовательного учреждения высшего профессионального образования «Новосибирский государственный университет»* на 2009–2018 годы.

Городняя, Л. В.
Г Парадигма программирования : курс лекций / Л. В. Городняя ; Новосиб. гос. ун-т. – Новосибирск : РИЦ НГУ, 2015. – 206 с.

ISBN _____

Курс лекций посвящен проблеме анализа, сравнения и определения парадигм программирования. Содержание представляет интерес для специалистов по программированию и информационным технологиям.

УДК 004.43 (042.4)
ББК 32.973-018

ISBN_____

СОДЕРЖАНИЕ

Содержание 3
Введение 5
Лекция 1. Проявление парадигм программирования 12
1.1. Многоликое программирование 12
1.2. Технологии программирования 15
1.3. Жизненный цикл программ 19
1.4. Развитие парадигм программирования 23
1.5. Эксплуатационная прагматика 25
Лекция 2. Поддержка парадигм программирования 27
2.1. Семантика 29
2.2. Абстрактная машина 35
2.3. Структуры данных 45
2.4. Реализационная прагматика 49
2.5. Определитель парадигм 53
Лекция 3. Языки низкого уровня 58
3.1. Императивное программирование на ассемблере 59
3.2. Стековая машина Forth 66
3.3. Продукционная макротехника 71
3.4. Языки управления процессами 79

Языки высокого уровня

Лекция 4. Императивно-процедурное программирование 91
4.1. Особенности представления программ на Си 91
4.2 Структурное программирование 93
4.3.Функциональная модель ИП 94
4.4. Спецификация 99
Лекция 5. Функциональное программирование 101
5.1. Основы 102
5.2. Функциональные ЯП 113
5.3. Отображения и функционалы 114
5.4. Отложенные действия 118
5.5. Свойства атомов 120
5.6. Гибкий интерпретатор 121
5.7. Функциональная модель взаимодействия монад 123
5.8. Спецификация 127

Лекция 6. Логическое программирование 128
6.1. Операционная семантика 129
6.2. Основы 131
6.3. Язык декларативного программирования Prolog 132
.
6.4. Функциональная модель ЛП 135
6.5. Модели недетерминизма 138
6.6. Спецификация 140
Лекция 7. Объектно-ориентированное программирование 141
7.1. Общее представление 142
7.2. Абстрактная машина 145
7.3. C++ 147
7.4. Функциональные модели ООП 153
7.5. Спецификация 157
7.6. Мультипарадигмальные языки программирования 158
Лекция 8. Параллельное программирование 164
8.1. Пространство решений 165
8.2. Параллельные алгоритмы 166
8.3. Практичные системы программирования 169
8.4. Модели параллелизма в языках программирования 175
8.5. Языки сверхвысокого уровня 185
8.6. Высокопроизводительное программирование 187
8.7. Трансформационная семантика 191
8.8. Абстрактный комплекс 193
8.9. Память 195
Заключение 197
Список литературы 199
Приложение 201

ВВЕДЕНИЕ

Курс лекций знакомит с разнообразием проявлений парадигм программирования (ПП) и подходов к их поддержке в языках и системах программирования (ЯСП). В центре внимания находятся исторически значимые и концептуальные языки программирования, в которых видны ключевые идеи и практические следствия их реализации. Стили и языки программирования, характерные для рассматриваемых парадигм, отражают эволюцию технологий программирования (ТП), используемых при решении задач системной и прикладной информатики от средств машинного программирования на стыке с аппаратурой до языков сверхвысокого уровня и систем высокопроизводительного программирования, включая средства поддержки жизненного цикла программ (ЖЦП).

Альтернативные подходы к обработке информации, сложившиеся при создании и применении языков и систем программирования, принято называть парадигмами программирования. Изучение и чёткая классификация уже сложившихся и новых ПП призваны помочь как обоснованному выбору, так и созданию компьютерных языков при формировании программных проектов и совершенствовании информационных технологий (ИТ).

Парадигмами программирования в форме языков и систем программирования представлено знание о потенциале ИТ. В момент создания язык программирования (ЯП) отражает некий прогноз относительно области приложения ИТ. Практика разработки и применения систем программирования (СП) конкретизирует и уточняет такое знание в форме отлаженных программ с комплектами данных для них и прецедентами успешного их применения. Успех применения ЯП можно рассматривать как результат удачного выбора ПП, задающей концептуальную схему постановки проблем и методов их решения с удобным инструментом «грамотного» описания фактов, событий, явлений и процессов, выделения частных и общих понятий. Развитие ПП отражает практичность языковых понятий и реализационных структур, используемых при создании сложных программных систем. Интересно отметить, что рейтинг популярности языков программирования отличается от рейтинга ЯП, использованных в успешно завершенных проектах.

Именно парадигмам программирования Роберт Флойд посвятил свою Тьюринговскую лекцию, в которой обратил внимание на значимость этого понятия в контексте проблемы обучения программистов. Авторитетный ученый, заложивший основы теорий анализа и верификации программ, автор ряда эффективных методов обработки данных счёл необходимым

подчеркнуть влияние ПП на успех программистских проектов, на особенности изучения разных ПП и на то, как они должны быть поддержаны в языках программирования. Рассматривая пример структурного программирования в качестве доминирующей методологии программирования, Р. Флойд отметил, во-первых, нацеленность этой парадигмы на нисходящее проектирование, пошаговое улучшение и сведение задачи к более простым подзадачам; во-вторых, переход от конкретных объектов и функций машинного уровня к более абстрактным объектам и функциям, позволяющим продумывать модули, выделяемые при нисходящем проектировании.

На своем программистском опыте Р. Флойд сделал интересное наблюдение: искусство программирования включает в себя расширение репертуара используемых парадигм. Его внимание привлекла задача подготовки рекурсивных сопрограмм, удобно формулируемых в терминах недетерминизма, неэффективность которого практично преодолевается макротехникой. Р. Флойд высоко оценил созданные в MIT многочисленные примеры потенциала программирования на универсальном языке Lisp, показывающие путь от простейших списков до универсальных структур данных (СД), способствующих манипулированию программами как данными.

Р. Флойд выразил уверенность, что можно наладить ясное обучение ряду таких семантических методов для всех уровней программного проекта, чтобы у обученных студентов хватало головы на решение полного спектра проблем от упрощённой учебной постановки задачи до непрерывно расширяющегося класса практических задач.

Следует отметить, что за тридцать лет, прошедших со времени Тьюринговой лекции Р. Флойда, число различных языков и систем программирования возросло с нескольких сотен до десятков тысяч. При этом число парадигм не столь велико. В разных источниках называют от двадцати до сорока парадигм, нередко включая в их перечень отдельные техники и методы.

Изучение и чёткая систематизация уже сложившихся ПП призваны помочь обоснованному выбору подходов к обеспечению производительности, надежности и эффективности сложных ИС, конструируемых с использованием разносортных ИТ и сервисов, применяемых в разных условиях.

Средства низкоуровневого программирования могут быть охарактеризованы возможностью реализации эффективных решений ценой использования общего доступа к слабо защищенным СД. Языкам высокого уровня свойственно использование расширяемой иерархии СД, компоненты

которой защищены от бесконтрольного взаимодействия независимо создаваемых фрагментов программы. Механизмы сверхвысокого уровня (языки спецификаций, языки параллельного программирования, системы представления знаний и т. д.) нацелены на полноту пространства реализационных решений, факторизуемого по отдельным направлениям проблем, связанных с разработкой и применением долгоживущих программ.

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. 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 эффективности 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.

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.

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. Ассемблер Эльбрус – яркий пример отечественной реализации ЯВУ аппаратными средствами.

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);
– различны 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 0

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 вспомогательных систем обработки целых и вещественных чисел в разных диапазонах, обычно зависящих от разрядности адресов и машинных слов. Имеются дополнительные системы работы с указателями, символами и строками и средства конструирования типов данных. Поддерживается набор стандартных операторов управления вычислениями: ветвления, циклы и вызовы процедур/функций. Кроме работы с глобальными объектами, используются разные схемы локализации хранимых в памяти объектов, что обеспечено организацией структур данных по принципу иерархии описаний или динамики исполнения.

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

Инструментальное ядро ЯВУ можно ограничить одной арифметической системой. Элементарные уровни опорных ЯВУ разных парадигм можно описать как расширения или конкретизации такого ядра.

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;
- универсальный язык Lisp дал жизнь машинно-независимому стилю символьной обработки данных, что привело к парадигме функционального программирования;
- популярность языка C связана с решением проблем машинно-зависимого переноса программ, что одновременно поддержало перенос на множество архитектур;
- логическое программирование на языке Prolog позволило работать с недоопределёнными постановками задач, повышая степень их изученности;
- появление C++ и ООП расширило сферы приложения информационных систем, при конструировании которых необходимо повторное программирование.

Теоретически различие ПП достаточно ясно выражается на уровне операционной семантики, представляющей детали структур памяти и механизмы выполнения укрупнённых действий. 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 и подходов к динамической оптимизации программ.

Курс лекций начинается с общего представления о разнообразии задач программирования и решений, принимаемых при программировании. Затем следует краткий экскурс в методы определения языков программирования, включая вопросы построения систем программирования как основных средств поддержки ПП. Рассматриваются вопросы прагматической классификации ПП на основе анализа особенностей их операционной семантики, представленной в форме семантических систем, реализационной прагматики, дающей примеры типовых механизмов, и эксплуатационной прагматики, показывающей практику их применения. Далее описаны конкретные средства и методы поддержки языков низкого уровня, основных парадигм языков высокого уровня, параллельного программирования и базовых парадигм учебных языков и систем программирования, имеющих образовательное значение. В заключении предпринята попытка резюмировать особенности развития парадигм программирования, описать динамику их кристаллизации и наметить технику парадигматической характеристики языков и систем программирования. Приведена сводка терминов, толкование которых может вызывать разночтения.

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

ЛЕКЦИЯ 1. ПРОЯВЛЕНИЕ ПАРАДИГМ ПРОГРАММИРОВАНИЯ

В середине прошлого (XX) века термин «программирование» не подразумевал связи с компьютером. Например, можно было увидеть название книги «Программирование для ЭВМ». Теперь по умолчанию термин «программирование» означает организацию процессов на компьютерах и компьютерных сетях.

Программирование как наука существенно отличается от математики и физики с точки зрения оценки результатов исследования. Уровень результатов, полученных физиками и математиками, обычно оценивают специалисты близкой или более высокой квалификации. В оценке результатов программирования большую роль играет оценка пользователя, не претендующего на программистские познания. Поэтому специалисты в области программирования частично выполняют функцию переводчика своих профессиональных терминов в понятия пользователя.

Программирование имеет свой специфичный метод установления достоверности результатов – это компьютерный эксперимент. Если в математике достоверность сводится к доказательным построениям, понятным лишь специалистам, а в физике – к воспроизводимому лабораторному эксперименту, требующему специального оснащения, то компьютерный эксперимент доступен широкой публике.

1.1. Многоликое программирование

Другая особенность программирования обусловлена его зависимостью от быстро развивающейся элементной и инструментальной базы. Для быстрого обновления знаний и навыков нужен классический фундамент. Программистские знания – это сочетание классики и моды.

Критерии качества программ весьма разнообразны. Их выбор и упорядочение, по существу, зависит от класса задач и условий применения программ:

- результативность;
- надежность;
- устойчивость;
- автоматизируемость;
- эффективное использование ресурсов (время, память, устройства, информация, люди);
- удобство разработки и применения;
- наглядность текста программы;
- наблюдаемость процесса работы программы;
- диагностика происходящего.

Порядок критериев нередко претерпевает изменения по мере развития области применения программы, роста квалификации пользователей, модернизации оборудования, информационных технологий и программотехники. Вытекающее из этого непрерывное развитие пространства, в котором решается задача, вводит дополнительные требования к стилю программирования информационных систем:

- гибкость;
- модифицируемость;
- верифицируемость;
- безопасность;
- мобильность/переносимость;
- адаптируемость;
- конструктивность;
- измеримость характеристик и качества;
- улучшаемость.

Программирование как наука, искусство и технология исследует и творчески развивает процессы создания и применения программ, определяет средства и методы конструирования программ, разнообразие которых складывается в практике и экспериментах и фиксируется в форме ЯП. Сложности классификации быстро расширяющегося множества ЯП приводят к выделению понятия «парадигмы программирования», число которых меняется не столь стремительно. Это ставит задачу определения принадлежности ЯП конкретной ПП или поддержки ПП определённым ЯП. Для решения этой задачи ПП характеризуется взаимодействием основных семантических систем, таких как вычисление, обработка структур данных, хранение данных и управление обработкой данных. ЯП, поддерживающий некоторую ПП, в значительной мере наследует характеристику ПП при его реализации в СП на уровне абстрагирования операционной семантики от возможностей доступной аппаратуры. При таком подходе выделяются три общие категории парадигм:

- низкоуровневое программирование;
- программирование на языках высокого уровня;
- подготовка программ на базе языков сверхвысокого уровня.

Низкоуровневое программирование связано со структурами данных, обусловленными архитектурой и оборудованием. При хранении данных и программ используется глобальная память и автоматная модель управления обработкой данных.

Программирование на языках высокого уровня приспособлено к заданию структур данных, отражающих природу решаемых задач. Используется расширяемая иерархия областей видимости структур данных и процедур их обработки, подчиненная структурно-логической модели управления, допускающей сходимость процесса отладки программ.

Подготовка программ на базе языков сверхвысокого уровня нацелена на представление регулярных, эффективно реализуемых структур данных, при обработке которых возможны преобразования представления данных и программ, использование подобий и доказательных построений, гарантирующих высокую производительность вычислений и надежность процесса разработки программ, включая подготовку программ для многопроцессорных конфигураций.

Дальнейшая детализация зависит от специфики реализационных решений, типичных для СП, поддерживающих конкретные ПП.

Есть основания полагать, что отмеченная Р. Флойдом потребность в расширении репертуара парадигм при программировании связана с динамикой представления знаний, сводимой к чередованию фаз восходящего и нисходящего проектирования решений развивающихся постановок задач. Динамика представления знаний сводится к переходу от одного представления к другому. Чередование стадий индуктивного и дедуктивного развития знания можно рассматривать как обоснование выбора метода программирования в зависимости от зрелости, степени изученности решаемой задаче.

В методике программирования конкретизация соответствует нисходящим методам «сверху вниз». (Вопреки лингвистической ассоциации нет причин считать нисходящие методы обратными восходящим. Обобщение психологически не симметрично конкретизации. Top_Down – Bottom_Up.)

По степени изученности существенно различаются следующие категории постановок задач, влияющие на стиль мышления и выбор методов решения задач:

- новые;
- исследовательские;
- практичные;
- эффективные.

Для новых постановок задач характерно отсутствие доступного прецедента решения задачи, новизна используемых средств или недостаток опыта исполнителей. Исследовательские постановки задач обычно усложнены требованиями уникальности и универсальности. Практичные

постановки задач нацелены на актуальность и удобство применения. Эффективные постановки задач включают в себя исследование возможностей используемых средств, связанных с мерой организованности созданной программы и рангом работоспособности реализованных решений.

Макетный образец решения новой задачи работоспособен при предъявлении автором небольшого набора подходящих данных. Для экспериментального полигона отладки решений исследовательских задач требуется приспособленность к обработке почти любых данных, которые могут быть заданы в качестве входных. Практичная версия может быть ограничена обработкой данных, реально встречающихся в сфере её приложения. Для эффективной реализации нужны специально подобранные данные, показывающие её превосходство над менее искусно выполненными решениями задачи.

По специфике программирования весьма существенно различаются следующие категории задач:

– многоуровневое абстрагирование для особо важных и сложных задач со специальными методами решения;
– бизнес-приложения, зависящие от динамики производственной деятельности человека;
– сбор фактов и накопление знаний для не вполне определенных задач;
– исследование и разработка новых алгоритмов и структур данных, включая создание новых языков;
– реализация хорошо изученных алгоритмов для корректных задач.

Независимо от области приложения, парадигмы программирования могут отличаться уровнем абстрагирования от возможностей аппаратуры, степенью изученности постановок задач, мерой организованности и рангом работоспособности программ решения задач. Дальнейшая классификация ПП проявляется в технологиях программирования, схемах жизненного цикла программ и лексиконе программирования.

1.2. Технологии программирования

Понятие «технология» подразумевает существование целевого производственного процесса, в рамках которого за определенное время на ограниченных ресурсах при известной квалификации персонала по конкретным техническим процедурам гарантированно может быть получен запланированный результат.

Основные вехи чисто программистской линии технологического повышения производительности труда в программировании связаны с кристаллизацией определённых ПП, обусловленных созданием конкретных ЯП, изобретением новых принципов реализации СП, появлением новых ТП:

- создание языка Fortran сопровождалось появлением технологии раздельной компиляции;
- язык Lisp дал жизнь технологии символьных вычислений и функционального программирования;
- разработка языка C как средства реализации операционных систем привела к технологии машинно-зависимого переноса программ;
- практика применения учебного языка Pascal позволила сформулировать принципы структурного программирования, выполняющего роль доминирующей методологии учебного программирования;
- язык Prolog связан с парадигмой логического программирования и реализацией средств логического вывода на базе частичного определения решений, достаточных для практики;
- идея организации процессов в языке Simula 67 и реализация работы с объектами в языке SmallTalk 80 дали импульс ООП с технологией декомпозиции программ по иерархии классов, допускающей расширение, не искажающее ранее отлаженные определения;
- разработка средств авторского форматирования текстов для публикаций TeX/LaTeX породила попытку технологии грамотного программирования (успешную, но не принятую программистами) и публикационно-ориентированную ветвь функционального программирования на языке Haskell.

За исключением языка Lisp, в большинстве этих подходов программа рассматривается как статический объект, тогда как в реальности она развивается и может частично видоизменяться в процессе разработки и исполнения. Ждут своего часа резервы верификации, обретающие практичность на современной технике.

В конце 1990-х годов стали набирать признание две перспективные технологии, представляющие собой разумный компромисс для разработки программ, обладающих новизной или исследовательским компонентом – экстремальное программирование (XP) и функционально-ориентированное проектирование (FDD).

Технология экстремального программирования сложилась в сфере производства игровых программ. В целом, экстремальное программирование показывает обнадёживающие результаты на постановках

задач с частыми обновлениями и соответствует восходящей методике программирования.

Технология функционально-ориентированного проектирования соответствует нисходящей методике программирования и учитывает ряд человеческих факторов, что позволяет процессу разработки достичь устойчивости. Такая технология пригодна для решения сложных задач с заметным исследовательским компонентом.

Следует отметить, что многие авторы современных технологий программирования выражают претензии к авторитетным рекомендациям по стилю и технике программирования, якобы сделавшим практику программирования беспомощной. Здесь желательно помнить о высоком темпе развития информационных технологий, осознание возможностей которых не успевает созревать.

Следует особо отметить трудоёмкость параллельного программирования, необходимость разработки методов верифицирующей компиляции и оптимизации программных компонентов, средств масштабируемой макрогенерации кода и автоматизируемых трансформаций программ с удостоверением сохранения свойств при их комплексации из ранее отлаженных компонентов.

Программа, не устаревая физически, подвержена сложным эффектам, связанным с обнаружением и исправлением ошибок и моральным устареванием не только реализованных решений, но и исходной постановки задачи.

Любое производство включает процессы поиска и устранения недочётов. В программировании это отладка и тестирование. Независимо от ПП и ТП в процессе разработки программ примерно 45% трудозатрат падает на долю автономного и комплексного тестирования и отладки программ. Столь высокая нагрузка на тестирование в практическом программировании требует ясности в понимании его целей, заметно отличающихся от естественно лингвистических представлений.

Тестирование – это организация такого применения программы, в котором обнаруживается наличие дефекта: ошибки или несоответствия ожиданиям конкретных групп пользователей. Это означает, что для определения множества недостаточно задать его элементы, надо ещё определить и всё, что множеству принадлежать не может.

Обычно подготовка тестовых данных подчинена ряду принципов и гипотез, нацеленных на экономию труда и надёжность результатов тестирования:

– для выбираемых входных данных сразу подбираются ожидаемые выходные данные;

– тест, не обнаруживающий ошибку, не имеет смысла как тест, но может быть полезен как демонстрационный материал;
– тестирование – это творческий процесс, т.к. возможности допустить ошибку в программе разнообразнее, чем её правильные построения;
– удачный тест выявляет новую, незамеченную ранее ошибку;

При автоматизации тестирования решают следующие проблемы:
– создание и накопление хорошего набора тестов;
– оценка набора на полноту по ряду критериев;
– исполнение программ на тестах, оценка результатов и их хранение;
– символьное исполнение программ;
– исполнение программ в альтернативных условиях, отличных от условий разработки.

Тонкости выбора тестового материала детально описаны в книге С. К. Черноножкина.

Различается отношение к источникам ошибок и мерам их профилактики в разных ПП. Если императивно-процедурное программирование (ИП) апеллирует к спецификации типов данных (ТД), обеспечивающих возможность статического контроля при компиляции программ, то функциональное программирование (ФП) предпочитает полный динамический контроль любых условий, гарантирующих корректность вычислений.

С математической точки зрения процесс тестирования можно представить как обнаружение точек графика реализуемой функции, через которые эта функция не должна бы проходить. Локализация ошибки – это установление отклоняющейся точки. Исправление ошибки – это преобразование определения функции к виду, дающему её уточнённый график. Может быть использовано сравнительное исполнение программы и её прототипа.

В целом, процесс разработки программ можно представить как последовательность шагов по уточнению постановки задачи, методов её решения, текста программы решения и набора данных, представляющих тесты и «нетесты». Если последовательность достигает состояния, в котором уточняемые сущности соответствуют друг другу, то утверждается, что завершена отладка программы.

Подходы к отладке не менее требовательны к творчеству, чем методы тестирования, но объём необходимых затрат на отладку может быть сокращён выбором стиля конструирования программ из проверенных шаблонов, рамками надёжных стандартов, использованием удобно

реализуемых моделей. Здесь многое даёт парадигма функционального программирования.

Сложность процесса отладки программы можно оценивать по мощности множества данных, на которых необходимо выполнить прогон программы. Так, например, множество данных для отладочного прогона программы, содержащей цикл, должно содержать не менее трёх наборов:

– без захода в тело цикла;
– с однократным заходом в тело цикла;
– с многократным прохождением тела цикла.

Таким образом, существует ряд ПП, появление которых обусловлено развитием ТП, представленных в форме языков и систем программирования, поддерживающих решение задач определённой степени изученности, меры организованности, уровня абстрагирования и ранга работоспособности программ при слабом учёте проблем тестирования и отладки, а также поддержки процесса разработки (полного жизненного цикла) программ.

1.3. Жизненный цикл программ (ЖЦП)

Важнейшая задача технологии программирования – обеспечение сходимости процесса отладки программы, т.е. достижение взаимного соответствия постановки задачи, методов её решения, текста программы решения и набора данных, представляющих тесты и «нетесты». Итоговая постановка задачи, решение которой реализовано в программе, может оказаться как обобщением, так и сужением исходной задачи, что можно назвать более общим термином – «уточнение». Обнаружение закономерностей в процессе разработки программ привело к понятию «жизненный цикл программы».

Изменение репертуара ПП в процессе создания программ связано со структурой расширяемого пространства, в котором принимаются решения при практическом программировании.

При решении самой простой задачи можно выделить такие этапы, как подготовка к решению, собственно реализация решений и оценка полученных результатов.

В прикладном программировании, использующем результаты докомпьютерного программирования, изначально выделяли этапы разработки и эксплуатации программы. Подразумевалось, что этапы строго упорядочены по времени и приводят к получению окончательного результата в виде программы решения поставленной задачи. Класс заранее

достаточно изученных для программирования задач быстро исчерпался, и потребовался неоднократный пересмотр принципов организации труда в программировании. Реальность заставила мириться с итеративностью фаз, любая из которых могла потребовать повторного прохождения, что получило название «принцип водопада». Кроме того, со временем обратили внимание на этап подготовки постановки задачи, опережающий этап разработки программ для решения новых задач. Много позднее возник этап «идентификации потребностей» – принятия решения о том, нужно ли вообще разрабатывать программу.

В производственном программировании сложилось понятие ЖЦП, структура которого требует не только четкого ответа на вопросы «Что? Как? Кто?», но и уяснения приоритетов между ними.

Что? Что должно получиться в результате решения задачи? Каковы исходные данные, обработку которых будет выполнять программа? В каком виде будет представлен результат обработки данных?

Как? Как будет устроена программа обработки? Каков алгоритм обработки данных? Из каких действий выстраивается функционирование программы?

Кто? Кто сумеет выполнить разработку? Какой должна быть его квалификация? Сможет ли он довести программирование до нужного уровня работоспособности программы на имеющемся оборудовании в заданный срок?

При разнообразии ответов на эти и многие другие сопутствующие вопросы возникли основания для выделения в ЖЦП возможных дополнительных фаз, существенных для зрелого долгоживущего производства, подразумевающего модернизацию программ и эволюцию требований к программе.

По мере расширения классов решаемых задач и областей приложения их решений сложилась каскадная схема ЖЦП, в которой задаются критерии завершения фаз, позволяющие управлять качеством разработки.

Таблица 1

Условия завершения фазы ЖЦП

№ фазы	*Название фазы*	*Условие завершения фазы*
0	постановка задачи	удостоверена достаточность средств управления качеством разрабатываемой программы;
1	определение требований	признана готовность документа,

		представляющего обзор желаемых результатов работы программы;
2	спецификация требований	выполнен выбор средств для подтверждения результатов программы;
3	проектирование	определены методы верификации правильности проекта;
4	реализация программы	проведено тестирование компонент программы на полном комплекте выбранных средств;
5	тестирование	выполнена интеграция/сборка программы из ранее готовых и разработанных компонентов;
6	сопровождение	проведена аттестация программы, показавшая фактические границы её работоспособности;
7	развитие	проверено соответствие функционирования программы пользовательским требованиям к решению задачи.

Расширение класса задач, методы решения которого находятся на уровне предварительного исследования, привели к табличному представлению совмещения фаз по времени выполнения разработки подобно диаграммам Г. Гантера. Шаги процесса разработки структурированы на фазы и технологические функции, образующие двумерное пространство с разметкой контрольных точек и возможных фазовых возвратов.

Фазы:

1) исследование;
2) анализ осуществимости;
3) конструирование возможных решений;
4) программирование;
5) оценка свойств программы;
6) использование программы.

Функции:

1) планирование;
2) разработка;
3) обслуживание;
4) документирование;

5) испытание;
6) поддержка;
7) сопровождение.

При создании такой таблицы рекомендовано учитывать следующие явления:

– программы подвержены итеративному развитию;
– существуют стабильные сценарии применения программы;
– неизбежно наращивание комплекса программируемых и выполняемых функций;
– ничто в разработке на делается однократно – раз и навсегда;
– не исключено размножение числа фаз.

Проявление зависимости трудоёмкости программирования от степени изученности/новизны решаемых задач наряду с высокой технологичностью программирования, допускающей перевод любых изученных задач в ранг готовых библиотечных модулей, привело к понятию «полный жизненный цикл программы» (ПЖЦП). Отличие от ЖЦП заключается в представлении схем, отражающих стадии созревания постановки задачи и допускающих развитие пространства реализуемых решений по мере продвижения от первых демонстрационных версий, прототипов, макетных образцов программы к готовому программному продукту, пригодному к полноценному функционированию без участия программиста.

С физической точки зрения программа не имеет износа. Неожиданности в ее функционировании могут быть связаны не столько с незамеченными при разработке недочётами, но и с появлением несоответствия принятых решений новым возможностям оборудования, изменению квалификации пользователей, развитию методов программирования. Поэтому программы подвержены моральному устареванию, которое влечёт за собой продолжение разработки программ с ранее завершенным ЖЦП, что приводит к уточнению перечня этапов разработки программы.

Схемы ПЖЦП выделяют стадии жизни программы, учитывающие разные степени изученности решаемых задач, что позволяет объективно оценивать число необходимых повторов в производстве программ.

Минимизацию повторов обеспечивают созданием специальных промышленных технологий программирования, противоречиво наследующих преимущества каскадной схемы ЖЦП и управления качеством по таблицам Гантера:

– форсированное принятие решений – сразу и как можно больше;

– управление качеством и профилактика необоснованных преждевременных решений.

Контрольные точки, используемые при управлении качеством, содержательно перекликаются с определениями критериев завершения фазы в каскадной схеме.

Для больших систем собственно программирование сводится к реализации или выбору модулей, сборке программ из модулей и документированию полученных результатов. Следует отметить, что реализация модулей обладает существенно большей (в 3–9 раз) трудоёмкостью, чем разработка функционально эквивалентных автономных программ. Сборка программ из готовых модулей имеет дополнительную нагрузку на ответственное изучение их устройства и функционирования.

Известно, что развитие постановок задач по степени изученности не обладает монотонностью изменения трудоёмкости реализации текущей версии программы. А именно, после небольших трудозатрат на разработку макетного образца, где главная цель – показать достоинства идеи, создание экспериментального полигона требует принципиально значительных трудозатрат на полноту исследования и разработки средств и методов решения задачи и определения границ практичности её реализации. Интуитивная оценка трудоёмкости реализации практичной версии в ранге производственного компромисса существенно меньше, что обычно провоцирует экономию на предварительных исследованиях в надежде угадать границы практичности, но обычно приводит к необходимости многократного версифицирования решения задачи.

Следует обратить внимание, что за понятием ПЖЦП стоит понятие «жизненный цикл новой задачи, решаемой методом разработки долго живущей программы». Можно сказать, что каждая стадия, этап, фаза заключается в том, что происходит уточнение информации о постановке решаемой задачи в изменяющихся условиях применения её решения.

В практике программирования встречается целый ряд схем ЖЦП, отличающихся по стилю мышления при постановке задач и подходам к методам решения задачи. Выбор ПП не только влияет на трудоёмкость, но и в заметной степени определяет работоспособность и живучесть разрабатываемой программы.

1.4. Развитие парадигм программирования

Знакомое из философии и лингвистики слово «парадигма» имеет в информатике и программировании узкопрофессиональный смысл. Парадигма программирования как исходная концептуальная схема

постановки проблем и их решения является инструментом грамматического описания фактов, событий, явлений и процессов, возможно, не существующих одновременно, но интуитивно объединяемых в общее понятие.

Наиболее распространённая практика прикладного программирования на основе императивного управления и процедурно-операторного стиля построения программ получила популярность более пятидесяти лет назад в сфере узкопрофессиональной деятельности специалистов по организации вычислительных и информационных процессов. Специалисты в области теоретического и экспериментального программирования исследовали значимость и фундаментальность формальных моделей в программировании и эффективных средств их реализации. Последние десятилетия резко расширили географию информатики, распространив ее на сферу массового общения и досуга. Это меняет критерии оценки информационных систем и предпочтения в выборе средств и методов обработки информации.

Прикладное программирование подчинено проблемной направленности, отражающей компьютеризацию информационных и вычислительных процессов численной обработки, исследованных задолго до появления ЭВМ. Именно здесь быстро проявился явный практический результат. Естественно, в таких областях программирование мало чем отличается от кодирования: для него, как правило, достаточно операторного стиля представления действий. В практике прикладного программирования принято доверять проверенным шаблонам и библиотекам процедур, избегать рискованных экспериментов. Ценится точность и устойчивость научных расчетов. Язык Fortran – ветеран прикладного программирования, постепенно стал несколько уступать в этой области языкам Pascal и C, а на суперкомпьютерах – языкам параллельного программирования, таким как Sisal. Теперь к ним присоединяются нововведения в Java, C#, Scala.

Теоретическое программирование нацелено на наглядность и сопоставимость представимых в публикациях результатов научных экспериментов в области программирования и информатики. Формальные модели унаследовали основные черты родственных математических понятий и утвердились как алгоритмический подход в информатике. Стремление к доказательности построений и оценка их эффективности, правдоподобия, правильности, корректности и других формализуемых отношений на схемах и текстах программ послужили основой структурного программирования и других методик достижения надежности процесса разработки программ, например, предложенное Д. Кнутом «грамотное

программирование». Стандартные подмножества Алгола и Паскаля, послужившие рабочим материалом для теории программирования, сменились более удобными для экспериментирования аппликативными языками, такими как ML, Miranda, Scheme, Haskell, Real, C-light и другие.

Экспериментальное программирование сформировалось при исследовании задач системного программирования, искусственного интеллекта и освоения новых горизонтов в информатике. Абстрактный подход к представлению информации, лаконичный, универсальный стиль построения функций, ясность обстановки исполнения для разных категорий функций, свобода рекурсивных построений, доверие интуиции математика и исследователя, уклонение от бремени преждевременного решения непринципиальных проблем распределения памяти, отказ от необоснованных ограничений на область действия определений – все это увязано Джоном Маккарти в идее универсального языка Lisp. Продуманность и методическая обоснованность первых реализаций Lisp-а позволила быстро накопить опыт решения новых задач, подготовить их для прикладного и теоретического программирования, что привело к кристаллизации идей функционального программирования (ФП). В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач и виды технических средств.

В наши дни признают в качестве основных парадигм программирования: ИП, ФП, логическое программирование (ЛП) и объектно-ориентированное программирование (ООП) – и отмечают образовательное значение таких базовых парадигм программирования как ФП, параллельное программирование, ИП, рассматривая ЛП и ООП как расширения базовых парадигм.

Важно отметить, что основные ПП по-разному проявляют себя на разных классах задач, и выбор ПП влияет на трудоёмкость программирования, успешность технологии программирования и надёжность применения программ.

1.5. Эксплуатационная прагматика

Отметим ключевые положения эксплуатационной прагматики:

– критерии качества программ могут изменяться в процессе программирования, что возможно повлечёт существенный пересмотр многих ранее принятых решений;

– развитие систем представления знаний сводится к чередованию фаз восходящего и нисходящего уточнения информации, что влияет на выбор технологии программирования;
– степень изученности решаемых задач – ведущий фактор прогноза трудоёмкости достижения работоспособной программы, пригодной для неавторского применения;
– технологии программирования обеспечивают гарантированное получение результата разработки при условии соответствия технологическим требованиям;
– тестирование и отладка программы требуют специального обустройства, средства которого за редким исключением не представлены в ЯП, но могут содержаться в СП;
– сходимость ЖЦП обеспечивается грамотным выбором соответствия степени изученности решаемой задачи и схемы ЖЦП, допускающей при подходящей технологии выполнение отладки программы, удовлетворяющей заданному критерию качества;
– ранг работоспособности реализованных решений зависит от полноты и универсальности программных компонент, созданных при разработке программы.

Выводы

1. Выбор ПП влияет на успех ТП и трудоёмкость ПЖЦП, а также на повышение изученности решаемой задачи, включая методы её решения, на организованность, работоспособность и живучесть разрабатываемой программы.

2. ПП проявляется в удобстве уточнения постановки задачи, метода её решения, декомпозиции программы на фрагменты, модули, аспекты или компоненты и представления принятых в проекте решений в терминах лексикона программирования и ЯСП, поддерживающих данную ПП.

3. Классификация ЯСП по поддерживаемым ПП требует анализа их операционной семантики и реализационной прагматики.

ЛЕКЦИЯ 2. ПОДДЕРЖКА ПАРАДИГМ ПРОГРАММИРОВАНИЯ

Задача этой лекции – показать, каким образом определение языка программирования (ЯП) и его реализация в системе программирования (СП) поддерживают разные парадигмы программирования (ПП). Рассмотрим средства и методы, используемые при определении языков программирования и их расширение при реализации систем программирования. Для примеров используется материал языка Pure Lisp и структурного программирования на языке Pascal.

Обычно определение ЯП задается раздельно на уровнях лексики, синтаксиса, семантики и прагматики. Реализация СП для определённой семантики ЯП может быть выполнена как интерпретатор или компилятор или и то и другое совместно. Компиляция может выполняться как для всей программы полностью, так и раздельно для указанных функций/процедур, в том числе «на лету».

Лексически близкие ЯП могут привлекать удобочитаемостью на первых шагах отладки программ.

Новые ЯП обычно избегают синтаксически сложно распознаваемых конструкций и сразу ориентируются на автоматическое построение распознавателей. Это способствует уменьшению числа ошибок при подготовке программ. Вариации лексики и синтаксиса влияют на удобочитаемость программ и распознавание принадлежности текста ЯП. Синтаксически подобные ЯП могут быть удобны при подготовке и преобразовании программ при их переносе на другие системы.

Таблица 2

Синтаксис бестипового учебного концентра языка Pascal[3]

БНФ	*Пояснение*
ident = letter {letter | digit}	Идентификатор
integer = digit {digit}	Целое
factor = ident | integer | "(" expression ")". expression = factor [("=" | "<" | "+"|"-" | "*" | "DIV") factor	Выражение: = < – Отношения между числами + - * DIV – Операции над числами
statement = [assignment | ProcedureCall | IfStatement | WhileStatement]	Оператор
assignment = ident ":=" expression	Присваивание
StatementSequence = statement {";" statement}	Последовательность исполнения

[3] Понятие «тип данных» будет привлечено позднее.

	команд
IfStatement = "IF" expression "THEN" StatementSequence ["ELSE" StatementSequence] "END"	Ветвление
WhileStatement = "WHILE" expression "DO" StatementSequence "END"	Цикл
IdentList = ident {"," ident}. FormalParameters = "(" [IdentList {";" = ["VAR"] IdentList }] ")". ProcedureHeading = "PROCEDURE" ident [FormalParameters]. ProcedureBody = declarations ["BEGIN" StatementSequence] "END" ident. ProcedureDeclaration = ProcedureHeading ";" ProcedureBody. declarations = ["CONST" {ident "=" expression ";"}] ["VAR" {IdentList;"}]	Определение функции
ActualParameters = "(" [expression {"," expression}] ")" . ProcedureCall = ident [ActualParameters]	Вызов функции
module = "MODULE" ident ";" declarations ["BEGIN" StatementSequence] "END" ident "."	

Ряд ЯП представляет программы непосредственно как структуры данных, используемые при представлении и выполнении программы. В таком случае синтаксис выглядит проще. Используется его конкретизация для семантической сопоставимости с другими ЯП.

Таблица 3

Синтаксис учебного концентра языка Lisp

Конкретизация синтаксиса для представления семантики	*Пояснение*
форма ::= переменная | (QUOTE S-выражение) | (COND {(форма форма)} | (функция {аргумент})	Переменная Константа Ветвление Вызов функции
аргумент ::= форма	
переменная ::= идентификатор	
функция ::= название | (LAMBDA список_переменных форма) | (FUNC название функция)	Имена, включая операции над списками Безымянная функция Именование функции
список_переменных ::= ({переменная})	Любое число, возможно ни одного
название ::= идентификатор	
идентификатор ::= атом	
Собственно синтаксис и лексика	
S-выражение ::= атом | (S-выражение . S-выражение) | ({S-выражение})	Атом Консолидация Список
атом ::= БУКВА {БУКВА | ЦИФРА}	

Исключение «синтаксического сахара» в ЯП обеспечивает внутреннее понимание сущности реализуемого алгоритма при подготовке и отладке программы автором.

Определение лексики и синтаксиса ЯП неявно связано с реализационной прагматикой (РП) той или иной ПП, поэтому без их детального рассмотрения переходим к анализу особенностей семантики и прагматики ЯП.

2.1 Семантика

Проблема определения ЯП и СП наиболее тщательно проработана в Венской методике определения языков программирования. Эта методика разработана в конце 60-х годов. Основная идея – использование абстрактного синтаксиса (АС) и абстрактной машины (АМ) при определении семантики языка программирования. Конкретный синтаксис

(КС) языка отображается в абстрактный, а абстрактная машина может быть реализована с помощью конкретной машины (КМ), причем и отображение и реализация могут иметь небольшой объем и невысокую сложность.

Диаграмма	*Пояснение*
КС ↔ АС → АМ → КМ	Существует отображение конкретного синтаксиса в абстрактный и обратно. Абстрактный синтаксис отображается в абстрактную машину. Абстрактная машина реализуется с помощью конкретной машины.

Рис. 1. Схема декомпозиции определения ЯП по Венской методике

Любое определение анализа текста программы выглядит как перебор распознавателей, передающих управление композициям из селекторов, достаточно определить набор распознавателей, выявляющих эти понятия, и селекторов, выделяющих их характеристики. Селекторы имеют смысл лишь при истинности соответствующего распознавателя.

Таблица 4

Основные конструкции ядра языка Pascal над целыми числами

X	Переменные
123	Константы
C	
(A1 + A2)	Вычисления
(A1 = A2)	Отношения
;	Пустой оператор
X := a	Присваивание
S1; S2	Последовательные операторы
if p then ST else SF	Ветвление
while P do S	Цикл
var X	Объявление переменной
const C = 123	Объявление именованной константы
proc Pr (V…) S	Определение процедуры
Pr (A…)	Вызов процедуры

Таблица 5

Основные конструкции ядра Pure Lisp над списками[4]

X	Переменная
(quote C)	Константой может быть любое символьное выражение
(atom X)	Проверка символьного выражения на неделимость
(eq X Y)	Совпадение адресов
(cond (P ST)…(T SF))	Ветвление
(lambda (X …) E…)	Безымянная функция
(defun F E)	Именование функции
(F A …)	Вызов функции

Унификация значений и функций позволяет в базовые средства не включать именование функций, рассматривая такой механизм как вспомогательную семантику «Категории объектов», которые появятся в более полном определении ЯП. Формально Defun можно свести к передаче параметра с помощью Lambda.

Следует принять во внимание, что ЯП с общей абстрактной машиной (АМ) равномощны по пространству порождаемых процессов.

Таблица 6

Абстрактный синтаксис ядра языка Pascal над целыми числами представлен в форме списков

X	(value X)
123	(value 123)
C	(value C)
var X	(var X)
const C = 123	(const C 123)
(A1 + A2)	(sum A1 A2)
(A1 = A2)	(eq A1 A2)
;	(empty) или (NIL)

[4] Может сам работать как свой АС.

X := a	(set X A)
S1; S2;	(step S1 S2)
if p then ST else SF;	(if P ST SF)
while p do S;	(while P S)
proc Pr (V…) S	(let Pr (V …) S)
Pr (A…)	(Pr (A…))

Использование списков в качестве абстрактного синтаксиса позволяет распознаватели разных конструкций (var, const, sum, eq, empty, assign, step, if, while) свести к анализу головы списка, что снимает вопрос конструирования или разработки анализатора.

Все селекторы (X,C,A1,A2,A,S1,S2,ST,SF,P,S) сводятся к композиции шагов доступа, выполняемых после соответствующего распознавателя. Такие определения, сводящиеся к выбору 2-го, 3-го или 4-го элементов списка, практически не требуют отладки, работают с первого предъявления.

Конструкцию while можно в базовую семантику (БС) не включать, т.к. она сводима к func. Можно выделить вспомогательную семантику (ВС) «Представление итераций», которое в полном ЯП будет включать и другие форматы циклов.

ЯП с общим АС семантически эквивалентны, они сравнимы по трудоёмкости отладки программ.

Различают два стиля задания семантики ЯП – семантика вычисления значений и семантика изменения состояний памяти. Основой определения интерпретатора для семантики вычисления значений является функция EVAL (evaluation), вычисляющая произвольные выражения языка с учетом состояния таблицы атомов TA, устроенной как стек. Для семантики изменения состояний памяти функция EXEC (execution) выполняет ту же работу на базе двух устроенных как векторы таблиц TA и TF, раздельно хранящих значения переменных и определения процедур, соответственно.

Универсальная функция, или интерпретация – это функция, которая может вычислять значение любой формы, включая формы, сводимые к вычислению любой заданной функции, применяемой к аргументам, представленным в этой же форме, по доступному описанию данной функции. (Конечно, если функция, которой предстоит интерпретироваться, имеет бесконечную рекурсию, интерпретация будет повторяться бесконечно.)

Рассмотрим универсальную функцию eval от аргумента expr. Это выражение, являющееся произвольной вычислимой формой языка Lisp.

Такая универсальная функция должна предусматривать основные виды вычисляемых форм, задающих значения аргументов, а также представления функций в соответствии со сводом правил языка. При интерпретации выражений учитываются следующие решения, представленные при определении АС:

- атом обычно понимается как переменная. Для него следует найти связанное с ним значение. Например, могут быть переменные вида x, elem, смысл которых зависит от контекста, в котором они вычисляются;
- константы, представленные как аргументы функции QUOTE, можно просто извлечь из списка ее аргументов. Например, значением константы (QUOTE T) является атом T, обычно символизирующий значение «истина»;
- условное выражение требует специального алгоритма для перебора предикатов и выбора нужной ветви;
- остальные формы выражений рассматриваются по общей схеме как список из функции и ее аргументов. Обычно аргументы вычисляются, а затем вычисленные значения передаются функции для интерпретации ее определения. Так обеспечивается возможность писать композиции функций;
- если функция представлена своим названием, то среди названий различаются имена встроенных элементарных функций, такие как CAR, CDR, CONS и т.п., и имена функций, введенных в программе;
- для встроенных функций интерпретация сама «знает», как найти их значение по заданным аргументам, а для введенных в программе функций использует их определение, которое находит подобно переменной по имени в таблице атомов – в контексте[5];
- если функция построена с помощью λ-конструктора, то, прежде чем её применять, понадобится связывать переменные из λ-списка параметров со значениями аргументов;
- если представление функции начинается с DEFUN, то понадобится сохранить имя функции с соответствующим ее определением так, чтобы корректно выполнялись рекурсивные вызовы функции. Определения функций накапливаются в «хвосте» системной переменной TA, то есть работают как глобальные определения.

[5] Похоже на резервированные слова, но в реальных СП все функции (более тысячи) равноправно являются встроенными функциями.

Таким образом, интерпретация выражений осуществляется как взаимодействие четырех ВС:

- обработка структур данных (cons, car, cdr, atom, eq);
- конструирование функциональных объектов (lambda);
- идентификация объектов (список параметров, set, defun);
- управление логикой вычислений и частичными вычислениями (композиции, quote, cond, eval).

Идентификация и управление отчасти привязаны к синтаксическим позициям без специальной лексики («синтаксический сахар»). В большинстве ЯП аналоги первых двух подсистем нацелены на обработку элементарных данных и конструирование составных значений, кроме того, иначе установлены границы между подсистемами.

Явное определение универсальной функции позволяет достичь четкости механизмов обработки Lisp-программ.

При написании на демонстрационном подмножестве Lispa определения функции EVAL согласно приведенной выше спецификации, задаётся переход от данного списочного представления выражения E к его значению с учетом заданной таблицы атомов TA, хранящей значения атомов.

Универсальная функция exec для минимального подмножества языка Pascal немного отличается:

- атом понимается как переменная или константа;
- константы строятся из так называемых литералов и могут быть поименованы;
- условное выражение состоит из предиката и двух позиций для выбора нужной ветви;
- существуют специальные формы для циклов, определения и вызова процедур и функций и другие схемы управления вычислениями;
- выражения обрабатываются с учетом таблицы приоритетов операций и расстановки скобок;
- определения переменных и функций или процедур хранятся раздельно.

Основные различия:

- exec не поддерживает безымянные функции;
- eval не поддерживает присваивания переменным;
- exec поддерживает раздельное хранение значений и определений функций/процедур;
- eval допускает конструирование определений функций в процессе вычислений.

2.2. Абстрактная машина

Особенности процесса компиляции достаточно сложны даже для простых языков, поэтому спецификация результата компиляции часто задается в терминах языково-ориентированных абстрактных машин (АМ). Система команд АМ представляет собой реализацию БС, дополненную рядом системных действий по передаче параметров и защите областей действия, подразумеваемых ЯП, но не имеющих чёткого синтаксического представления. Такое определение может быть машинно-независимым и переносимым.

АМ различает обычно следующие категории команд:

– засылка значений из памяти в стек;
– вычисления над безымянными операндами в стеке при обработке выражений;
– пересылка значений из стека в память с учетом локализации;
– организация ветвлений и циклов;
– организация вызовов процедур и функций с сохранением/восстановлением контекста.

Могут быть и другие категории команд.

При сравнении императивного и функционального подходов к программированию, П. Лэндин (P.J. Landin) предложил специальную абстрактную машину SECD, удобную для спецификации машинно-зависимых аспектов семантики Lispa. Подробное описание этой машины можно найти в книге Хендерсона по функциональному программированию [15].

Машина SECD работает над четырьмя регистрами: стек для промежуточных результатов, контекст для размещения именованных значений, управляющая вычислениями программа, резервная память (Stack, Environment, Control list, Dump). Регистры приспособлены для хранения выражений в форме атомов или списков. Состояние машины полностью определяется содержимым этих регистров. Поэтому функционирование машины можно описать достаточно точно в терминах изменения содержимого регистров при выполнении команд, что выражается следующим образом:

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

Встраиваемые в ядро интерпретатора операции должны соответствовать стандартным правилам доступа к параметрам и размещения выработанного

результата. Таким же правилам должен подчиняться и компилируемый код. Это позволяет формально считать равноправными встроенные и программируемые функции. Компилятор по исходному тексту программы строит код программы, эквивалентный тексту.

Для характеристики встроенных команд интерпретатора и результата компиляции программ базового Lisp-а понадобятся следующие команды:

LD – ввод данного из контекста в стек;
LDC – ввод константы из программы в стек;
LDF – ввод определения функции в стек;
AP – применение функции, определение которой уже в стеке;
RTN – возврат из определения функции к вызвавшей ее программе;
RAP – применение рекурсивной функции
DUM – резервирование памяти для хранения аргументов рекурсивной функции.
SEL – ветвление в зависимости от активного (верхнего) значения стека;
JOIN – переход к общей точке после ветвления;
CAR – первый элемент из активного значения стека;
CDR – без первого элемента активное значение стека;
CONS – формирование узла по двум верхним значениям стека;
ATOM – неделимость (атомарность) верхнего элемента стека;
EQ – равенство двух верхних значений стека;
STOP – останов.

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

$$s\ e\ (STOP)\ d \rightarrow s\ e\ (STOP)\ d$$

Следуя Хендерсону, для четкого отделения обрабатываемых элементов от остальной части списка будем использовать следующие обозначения:

– (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.

Теперь мы можем методично описать эффекты всех перечисленных выше команд.

s e(LDC q . c)d → (q . s) e c d
(a b . s)e(CONS . c)d → ((a . b) . s) e c d
((a . b) . s)e(CAR . c) → (a . s) e c d
((a . b) . s)e(CDR . c) → (b . s) e c d

Для предиката оговариваем, в каких случаях вырабатывается значение «истина».

((a . b) . s)e(ATOM . c)d → (NIL . s) e c d
(A . s)e(ATOM . c)d → (T . s) e c d

Для неатомарных значений – NIL, , т. е. ложь, истина «T» - для атомов,

(a a . s)e(EQ . c)d → (T . s) e c d
(a b . s)e(EQ . c)d → (NIL . s) e c d

Истина «T» – для совпадающих указателей, для несовпадающих – NIL, т. е. ложь.

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

s e (LD n . c) d → (x . s) e c d

x – это значение (N-th n e).

При реализации ветвлений управляющая программа соответствует следующему шаблону:

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

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

(NIL . s) e (SEL c1 c0 . c) d → s e c0 (c . d)
(T . s) e (SEL c1 c0 . c) d → s e c1 (c . d)
s e (JOIN) (c . d) → s e c d

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

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

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

Рекурсивные вызовы функций требуют резервирования памяти для уникальных ссылок на разные поколения фактических аргументов, что достигается путем размещения фиктивного объекта «ω», который функция rplaca, в порядке исключения, замещает на реальные данные:

s e (DUM . c) d → s (ω . e) c d
((f . ef) vf . s) e (RAP . c) d → NIL (rplaca vf ef) f (s e c . d)

Для SECD реализационное замыкание ядра включает в себя структуро-разрушающие функции rplaca и rplacd, размещающие свой результат непосредственно в памяти второго аргумента. Это требует соответствующих ветвей в определениях синтаксиса и интерпретатора, что можно рассматривать как шаг раскрутки СП.

Таблица 7

Реализационное дополнение Pure Lisp

	Новая конструкция	*Значение*	*Примечание*
Семантика	(RPLACA x (y . z))	(x . z)	Добавление новых операций над данными
	(RPLACD x (y . z))	(y . x)	

Таким же образом система команд АМ может быть расширена простым дополнением правил. Так, можно её механизм распространить на другие типы данных, например, на целые числа:

(a b . s)e(SUM . c)d → ((a + b) . s) e c d

(a b . s)e(DIF . c)d → ((a - b) . s) e c d

(a b . s)e(MUL . c)d → ((a * b) . s) e c d

(a b . s)e(DIV . c)d → ((a / b) . s) e c d

(a . s)e(ADD1 . c) → ((a + 1) . s) e c d

(a . s)e(MIN1 . c) → ((a - 1) . s) e c d

(a . s)e(NUMBER . c) → (t . s) e c d

t – истинностное значение NIL или T.

Соответствующее расширение ЯП может быть выполнено на уровне лексики, синтаксиса и семантики следующим образом.

Таблица 8

Расширение Pure Lisp и SECD средствами обработки чисел.

	Новая конструкция	*Значение*	*Примечание*
Лексика	Number = digit {digit}	целое	Новая область данных
Синтаксис	атом = Number		Встраивание новой области в систему понятий ЯП
Семантика	(SUM x y)	(x + y)	Добавление операций над новыми данными
	(DIF x y)	(x - y)	
	(MUL x y)	(x * y)	

	(DIV x y)	(x / y)	
	(ADD1 x y)	(x + 1)	
	(MIN1 x y)	(x - 1)	
	(NUMBER x)	NIL или T	Предикат, выделяющий числа

Аналогичная машина для языков Pascal и Oberon содержит следующие команды:

LD – ввод данного из контекста в стек промежуточных значений;
LDC – ввод константы из программы в стек;
LDP – ввод определения процедуры;
LDW – ввод числа из памяти в стек;
STW – сохранение значения в глобальной памяти;
STE – сохранение значения в локальном контексте;
RET – возврат из процедуры к вызвавшей ее программе;
IF – ветвление в зависимости от активного (верхнего) значения стека;
EQ – равенство двух верхних значений стека;
LT – верхнее значение в стеке меньше, чем второе;
INC – увеличение числа на 1;
DEC – уменьшение числа на 1;
SUB – вычитание из верхнего элемента стека;
ADD – прибавление к верхнему элементу стека;
MUL – произведение двух чисел;
DIV – частное от деления верхнего элемента стека на второй элемент;
BR – безусловная передача управления по абсолютному адресу;
AP – применение процедуры к аргументам, расположенным в стеке;
STOP – останов.

Машина SECM, как и SECD, работает над четырьмя регистрами: стек для промежуточных значений, контекст для размещения аргументов, локальных значений переменных и регистра возврата, управляющая вычислениями программа, память (Stack, Environment, Control program, Memory). Регистры приспособлены для хранения данных в форме векторов или списков, не пересекающихся по фактическим адресам. Состояние машины полностью определяется содержимым этих регистров. Поэтому функционирование машины можно описать достаточно точно в терминах изменения содержимого регистров при выполнении команд, что выражается следующим образом:

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

Теперь мы можем методично описать эффекты всех перечисленных выше команд.

```
s e (LDC q . c)m                → (q . s) e c m
(a . s) e (INC . c)m            → (a+1 . s) e c m
(a . s) e (DEC . c)m            → (a-1 . s) e c m
(a b . s) e (ADD . c)m          → (a+b . s) e c m
(a b . s) e (SUB . c)m          → (a-b . s) e c m
(a b . s) e (MUL . c)m          → (a*b . s) e c m
(a b . s) e (DIV . c)m          → (a/b . s) e c m
(a a . s) e (EQ . c)m           → (TRUE . s) e c m
(a b . s) e (EQ . c)m           → (FALSE . s) e c m
(a b . s) e (LT . c)m           → (t . s) e c m
(TRUE . s) e (IF ct cf . c) m   → s e (ct . c) m
(FALSE . s) e (IF ct cf . c)    → s e (cf . c) m
m
```

t имеет значение «TRUE» или «FALSE».

Далее используем дополнительные обозначения:

[x] – содержимое памяти по адресу x;
e[n] – содержимое n-го элемента контекста;
A(Pr) – число аргументов процедуры Pr;
L(Pr) – число локальных переменных процедуры Pr;
@c – адрес позиции «c» в программе;
_ – Произвольное значение (_ подчерк).

```
s e(LDW x . c) m                → ([x] . s) e c m
s e (LD n . c) m                → (e[n] . s) e c m
(a . s) e (STW x . c) m         → s e c (m[x]=a)
(a . s) e (STE x . c) m         → s (e[x]=a) c m
s e (LDP @Pr. c) m              → (@f L(Pr) A(Pr) . s) e c m
(@Pr KL N a1 ... aN . s) e (AP . c) m   → s ((KL+N) _ ... _ a1 ... aN @c . e) Pr m
s (K e1 ... eK p . e) (RET . c) m   → ( ) e p m
s e(BR @Pr . c) m               → s e Pr m
```

Разница между SECD и SECM в основном сводится к операциям над данными. Кроме того, SECM имеет дополнительные команды по

непосредственной работе с памятью для реализации присваиваний и передаче управления для реализации итераторов.

Обе абстрактные машины, как SECD, так и SECM, содержат, кроме образа базовых средств ЯП, дополнительные команды, обеспечивающие пересылки данных между регистрами и реализационные средства, поддерживающие эффективность реализации ЯП (rplaca, BR, метки и др.). В результате АМ способна выполнять вычисления несколько более широкого класса, чем задано БС данного ЯП. Это приводит к идее определения реализационного замыкания ЯП в соответствии с фактическими возможностями АМ. Такое замыкание выполняет роль ядра ЯП.

Предстоящие шаги инкрементального, т. е. пошагового, процесса раскрутки СП могут быть связаны с расширением типов обрабатываемых данных, подключением удобных форматов управления вычислениями, специализацией особых категорий функций, созданием программного инструментария и т. д. до полного или практически достаточного покрытия ЯП его реализацией в СП.

Дальнейшее развитие СП заключается в пошаговом расширении спектра обрабатываемых данных и присоединении ВС до полного покрытия ЯП его реализацией в СП. Каждую новую область данных подключают к АМ как комплект подпрограмм, реализующий предикат для выяснения принадлежности данного новой области и операции обработки данных. ВС, как правило, могут быть определены в терминах самого ЯП с помощью средств его ядра.

Интеграция вспомогательной семантики новых компонент ЯП в ранее реализованную версию СП осуществляется комплексным включением ряда согласованных определений на всех уровнях определения ЯП:

- дописывание БНФ;
- добавление ветвей в определение интерпретатора;
- дополнение АМ новыми командами.

В случае SECM кроме образа БС фактически реализована работа с векторами и указателями (адреса в памяти). Поэтому реализационное замыкание учебного концентра языка Pascal естественно будет включать в себя обработку структур данных. Соответствующие, фактически поддержанные на уровне АМ, правила языка:

Данные = array of

АМ: x[i] := y
X := y [i]

*x := @y доступ по указателю к адресу.

Допустимость таких операций существенно зависит от распределения памяти и независимости её частей, что приводит к необходимости контроля границ памяти при индексировании векторов и доступе по указателю. Информация для такого контроля во многих ЯП представляется в форме предписания типа данных (ТД) переменным. Таким образом, реализационное замыкание ЯП над SECM включает в себя представление ТД для всех переменных, включая параметры процедур.

Таблица 9

Встраивание нового понятия в определение синтаксиса ЯП

Новая конструкция	*Пояснение*	*Пример*
selector ={"[" expression "]"}.	Индексирование доступа к элементу вектора	[ind]
factor = ident selector	Доступ к элементу вектора	X [ind]
assignment = ident selector ":=" expression	Присваивание элементу вектора	X [ind] := 6
ArrayType = "ARRAY" expression "OF" type	Вектор как тип данных	ARRAY 10 OF integer
type = ident | ArrayType	Типы данных: целое или вектор	integer
FPSection = ["VAR"] IdentList ":" type	Раздел параметров процедуры, возможно изменяемых	VAR x,y : integer
FormalParameters = "(" [FPSection {";" FPSection}] ")"	Список формальных параметров процедуры	(a,b: integer ; VAR x,y : integer)
declarations = ["CONST" {ident "=" expression ";"}] ["TYPE" {ident "=" type ";"}] ["VAR" {IdentList ":" type ";"}]	Объявления констант, типов данных и переменных	CONST id = 2+3; TYPE arr10 = ARRAY 10 OF integer; VAR a1,a2,a3 : intger; xx,yy : arr10 ;

Дальнейшее расширение ЯП может быть сведено к подключению ТД и присоединению ВС методом раскрутки.

ЯП с общей АМ семантически равномощны, на их основе достижима сравнимая эффективность процессов вычислений.

Трудоёмкость реализации АМ с помощью конкретной машины можно оценить на основе материала по низкоуровневому программированию, а также при сравнении АМ с Пи-кодом и RISK-машиной, предложенными Н. Виртом при разработке учебных языков Pascal и Oberon, и учебными машинами MIX и MMIX, описанными Д. Кнутом.

Семантический спуск от полного ЯП к его БС характеризуется снижением трудоёмкости реализации ядра ЯП и его АМ, сопровождаемое увеличением трудоёмкости применения частичной реализации ЯП.

Трудоёмкость применения можно оценивать числом понятий, возникающих при программировании сверх тех, что присущи решению типовых задач.

Цели раскрутки:

- снижение трудоёмкости начального этапа программирования;
- увеличение потенциала СП, т.е. числа типовых задач, решение которых обладает приемлемой для практики трудоёмкостью.

Исследование разных схем частичных, смешанных, «ленивых» вычислений и метакомпиляции приводит к выводу о целесообразности совмещения таких схем в рамках общей СП с целью использования их преимуществ на разных уровнях изученности решаемых задач.

Частичные вычисления допускают прогон программы при неполном комплекте входных данных. В результате выполняются те операции, для которых имеются данные. Одновременно строится остаточная программа, которую можно выполнить с недостающими данными и получить итоговый результат, такой, как если бы все данные были заданы изначально.

Смешанные вычисления допускают произвольную разметку программы на выполнимые и задержанные действия. Выполняются маршруты, которым задержанные действия не препятствуют и выводится остаточная программа, которая может быть выполнена после снятия блокировки с задержанных действий.

«Ленивые» вычисления выполняются в зависимости от необходимости операций для получения результата с запоминанием промежуточных значений выражений для экономии повторных вычислений.

Метакомпиляция обрабатывает программу совместно с комплектом типовых данных.

Система программирования может содержать пару «интерпретатор – компилятор». На практике достоинства интерпретации проявляются при отладке программ, а преимущества компиляции – при эксплуатации готового программного продукта. Более подробное обсуждение этой темы заслуживает отдельного разговора.

Теория программирования утверждает, что определение компилятора может быть выведено из определения интерпретатора методами смешанных вычислений. Компилятор по такой методике получается как остаточная программа при смешанном вычислении интерпретатора.

2.3. Структуры данных

Кроме распределения памяти при компиляции различимы дисциплины, обеспечивающие эффективность работы с памятью в процессе исполнения программ:

- new – delete – динамические запросы к системе памяти типа «куча»;
- компактизация – уплотнение пространства с целью размещения крупных целостных объектов;
- «близнецы» – метод укрупнения памяти и профилактики её чрезмерной фрагментации при распределении на разно объемные блоки;
- стек – дисциплина доступа FILO;
- Setl – более 17-ти разных СД, динамически выбираемых СП для представления множеств в зависимости от характера их обработки и наличия свободной памяти.

Типичная реализация структур данных во многих системах программирования:

- векторы с паспортом, хранящим при компиляции сведения о размерах и типе элементов, не доступны при исполнении программы;
- запись или структура, обеспечивающая доступ к заданному перечню разносортных элементов по статически определённым ключам;
- объединение заданного перечня разных ТД, размещаемых в разное время по определённому адресу;
- множество небольшого числа перечислимых элементов, обработки которых не выводит за пределы машинных команд над битовыми кодами;
- стек, допускающий две дисциплины доступа, – FILO и вектор.

В машине списки хранятся не как последовательности символов, а как структурные формы, построенные из машинных слов как частей деревьев, подобно записям в Паскале при реализации односвязных списков.

– размер и даже число выражений, с которыми программа будет иметь дело, можно не предсказывать. Кроме того, можно не организовать для размещения выражений блоки памяти фиксированной длины;
– ячейки можно переносить в список свободной памяти, как только исчезнет необходимость в них. Даже возврат одной ячейки в список может быть полезен, но если данные хранятся линейно, то организовать использование лишнего или освободившегося пространства из блоков ячеек трудно;
– выражения, являющиеся продолжением нескольких выражений, могут быть предоставлены однократно.

В любой момент времени только часть памяти, отведенной для списков, действительно используется для хранения полезных данных. Остальные ячейки организованы в простой список, называемый списком свободной памяти.

Самым интересным, можно сказать, революционным, механизмом работы с памятью в языке Lisp, стала автоматизация повторного использования памяти - «сборка мусора». С начала 1960-х годов методам такой работы посвящены многочисленные исследования, которые продолжаются до наших дней и сильно активизировались в связи с включением похожего механизма в реализацию языка Java.

Общая идея всех таких методов достаточно проста:

– пока памяти хватает, о ней можно не беспокоиться и располагать новые данные в новых блоках памяти;
– если свободной памяти вдруг не оказалось, то надо выполнить «сборку мусора», в процессе которой, возможно, найдутся ставшие бесполезными для программы блоки;
– если память нашлась, ее снова можно тратить.

Специальная программа «Сборщик мусора» выполняет анализ достижимости всех блоков памяти простой пометкой узлов, видимых из конечного числа рабочих регистров системы программирования. К таким регистрам относятся промежуточные результаты вычислений, активная часть стека, ассоциативный список, таблица атомов и др. После пометки все непомеченные узлы объединяются в список свободной памяти, передающий их для повторного использования новым вызовам функции CONS. Такая автоматизация не лишена недостатков.

Ряд неприятных следствий: непредсказуемые длинноты (время работы) при поиске очередной порции ячеек и «перегрев системы», если такие порции слишком малы для продолжения счета. По мере роста производительности оборудования разработаны простые и не столь обременительные алгоритмы повторного использования памяти на базе параллельных процессов и профилактического копирования активных структур данных в дополнительные блоки памяти. Такие методы не требуют сложной разметки и анализа достижимости. Кроме того, почти исключается необходимость присваиваний. Они в программах заменяются именованием.

Память обычно распределена по блокам, содержащим ряд слов, образующих структуры данных. Физический объем памяти, логическая длина данных и состав информации, полезной для продолжения вычислений, могут существенно различаться. Минимизацию потерь в результативности работы с памятью дает динамическая обработка бинарных деревьев – нет простоев из-за незаполненности части полей. Каждый узел такого дерева имеет небольшой объем, достаточный для хранения двух типизированных указателей (CAR и CDR, левый и правый). Типизация указателей нужна для оперативного динамического контроля соответствия данных и операций по их обработке. NIL, атомы, списки, числа, строки – все это реализационно различимые типы данных. Утверждение о бестиповости языка Lisp устарело и в наши дни заменилось признанием полноты типового контроля в динамике исполнения программ.

Ранее *бестиповостью* называлось отсутствие статического связывания в тексте программы имен переменных с типами их значений. Для компиляции приходится дополнять Lisp-программы сведениями о типах значений свободных переменных, но далеко не каждая программа доживает до компиляции. Существуют реализации, поддерживающие автоматический вывод типов данных. Языку Lisp свойственна функциональная классификация значимых типов данных, т.е. именно реализационно различимых.

Эффективность приведенного выше определения интерпретатора с использованием ассоциативного списка существенно зависит от числа различимых атомов в программе. Такая зависимость обычно смягчается механизмом функций расстановки (хэш-функций), обеспечивающим доступ к информации по ключу. В качестве ключа используется имя атома. Число свойств атома не ограничено. Свойства бывают встроенные, системные или вводимые программистом. Значения атомов, адреса встроенных операций, определяющие выражения функций – примеры встроенных свойств.

Обычно с машиной связывается представление о блоках информации фиксированного объема, таких как слова, байты, регистры. Функциональное программирование и ООП нацелены на более крупные построения – структуры данных не ограниченной заранее длины. Такие структуры достаточно эффективно реализуются посредством специального стека, приспособленного к обработке произвольного числа компонентов текущего уровня иерархической структуры данных. От обычного стека он отличается выделением указателя на конец текущего уровня.

В результате стек хранит ссылки на границы уровней, что обеспечивает возможность возврата на любой нужный уровень, в частности, восстановления процесса обработки в случае неожиданных ситуаций.

Значительный резерв производительности функциональных программ дают деструктивные функции, являющиеся формальными аналогами чистых функций, но при выполнении сопровождаемые побочными эффектами. Такой подход позволяет при необходимости повышать эффективность программ, отлаженных в стиле без использования присваиваний, простым привлечением деструктивных аналогов функций под ответственность программиста, точно знающего границы допустимых изменений хранимых данных.

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

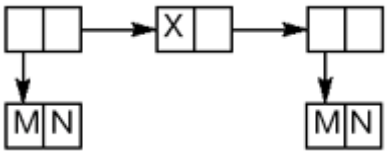
Диаграмма	*Пояснение*
	При создании структур многократные вхождения одинаковых данных получают независимое расположение в памяти.

Рис. 2. Пример ((M . N) X (M . N))

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

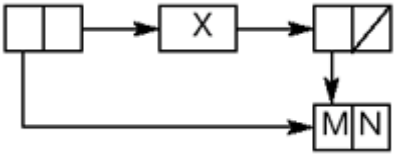
Диаграмма	*Пояснение*
	Слияние многократных вхождений одинаковых данных.

Рис. 3. Графическое представление эффективного размещения данных

Непосредственное текстовое представление в точности такой структуры невозможно, но ее можно построить с помощью одного из выражений:

```
(LET ((a '(M . N))) (SETQ b (LIST a 'X a)) )
((LAMBDA (a) (LIST a 'X a) )'(M . N))
```

2.4. Реализационная прагматика

При классификации ЯСП ключевое значение имеет семантика, но для уяснения преимуществ ПП, поддерживаемой ЯП, требуется понимание реализационной прагматики (РП), которая может быть не представлена в определении или стандарте ЯП, но подразумеваться традиционно.

Реализационная прагматика, затрагивая все уровни определения ЯП, в основном представляет решения в области конкретной организации работы с памятью, уточняющей решения и принципы, провозглашенные в определении АМ. В первую очередь это относится к вопросам защиты областей памяти и их конечности, т.е. реагирования на дефицит памяти.

Реализационная прагматика, поддерживающая разные ЯП:

ФП – списки, мусорщик, списки свойств атома;

ИП – побочные эффекты, векторы, ТД (переменная-значение);

ЛП – разностные списки, последовательный перебор, откатка;

ООП – ссылки, виртуальные и абстрактные методы и классы, множественное наследование.

ЯП с общей РП реализационно равнозначны, они сравнимы по трудоёмкости реализации СП.

Выводы

1. Поддержка ПП при определении ЯП и реализации СП выражается в реализации средств организации вычислений, механизмов обработки параметров и использования СД и их размещения в памяти, методов контроля за вычислениями и управления ходом вычислений. Успех применения ПП зависит от того, в какой мере используемые ЯСП поддерживают выбранную парадигму.

2. Лексически близкие и синтаксически подобные ЯП могут иметь существенные различия на уровне семантики и реализационной прагматики.

3. Диалекты ЯП часто бывают реализационно равнозначны, возможно семантически эквиваленты и равномощны, но различимы по эксплуатационной прагматике, лексике и синтаксису:

– учебные концентры для ознакомления с основными идеями;
– практичные подъязыки для программирования решений типовых задач;
– проблемно-ориентированные вариации для расширения сферы применения;
– полные языки для исследования и выбора улучшений.

Таблица 10

Список понятий ЯП, различаемых операционной семантикой, определяемыми для сравнения

Понятие	*Пояснение*
Атом/Скаляр/Литерал	Атомы и скаляры могут быть разных категорий или типов, различаемых динамически или декларативно.
Структура данных	Возможны ограничения на характер элементов структуры, их число и динамику их изменений.
Переменная	Может быть инициирована до вычислений, ограничена предписанным типом данных, заданным статически или выводимым по программе.
Значение	Разные типы значений связаны с различными операциями их обработки и синтаксическими позициями в тексте программы, допускающими или требующими их вхождение.
Выражение	Форма, результат которой может быть вычислен и использован как параметр в других формах.
Действие/Операция	Встроенная команда или подпрограмма, рассматриваемая как элементарная база при организации вычислений.

Условие/Логика	Концепция истинностных значений может требовать как специального типа данных, так и рассматриваться как нагрузка обычных значений (0, NIL).
Функция[6]	Возможно параметризованный фрагмент программы, представляющий укрупнённую единицу, используемую наравне с операциями при организации вычислений.
Аргумент	Фактический параметр используемой функции.
Вызов функции	Форма, используемая для исполнения функции при заданных параметрах.
Определение функции	Форма, представляющая фрагмент программы, предназначенный для использования в качестве функции.
Идентификатор/Имя	Уникальная форма, создаваемая как синоним многократно используемого элемента данных.

При неформальной характеристике стиля программирования отмечаются различия в акцентах при ответе на следующие вопросы:

1. В чем заключается основной метод обработки программы при отладке?
2. Что показывает результативную активность программы?
3. Когда принимается решение о продолжении незавершённых вычислений?
4. В каких пределах планируется функционирование участков повторяемости?
5. Каким способом гарантирована корректность сложной информационной обработки?

Варианты ответов:

1. Основные методы обработки программы при отладке:
   – интерпретация текста программы, приводящая к результату её выполнения;
   – интерпретация структуры программы, приводящая к результату её выполнения;
   – компиляция текста программы, приводящая к коду программы, выполнение которого дает результат;
   – сборка кода программы из готовых типовых компонентов;
   – редактирование заранее подготовленных шаблонов;
   – генерация кода по верифицированной спецификации цели программы.

[6] Операторы управления, процедуры, макросы и т. п. рассматриваются как отдельные категории функций – укрупнение действий.

2. Результативную активность программы показывает:
- изменение состояния отдельных элементов памяти;
- вычисление значения выражения;
- протокол обмена данными между программой и пользователем;
- изображение хода вычислений в виде диаграммы.

3. Решение о продолжении незавершённых вычислений принимается на основе:
- наблюдения за обработкой прерываний;
- получения диагностических сообщений;
- анализа результатов повторного прогона программы;
- подготовки обработчиков прерываний.

4. Функционирование участков повторяемости планируется:
- пока имеется свободная память;
- когда задано максимальное число повторений тела цикла или функции;
- если известна временная граница для выполнения любой команды, включая вызов процедуры.

5. Корректность сложной информационной обработки гарантируют следующие механизмы:
- статическая проверка соответствия типа данных переменных и операций;
- защитные условия и инварианты циклов;
- динамический контроль типов значений и допустимости операций;
- верификация программ на моделях;
- конструирование программ, корректных по построению.

Описание концептуального языка или ядра ЯП, приспособленное для его сравнения с другими языками, содержит следующие части:
- список общих понятий показывает уровень сопоставимости сравниваемых ЯП;
- АС позволяет оценить глубину проработки используемых понятий;
- АМ показывает масштаб переносимости СП;
- РП дает оценку трудоёмкости реализации СП;
- схема интерпретации или компиляции ЯП – даёт показатель организованности процесса вычислений;

– примеры программ решения типовых задач для иллюстрации концепции ЯП позволяют продемонстрировать парадигматические вариации в решении типовых задач.

Кроме того, эксплуатационная прагматика представляет экспертную оценку требований к условиям применения ЯСП и критериев успешности результата программирования, что дает основания для рекомендаций по выбору ЯП, поддерживающего требуемую ПП.

2.5. Определитель парадигм

Первые языки программирования обладали машинной ориентированностью и поддерживали принципиально важную, но небольшую по длительности и трудозатратам часть ЖЦП – от 2-х до 5 %, заключающуюся в кодировании готовых алгоритмов в терминах автоматов. Появление ЯВУ расширило языковое покрытие ЖЦП примерно до 10 % для хорошо поставленных задач, имеющих алгоритмы решения над типовыми структурами данных, причём алгоритмы приспособлены для нисходящих методик программирования.

Парадигма функционального программирования посягнула на ЖЦП для задач с исследовательским компонентом и расширила его языковое покрытие до 50 % благодаря механизмам хранения и накопления информации о свойствах информационных объектов, полезной при отладке и модификации программ, составленных из небольших универсальных компонент, допускающих как нисходящую, так и восходящую методику разработки.

Логическое программирование распространило эти механизмы на не вполне определенные постановки задач, что дало языковую поддержку предварительному сбору фактического материала, созданию демонстрационных версий, пробному прототипированию, отчасти тестированию и довело общее языковое покрытие ЖЦП почти до 60 % при восходящей методике разработки.

Появление объектно-ориентированного программирования смягчило итеративность ЖЦП для задач, связанных с развивающимися областями приложения, что довело языковое покрытие примерно до 80 %.

Определитель парадигмы языка программирования содержит следующие процедуры:

– разложение языка на фрагменты по уровням/концентрам и слоям с целью выделения базовых средств языка и его реализационного ядра – семантический базис;

– декомпозиция семантического базиса языка на основные семантические системы с минимизацией их сложности и, возможно, их описание относительно концептуальных языков;
– определение АМ языка и интерпретатора, формально достаточного для построения расширений, эквивалентных исходному языку – нормализованное определение;
– сравнение полученного определения с описаниями известных парадигм и концептуальных языков;
– сбоснование выводов относительно парадигмы исследуемого языка;
– фразеологический словарь ПП, используемый при определении ЯП;
– определение уровня языка и его ниши в жизненном цикле программ и деятельности программистов (цели и задачи), базовых языков, использованных при его создании и реализации, как основы для рекомендаций по выбору и применению ЯП и его СП.

Примеры представления результатов парадигматического анализа языков программирования можно выразить в табличной форме.

Таблица 11

Парадигматическая характеристика Pure Lisp

Параметр	*Описание*
Эксплуатационная прагматика ЯП[7]	Язык учебного назначения, созданный специально для изучения методов функционального программирования на языке Lisp, успешно применяется при решении задач с исследовательским компонентом, требующих быстрой отладки.
Особенности системы понятий	Все понятия сведены к разным категориям понятия «функция», унифицированы представления функций и значений, выполнение программы рассматривается как отображение списка аргументов в результат.
Перечень понятий, распознаваемых на уровне абстрактного синтаксиса	Символьное выражение (S-выражение), атом, вычислимая форма, переменная, константа, ветвление, элементарные функции, определение функции, именование функции, применение функции, аргумент функции, значение.
Базовые средства ЯП	NIL, CONS, CAR, CDR, ATOM, EQ, QUOTE, EVAL, COND, LAMBDA, DEFUN.

[7] Уровень языка, его ниша в полном жизненном цикле программ, соответствующая технология.

Семантические расширения[8]	Работа с числами и строками рассматривается как вспомогательная семантика. Ввод и вывод данных считаются псевдо-функциями, обслуживающими отладку. Отдельное расширение языка поддерживает функции с пост-вычислением аргументов – специальные функции.
Регистры абстрактной машины	Стеки для хранения результатов, локальных значений переменных, вычисляемой формы и дампа, обеспечивающего защиту контекста вычислений (S E C D)
Категории команд абстрактной машины	Засылки в стек, вычисления над стеком, пересылки из стека, ветвления, применение функции, выходы из ветвлений и функций, восстановление контекста, поддержка рекурсии.
Реализационная прагматика [9]	Списки из бинарных узлов, содержащих пару тэгированных указателей. Тэг показывает тип данного, адресуемого указателем. Автоматизировано освобождение памяти служебной программой «Сборщик мусора».
Парадигматическая специфика [10]	Исторически основополагающий классический язык, поддерживающий функциональное программирование в полном объёме.

Ещё один пример такой же компактной парадигматической характеристики.

Таблица 12

Парадигматическая характеристика ядра языка Pascal, поддерживающего методику структурного программирования

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Язык учебного назначения, созданный для обучения студентов методам программирования решений задач, готовых для эффективной реализации при поддержке автоматным моделированием.
Особенности системы понятий	Программа и данные – раздельные сущности. Выполнение программы сводится к шагам изменения состояний памяти, хранящей данные. Используемые в программе идентификаторы подчинены иерархии

[8] Вспомогательные семантики, их описание относительно концептуальных языков.

[9] Структуры данных и традиционно подразумеваемые механизмы реализации

[10] Роль языка в формировании поддерживаемой им парадигмы и/или перечень поддерживаемых парадигм.

	областей видимости, задаваемых вложенностью определений процедур и функций. Процедуры и типы данных не являются значениями. Возможно конструирование новых типов данных и приведение типа данных к заданному.
Перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Скаляр, вектор, значение, ключевые слова, константа, переменная, тип данных, операция, выражение, операнд, элемент вектора, сравнение, действие, последовательность действий, ветвление, цикл, определение процедуры/функции, вызов процедуры/функции.
Базовые средства ЯП.	TRUE, FALSE, перечислимые значения, :=, a[i], IF, WHILE, PROCEDURE, st1 ; st2 .
Семантические расширения	Разные виды чисел, записи, множества, указатели, метки, передачи управления, функции, переключатели, конструирование типов данных, ввод-вывод, файлы, строки, библиотеки.
Регистры абстрактной машины	Стеки промежуточных результатов, локальных переменных и параметров, выполняемой процедуры и вектор памяти (S E C M).
Категории команд абстрактной машины	Засылки в стек, вычисления над стеком и памятью, пересылки из стека и локальных переменных, ветвления, передачи управления, вызовы процедур.
Реализационная прагматика	Память распределяется статически по блокам заданного размера, обработка векторов использует вычисление смещений от базового адреса и подразумевает контроль границ отведенной под вектор памяти, при необходимости привлекаются библиотеки, поддерживающие ввод-вывод, работу с файлами и динамической памятью.
Парадигматическая специфика	Прецедент строго определения языка программирования, обладающего не слишком высокой сложностью в реализации, поддерживающего методику результативного структурного программирования. Может выполнять роль эталонного монопарадигматического языка при сравнительном анализе языков и определении их парадигматической характеристики.

Средства и методы программирования складывались на фоне быстрого расширения сферы применения компьютерных технологий и стремительного взлёта эффективности элементной базы. Наряду с

расслоением парадигм программирования в зависимости от глубины и общности технических решений по организации процессов обработки данных происходит их интеграция в рамках новых языков программирования, всё более полно поддерживающих жизненный цикл программ. В результате обнаружились явные сложности классификации языков программирования и определения их принадлежности конкретной парадигме программирования.

Рассмотрение технических особенностей языков и систем программирования через их парадигматическую характеристику может дать достаточно лаконичные формы их определения относительно ряда простых концептуальных языков.

ЛЕКЦИЯ 3. ЯЗЫКИ НИЗКОГО УРОВНЯ

Методика парадигматической характеристики ЯП иллюстрируется на материале языков низкого уровня (ЯНУ). Программирование или кодирование на ЯНУ ассоциируется с одноуровневыми структурами данных, обусловленными архитектурой и оборудованием[11]. При хранении данных и программ используется общая глобальная память с произвольным доступом. В принципе достижима предельная эффективность программ, но их отладка осложнена сочетанием «низкий старт – высокий финиш». Иными словами, легко достичь успеха в первых упражнениях, но трудно создать программный продукт и обеспечить его квалифицированное сопровождение. Для ЯНУ характерна однозначность соответствия между программой и процессом, порождаемым при ее реализации. Поэтому анализ операционной семантики ЯНУ можно выполнить на уровне абстрактной машины (АМ), вполне определяющей свойства программ и процессов, подготовленных с помощью ЯНУ. Как правило, при определении абстрактной машины ЯНУ достаточно трех регистров, назначение которых соответствует реализации понятий «результат», «программа», «память» или «результат», «контекст», «программа».

Традиционно к ЯНУ относят машинно-зависимые языки ассемблера, макропроцессоры, машинно-ориентированные языки, языки управления процессами. Для таких ЯНУ характерно, что все действия в программе выражены явно. Программа – произвольная смесь команд, соседство которых практически не ограничивается. Доступны любые фрагменты данных и программ. Предопределены все базовые средства по представлению значений и структур данных в памяти и схема управления их обработкой, что позволяет четко относить ЯНУ к конкретной парадигме.

Представляют интерес и другие подходы к машинно-ориентированному эффективному программированию. Язык Forth – пример организации вычислений над стеком. Его можно рассматривать как язык-ядро с возможностью практически неограниченного проблемно-ориентированного расширения.

Операционная семантика ЯНУ обычно содержит целочисленную арифметику, ограниченную разрядностью адресов или машинных слов, использует работу с общими, глобальными идентификаторами, поддерживает последовательное управление вычислениями и осуществляет организацию структур данных по принципу соседства элементов,

[11] Ассемблер «Эльбрус» и автокод «Инженер» – контрпримеры, показывающие недостаточность чисто приаппаратной оценки уровня языка [14].

расположенных в памяти (вектора, строки, стеки, очереди, файлы). Можно рассчитывать, что так определена базовая часть учебного концентра (элементарного уровня) любого ЯП. Другие вспомогательные семантики языков ассемблера, макропроцессора, вычислений над стеком и управления процессами, машинно-зависимых и машинно-независимых машинно-ориентированных ЯП могут рассматриваться как расширения такой базовой части. Здесь не рассмотрены СУБД, языки работы с сайтами и ряд других средств создания распределённых систем.

3.1. Императивное программирование на ассемблере

Обработка данных с помощью программ, представленных на языке ассемблера, сводится к императивной машинно-ориентированной модели управления процессом выполнения действий, порожденных программой. В центре внимания – конфигурация оборудования, состояние памяти, система команд, передачи управления, очередность событий, исключения и неожиданности, время реакции устройств и успешность процессов обработки информации, нацеленных на свободный доступ к любым возможностям оборудования. Кодирование алгоритма осуществляется на фоне применения дополнительных средств, таких как блок-схемы и документирование, отчасти компенсирующих отсутствие в языке ассемблера понятий уровня программистской фразеологии, а в естественных языках – понятий, возникающих при такой «сверхточной» детализации программ на языке ассемблера. Результативность представления программ обеспечивает макротехника или автоматный подход к реализации алгоритмов – методика автоматного программирования, поддерживающая выделение шагов модели программы независимо от базовых средств ЯНУ. Автоматное программирование возникло до появления ЯВУ.

Для языков ассемблера в дополнение к общей семантике элементарного уровня ЯНУ характерно наличие команд над вещественными числами и обработки кодов. Работа с идентификаторами реализуется аппаратными возможностями адресации памяти, обычно обеспечивающей доступ практически к любому хранимому в памяти данному или команде. Управление вычислениями может учитывать результаты промежуточных вычислений (ветвления и переключатели), выполнять итерирование участков повторяемости (циклы) и вызовы подпрограмм, а также обрабатывать внутренние и внешние прерывания. В качестве поддержки структур данных можно рассматривать пересылки блоков заданной длины,

а также средства работы с текстом программы. В качестве опорных языков рассмотрены ассемблеры MIX, MSX, P-код, RISC-машина, byte-код (muLisp и Java), MASM и БЭМШ.

Традиционно ассемблер реализуют как упрощенный компилятор. Учитывая повышенную нагрузку низкоуровневого программирования на отладку программ, иногда включают в систему программирования дизассемблер и интерпретатор ассемблера, обеспечивающие кроме удобства отладки широкий спектр преобразования программ на ассемблере, их оптимизации и адаптации к развитию аппаратуры. Интерпретирующий автомат для ассемблера устроен проще, чем автомат для абстрактной машины SECD, благодаря встроенной реализации команд – языка конкретной машины и отсутствию локализации имен и их областей действия. Процесс перевода программы с языка ассемблера в язык машинных команд называют ассемблированием.

Язык ассемблера оперирует такими данными, как адреса и значения. Нередко для наглядности в записи операндов команд вводится внешнее различие @адресов и #значений с помощью префиксных символов. Возможны специальные формы записи для блоков данных и литералов.

При ассемблировании текст программы отображается в частично адресуемый код программы, исполнение которого обычно сводится к поячеечному изменению состояния памяти:

$$\text{ассемблер} = (\text{Текст} \rightarrow \{\text{Код} \mid \text{Адрес}\}): \text{Пам}[\text{ячейка}] \rightarrow \text{Пам}$$

Число слов, отводимое ассемблером под одну символическую команду, зависит не только от собственно кода команды, но и от метода адресации операндов, а возможно, и от других аспектов кодирования программ и данных, обсуждение которых здесь не предполагается. Достаточно констатировать, что программа при ассемблировании распадается на конечные последовательности команд K1 ... Kn, которым сопоставляются конечные интервалы машинных слов W1 ... Wm(a) в зависимости от **a** – системы аспектов кодирования:

$$[K1 \ldots Kn] \rightarrow [W1 \ldots Wm(a)]$$

Фактически, операндная часть команды – это адреса, специальные регистры, сумматоры, значения в общей памяти, шкала прерываний и состояний или что-то другое, специфичное для конкретной архитектуры.

Парадигма низкоуровневого кодирования на ассемблере нацелена на учет любых особенностей компьютерных архитектур. Архитектура

компьютера часто определяется как множество ресурсов, доступных пользователю. Это система команд, общие регистры, слово состояния процессора и адресное пространство. Процесс ассемблирования заключается в следующем:

– резервирование памяти для последовательности команд, образующих ассемблируемую программу;
– сопоставление используемых в программе идентификаторов с адресами в памяти;
– отображение ассемблерных команд и идентификаторов в их машинные эквиваленты.

Для реализации такого процесса требуется счетчик адресов и таблица идентификаторов. Программист не знает абсолютных адресов ячеек памяти, занятых для хранения констант, команд и промежуточных результатов вычислений, но обычно предполагается, что последовательно написанные команды будут расположены в последовательных ячейках памяти.

Язык ассемблера обычно различает следующие категории команд:

– вычисления с результатом в сумматоре, для которых следующая команда расположена по соседству;
– изменение части состояния общей памяти, при котором следующая команда расположена по соседству;
– обработка текста программы без генерации нового кода;
– управление со своими правилами выбора следующей команды.

При записи команд на ассемблере принято структурировать строки на поля, предназначенные для последовательного размещения метки, кода команды, операндов и комментария. Наибольшее разнообразие возможных решений связано с формами адресации операндов на уровне машинного языка. В зависимости от команды используются разные методы адресации операндов.

Обычно система команд ассемблера, кроме команд арифметических вычислений и передач управления, содержит команды манипулирования данными, их ввода-вывода через буфер обмена, возможно, доступный через механизм прерываний. При этом используется код состояния процесса, отражающий свойства текущей выполняемой команды, такие как выработка нулевого или ненулевого кода, отрицательного или положительного числа, переноса разряда за границы машинного слова и др.

Имеются команды условного перехода, возвратный вызов процедуры с использованием регистра возврата и передача управления со счетчиком числа циклов. Встречаются и другие команды, разнообразие которых не

влияет на особенности ассемблирования и применения ассемблеров в системах программирования. Именно язык ассемблера традиционно выступает в системах программирования в роли конкретной машины (КМ) при компиляции программ.

Программирование на ассемблере подразумевает знание специфики системы команд процессора, методов обслуживания устройств и обработки прерываний. Система команд может быть расширена микропрограммами и системными вызовами в зависимости от комплектации оборудования и операционной системы. Это влияет на решения по адресации памяти и коммутации комплекта доступных устройств. Но есть и достаточно общие соглашения о представлении и реализации средств обработки информации на уровне машинного кода.

Обычно ассемблер обеспечивает средства управления распределением памяти и управления компиляцией кода независимо от системы команд, что позволяет программисту принимать достаточно точные решения на уровне машинного языка. Существуют средства отладочного исполнения и распечатки листингов. Управление ассемблированием обычно обеспечивает средства авторизации, взаимодействия с отладчиком, текстовым редактором, операционной системой и средствами приаппаратного уровня, доступными через операционную систему и аппаратные средства ввода-вывода (BIOS).

По тексту программы при ассемблировании формируется код программы, выполняющий пошаговое преобразование памяти.

Программирование на ассемблере требует знания особенностей применяемых команд, их операндов и результатов. Большинство команд изменяют один регистр, если не считать счётчик команд. Некоторые команды могут изменять более одного регистра. Например, умножение вещественных чисел может формировать регистр младших разрядов для вычислений с повышенной точностью, деление целых чисел может формировать регистр с остатком от деления, пересылка меняет содержимое целого ряда машинных слов.

Спецификация команд абстрактной машины ассемблера может быть задана над тройкой <S, C, M>, в которой S и C – простые регистры, а M – вектор, представляющий общую память.

Приведём пример системы команд.

Таблица 13

Пример системы команд ассемблера

Обозначение команды	*Описание команды* [12]	*Примечание*
LDC	Засылка адреса в сумматор.	Подготовка операндов
LDM	Засылка адресуемого слова в сумматор.	
LDS	Косвенная засылка слова в сумматор.	
RV	Пересылка из сумматора в память.[13]	Сохранение результата
RVS	Пересылка слова по адресу из сумматора в память.	
GO	Безусловная передача управления.	
GS0	Условная передача управления по нулевому значению сумматора.	Ветвление
GS1	Условная передача управления по ненулевому значению сумматора.	
GSUB	Передача управления с запоминанием адреса возврата в сумматоре.	Подпрограмма
DEC	Вычитание 1 из содержимого сумматора.	Арифметические операции
INC	Прибавить 1 к содержимому сумматора.	
ADD	Прибавление адреса к содержимому сумматора.	
SUM	Суммирование адресуемого слова и содержимого сумматора.	

Таблица 14

Пример спецификации команд ассемблера

RA	RA’	*Примечание*
s (LDC Adr . c) m	→ (s:=Adr) c m	Операнд из команды
s (LDM Adr . c) m	→ (s:=[Adr]) c m	Операнд из памяти
(s:Adr) (LDS . c) m	→ (s:=[Adr]) c m	Операнд по адресу в сумматоре
s (RV Adr . c) m	→ s c (m|m.Adr:=[s])	Результат из сумматора
(s:Adrs) (RVS Adr . c) m	→ s c (m|m.Adr:=[Adrs])	Результат по адресу
s (GO Adr . c) m	→ s Adr m	

[12] Следующая команда расположена по соседству или задана командой передачи управления

[13] Пересылки – это изменение части состояния общей памяти.

(s:0) (GS0 Adr . c) m	→ s Adr m	
(s:1) (GS1 Adr . c) m	→ s Adr m	
s (GSUB Adr . c) m	→ @c Adr m	Укрупнение действий
s (DEC . c) m	→ (s-1) c m	
s (INC . c) m	→ (s+1) c m	
s (ADD Adr . c) m	→ (s+Adr) c m	
s (SUM Adr . c) m	→ (s+[Adr]) c m	

Применение ассемблера для разработки многократно используемых модулей налагает определённые ограничения на структуру кода программы и его свойства:

– перемещаемость. Свойство удобно, когда код программы устроен так, что его можно расположить по любому абсолютному адресу;
– листание страниц памяти. Соотношение между адресуемой и реально доступной памятью может требовать пересмотра в процессе выполнения программы;
– зависимость от данных. Программа должна учитывать готовность данных и вероятность их обновления, особенно в случае обмена информацией через порты или буферы устройств;
– динамика размещения. Программа может размещаться в памяти пошаговым образом, методом раскрутки, с использованием динамической оптимизации, учитывающей статистику реального использования ее составляющих.

Примеры программ решения типовых задач средствами ассемблера:

Фрагмент программы	*Примечание*
LD HL, ADR1 откуда LD DE, ADR2 куда LD BC, 2048 сколько LDIR	Установить регистр HL Установить регистр DE Установить регистр BC Переместить данные в соответствии с содержимым регистров HL, DE, BC

Пример 1. Пример из ассемблера MSX:
пересылка блока (ТД не имеет значения)

Ассемблер отличается от компилятора меньшей сложностью исходного языка, перевод с которого в машинный язык можно выполнить «один-в-один». Ассемблер часто сопровождается возможностью дизассемблирования, что отличает его от большинства других языков программирования. Изучить язык ассемблера проще, чем любой язык

высокого уровня. Знание ассемблера помогает понимать код программы, подготовленной на других языках.

Основной механизм укрупнения действий – передача управления подпрограмме. При необходимости уровень языка может быть повышен с помощью макросов.

Императивный стиль программирования наследуется большинством ЯВУ, поддерживающих процедурно-императивное и объектно-ориентированное программирование.

Таблица 15

Парадигматическая характеристика ассемблера

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Ассемблер – средство эффективного программирования при решении задач доступа к полному спектру возможностей оборудования
Регистры абстрактной машины	S C M S – сумматор; C – поток команд программы с указателем на текущую команду; M – вектор памяти произвольного доступа. Результат рассредоточен по ячейкам памяти
Категории команд абстрактной машины	- вычисления с результатом в сумматоре, для которых следующая команда расположена по соседству; - изменение части состояния общей памяти, при котором следующая команда расположена по соседству; - обработка текста программы без генерации нового кода; - управление со своими правилами выбора следующей команды; - команды «знают», где их операнды.
Реализационная прагматика	- перемещаемость. Код можно расположить по любому абсолютному адресу; - листание страниц памяти; - зависимость от данных; - динамика размещения; - объектный код; - билистинг;

	- прокрутка и отладчик; - реассемблирование.
Парадигматическая специфика	Классический язык императивного машинно-ориентированного программирования.

3.2. Стековая машина Forth

Для машинно-ориентированных языков, таких как Forth [1], система вычислений распадается на подсистемы по величине обрабатываемого слова (16 и 32, возможно, 64). Основа работы с памятью – стек. Средства управления вычислениями обогащены средствами блокировки и кодирования программ, что позволяет повышать эффективность информационной обработки. Используется механизм замкнутых процедур с неявными – стековыми – параметрами. Стек реализован как указатель на текущий элемент в предположении, что перед ним по порядку расположены предшествующие элементы.

Программа – отдельный поток, использующий расширяемый словарь. Принята постфиксная запись, удобная для стековой обработки данных. Стек-ориентированная дисциплина обработки освобождает от необходимости в понятии «переменная», хотя оно при необходимости моделируется.

Программирование на Forth-е сопровождается систематической сверткой понятий, синтаксис применения которых созвучен польской записи. Можно сказать, что хорошая программа на Forth-е – это специализированная виртуальная машина, приспособленная к дальнейшему расширению по мере развития постановки задачи.

Интерпретатор языка Forth сортирует слова по принадлежности словарю:

- слова, не найденные в словаре, записываются в стек для предстоящей обработки;
- словарным словам, встроенным в интерпретатор, соответствует правило преобразования стека;
- возможно определение новых слов, запоминаемое в словаре (от «:» до «;»);
- за корректность воздействий на стек отвечает программа;
- результатом считается состояние стека при завершении программы.

Программа на языке Forth строится как последовательность слов, некоторые их которых включены в расширяемый словарь языка:

Forth: текст/словарь → стек: стек → стек'
: слово → словарь

Данные – это тоже слова. Логическое значение «истина» – 0.
Новые слова можно вводить в форме:

: имя опр;

Абстрактный синтаксис языка отличается от АС ассемблера допущением произвольного числа неявных аргументов определяемых команд:

S E C → S' E' C'

S – стек результатов, E – словарь, представленный как вектор строк,
C – поток слов, образующих программу.

Исполнение программы организовано как диалог над стеком. Каждая команда знает, что взять из стека, во что преобразовать для получения результата программы и какие результаты разместить в стеке.

Таблица 16

Команды стековой машины языка Forth

SCS	*Описание команды*	*Примечание*
LDC	Засылка из программы в стек.	Операнды.
DROP	Сбросить из стека верхний элемент.	Манипуляции с фиксированным числом элементов стека.
DUP	Скопировать верхний элемент.	
NIP	Сбросить предпоследний элемент стека.	
OVER	Предпоследний элемент переставить наверх.	
PICK	Выборка из стека элемента с указанным номером.	
SWAP	Обмен местами двух верхних элементов.	
TUCK	Копирование верхнего элемента под второй с сохранением.	
DEPTH	Глубина стека.	

PICK	Выборка из стека элемента с указанным номером.	Манипуляции с заданным числом элементов стека.
ROLL	Перестановка на заданную глубину элемента из стека.	
?DUP	Скопировать ненулевой верхний элемент.	
LDF	Загрузить в словарь определение.	Укрупнение действий.
AP	Применить словарное определение.	

Укрупнение действий – это новые определения с неявными параметрами в стеке.

Таблица 17

Спецификация команд стековой машины языка Forth

RS	RS	*Примечание*
s e (LDC X . c)	(X . s) e c	Подготовка стека к предстоящим операциям. N – длина стека
(X . s) e (DROP . c)	→ s e c	
(X . s) e (DUP . c)	→ (X X . s) e c	
(X2 X1. s) e (NIP . c)	→ (X1 . s) e c	
(X2 X1. s) e (OVER . c)	→ (X2 X1 X2 . s) e c	
(XN ... X0 N . s) e (PICK . c)	→ (XN ... X0 XN . s) e c	
(XN ... X0 N. s) e (ROLL . c)	→ (XN-1 ... X0 XN . s) e c	
(X3 X2 X1. s) e (ROT . c)	→ (X2 X1 X3 . s) e c	
(X2 X1. s) e (SWAP . c)	→ (X1 X2 . s) e c	
(X2 X1. s) e (TUCK . c)	→ (X1 X2 X1 . s) e c	
s (DEPTH . c)	→ (N . s) e c	
(0 . s) e (?DUP . c)	→ (0 . s) e c	
(X . s) e (?DUP. c)	→ (X X . s) e c	
(NF . s) e (LDF F “;” . c)	→ s (e|e[NF]=F) c	Пополнение словаря.
(NF . s) e (AP . c)	→ s e (e[NF] . c)	Применение словарного определения.

Система программирования для языка Forth содержит пару «интерпретатор – компилятор», причем техника компиляции весьма эффективна. Система использует единый порядок представления данных и команд в программе это последовательности слов. Данные располагают перед операциями по их обработке. Операция – это известное системе

слово. Данные просто загружаются на стек, из которого операция берет их в соответствии с числом ее параметров.

Интерпретирующий автомат для языка Forth по сложности сравним с автоматом для ассемблера. Основные различия таковы:

- словарь Forth-а хранит строки произвольной длины, а таблица меток ассемблера хранит адреса фиксированного размера;
- вместо запоминания адреса возврата при организации подпрограмм/функций определения размещаются расширяемом словаре;
- неявные параметры функций заранее размещаются в стеке, их число известно;
- результаты вычислений сконцентрированы в стеке.

Таким образом, обеспечены базовые функциональные возможности, характерные для систем программирования на языках высокого уровня.

Фрагмент программы	*Примечание*
(x y z → x y*z)	Перемножение двух верхних элементов стека
OVER (x y*z → x y*z x)	Копирование второго элемента на верх стека
- (x y*z x → x y*z - x)	Разность двух верхних элементов
SWAP (x y*z - x → y*z – x x)	Перестановка двух верхних элементов
DUP (y*z – x x → y*z – x x x)	Дубль верхнего элемента
* (y*z – x x x → y*z – x x*x)	Перемножение двух верхних элементов
+ (y*z – x x*x → y*z – x + x*x)	Сумма

Пример 2. Программа на языке Forth для подсчета по формуле
(x y z → x**2 + y*z – x)

Фрагмент программы	*Примечание*
: SQ (вводится слово SQ в словарь)	Объявление имени укрупнённого действия
DUP (A B → A B B)	Дубль верхнего элемента
* SWAP (A B B → A B**2 → B**2 A)	Перемножение и перестановка
DUP * (B**2 A → B**2 A A → B**2 A**2)	= Дубль и квадрат
+ (B**2 A**2 → B**2 + A**2)	Сумма – результат в стеке
; (конец определения нового слова)	

Пример 3. Введение нового слова SQ для подсчета суммы квадратов
(A B → A**2 + B**2)

Для удобства программирования определений функций привлекается методика моделирования переменных и констант, а также средств управления процессами с помощью ветвления: условное выражение и переключатель, моделирование циклов.

Язык Forth – пример организации вычислений над стеком. Его можно рассматривать как язык-ядро с возможностью практически неограниченного проблемно-ориентированного расширения машинно-независимых эффективных средств программирования. Язык допускает порождение эффективного кода «хорошо» написанных программ.

Программирование на Forth-е требует вдумчивости и аккуратности. Достижимость лаконичных форм дается ценой нестандартных индивидуальных решений, мало приспособленных к передаче программ в чужие руки. Лозунги «Программируйте все сами!» и «Не бойтесь все переписывать заново!» правильно отражают подход к программированию на Forth-е. Успех достигается максимализмом в тщательной отладке и способностью видеть задачу программирования в развитии.

Автор языка Forth Чарльз Маури в 1968 году отметил: «Forth не уравнитель, а усилитель!».

К середине 70-х Forth стал третьим по популярности после Бейсика и Паскаля, завоевав свои позиции при освоении микропроцессорных средств. По технике программирования Forth похож на макроассемблер, только вместо системы команд над машинными словами в нем используется система операций на стеком.

Таблица 18

Парадигматическая характеристика языка Forth

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Язык Forth – пример организации вычислений над стеком, применяемый как язык-ядро с возможностью практически неограниченного проблемно-ориентированного расширения машинно-независимых эффективных средств программирования. Допускает порождение эффективного кода «хорошо» написанных программ. Средство разработки специализированных систем, создаваемых по методике раскрутки, начиная с тщательно минимизированного ядра с рядом

	последовательных шагов расширения в рамках единой оболочки. Мало приспособлен к передаче программ в чужие руки. Компактный язык-ядро для пошагового решения задач с расширяемой постановкой.
Регистры абстрактной машины	S E C S – стек операндов и результатов. E – словарь определений. C – программа – поток операций и значений
Категории команд абстрактной машины	Преобразование содержимого стека. Вычисление над стеком. Применение программируемых определений
Реализационная прагматика .	Стек – вектор заданного системой размера. Элементы стека – слова фиксированной разрядности. Система программирования использует пару интерпретатор-компилятор.
Парадигматическая специфика	Механизм реализации выражений в большинстве ЯВУ. Forth обладает концептуальным родством с языком Lisp.

3.3. Продукционная макротехника

Макротехника дает весьма мощные, но не вполне безопасные средства повышения выразительности ЯП. Для макропроцессоров семантика ЯНУ обычно сопровождает средства работы со строками в стиле открытых процедур. Программа представляет собой поток макроопределений и макровызовов. Имена макросов могут рассматриваться равноправно с базовыми средствами. При определении и реализации макросов используется понятие позиции и шаблона для подстановки параметров. Встречается стиль нумерации позиций, который позволяет обойтись без их именования в виде переменных. В результате подстановки макросов формируется текст, равноправный с исходным, возможно, содержащий вторичные макросы. Макропроцессор часто используется в паре с ассемблером (макроассемблер) и другими ЯП. В качестве опорных рассмотрены макропроцессоры GPM, TRAC, а также два макропроцессора из системы подготовки программ на языке SETL.

Для автономных макропроцессоров характерны специальные механизмы регулярного конструирования различимых текстов:

– управляющие символы;

– счетчики;
– генераторы уникальных значений;
– блоки активности (разметка);
– встроенные функции;
– управление вводом-выводом.

Большинство макропроцессоров допускают переменные макропериода – глобальные и локальные макропеременные.

Для макропроцессора программа – это содержащий макровызовы текст, при обработке которого программа преобразуется в новый текст программы, полученный в результате макропреобразований:

$$Макро = (Прог = Текст): Прог \rightarrow Прог'$$

Интерпретатор макропроцессора при последовательном сканировании текста выделяет в нем следующие категории строк:

– макроопределение, которое следует поместить в таблицу макросов ($c \rightarrow e$);
– макровызов без параметров, определение которого следует скопировать из таблицы в результат ($e \rightarrow c$);
– макровызов с параметрами, значения которых следует установить, а затем подставить в буферную копию определения, и преобразованное определение разместить в результат ($e \rightarrow p \rightarrow c$);
– простая строка, сохраняемая в результате без преобразований;
– конец текста.

Абстрактный синтаксис макротекстов показывает появление функций с произвольным числом параметров, открыто подставляемых в шаблон определения.

(DEF Name Arg1 Arg2 … ArgN Patern) – разделители и ограничители сняты при переходе к АС.

(Name Txt1 Txt2 … TxtN) – однократная открытая подстановка текстов в место соответствующих аргументов.

Спецификация команд абстрактной машины макропроцессора может быть задана парой <E, C>, в которой C – строка, а E – вектор определений.

Таблица 19

Пример системы команд макропроцессора

SCM	*Описание команды*	*Примечание*
LDF	Ввод шаблона макроса в таблицу под заданным именем.	Укрупнение действия.
LDN	Размещение параметра макроса в таблице под заданным номером.	Локальные параметры.
SP	Сцепление строки с текстом = обход строки.	Константа.
ARG	Копирование параметра в текст	Подстановка параметра.
AP	Применение макроса.	Применение макроса.

Таблица 20

Спецификация макрокоманд

RM	RM’	*Примечание*
e (LDF Mac Ptn . c)	→ (e|e[Mac]:=Ptn) c	Размещение макроса в таблица определений.
e (LDN Num Arg . c)	→ (e|e[Num]:=Arg) c	Размещение параметров в таблице определений.
e (SP X . c)	→ e c	Пропуск константы.
e (ARG Num . c)	→ e (e[Num] | c)	Подстановка параметра.
e (AP Mac . c)	→ e (e[Mac] | c)	Подстановка макроса.

Макропреобразования могут использовать локальные или глобальные переменные, вложенность областей действия определений, рекурсию. Макропроцессор может быть встроен в компилятор, может быть автономным инструментом системы программирования, таким как текстовый редактор, оптимизатор или отладчик, или может существовать самостоятельно как универсальный инструмент общего назначения.

Макровызовы могут выполняться за одни просмотр или до исчерпания при итеративном анализе текста. Встречаются средства управления глубиной макроподстановки. Популярно синтаксическое подобие макросов выражениям базового языка, хотя это может вызывать путаницу в понимании реальных механизмов при порождении кода программы.

Техника строковой обработки обычно поддерживается операциями вычисления длины строки, выделения подстроки и конкатенации строк.

В системах программирования макротехника применяется на двух уровнях: препроцессоры обычно формируют входной текст для компилятора, а макроассемблеры выполняют сборку кода на уровне генерации ассемблерной программы или её объектного кода.

Макропроцессор получает логическое завершение, поддерживая динамически формируемые макроопределения, возможно используя макровызовы в аргументах.

Встречается интересное применение вложенности макровызовов и макроопределений, включая рекурсию вида ФАКТ (сч) = если <сч = 0> то [1] иначе [(| сч | +] | ФАКТ [сч - 1] | [ )] .

Макроопределение	*Примечание*
ФАКТ (сч) = | <сч = 0> → [0] | | | [(| сч | *] | | | ФАКТ [сч - 1] | | | [ )] ФАКТ (6) = (6 * (5 * (4 * (3 * (2 * (1 * 1))))))	Строка для значения 0 Строка из «(», значения «сч» и «*» Строка из макроса от «сч», уменьшенного на 1 Строка из «)» Результат макроса на значении 6

Пример 4. Рекурсивное макроопределение факториала

Макроопределение	*Примечание*
§A, §Def, A , <D>; §Def, B, <C>;;	Вызов «A», где «A» указывает на «D», а «B» на «C»

Пример 5. Пример макропрограммы на GPM. Моделирование « if A=B then C else D » обеспечено побочным эффектом на глобальной таблице имен

Техника выполнения макропреобразований достаточно разнообразна. Так, например, язык GPM всю работу с макросами сводит к макровызову вида:

§ mak, a1, a2, ... aN; – вызов макроса.

Позиции макровызова занумерованы по числу предшествующих запятых, что делает ненужным описание переменных и дает возможность самоприменения определений:

~0 ~1 ~2 ... ~N – описание не нужно.

Кроме того используются скобки, блокирующие подстановки при необходимости:

< S > – блокировка подстановок в S.

Достаточно всего одной встроенной функции DEF, выполняющей введение макроопределений.

Макроопределение	*Примечание*
§Def, mak, опр;	Команда создания нового макроса

Пример 6. Введение новых макроопределений GPM

Макроопределение	*Примечание*
§Def, size, 6; §size; => 6 x (§size, §size) => x(6,6) size§size => size6	Определение макроса Варианты вызовов макроса

Пример 7. Использование макроопределений GPM

Макроопределение	*Примечание*
§Def, opp, UN~1; §opp, R; §Def, opp, <UN~1>; §opp, R; => UNR	Параметр «~1» вне подстановки => ОШ – нет определения Параметр может быть подставлен => UNR

Пример 8. Использование блокировок в макроопределениях GPM

Совершенно иначе выглядит макротехника в не менее лаконичном языке макропроцессора TRAC. Все сводится к макровызовам функций, встроенных и определяемых:

(F, s1,s2,...,sN)

Встроенные функции:

ds – определение строки,

cl – вызов определение,
ss – выделить сегменты,
rs – чтение строки.

Макроопределение	*Примечание*
#(ds,ПРИМЕР, собака сидит на ковре) #(ss,ПРИМЕР, собака, ковре) #(cl,ПРИМЕР, кошка, кресле) = кошка сидит на кресле	Исходная строка. Выделены замещаемые сегменты. Задана подстановка = результат.

Пример 9. Работа с шаблонами на языке Trac

Два интересных механизма макротехники были реализованы в проекте языка Setl при попытке его эффективной реализации посредством языка Little.

Для поддержки переноса программы на разные архитектуры предлагалась специальная разметка текста с помощью флагов, в зависимости от значения которых блоки строк включались во входной текст для компилятора. Значения флагов можно было инициировать, наращивать или редуцировать и обнулять.

+ flag – включить строку;

.flag – завершение блока, сопровождается увеличением или уменьшением счетчика, одноименного с флагом;

- flag – пропустить строку.

Для автоматизации формирования фрагментов текста, обладающих зависимостью от численных характеристик или кратности вхождения в программу, использовался специальный механизм специальных макропеременных.

Макроопределение	*Примечание*
zxN => N + I zyN = N' => N' (zyN := N') zaN => A(N+i)	В строке размещается значение счетчика. Задание значения спецпеременной. В строке размещается имя “A”, сцепленное со значением счетчика.

Пример 10. Представление зависимости от процесса формирования текста

Общеизвестно, что макрос легче применять, чем определять. Внешняя простота введения макросов сопряжена с вероятностью порождения трудно обнаруживаемых ошибок периода исполнения программы, индуцированных случайным сходством с подпрограммами на основном языке программирования при существенном различии:

- макрос меняет текст программы,
- подпрограмма меняет данные программы и логику процесса исполнения программы.

Макротехника приносит результаты не только на текстах, но и на геометрических фигурах, графах и кодах. Например, макросами можно описать пентамино, оптимизацию и кодогенерацию программ.

Иногда встречаются более специализированные средства, использующие счётчиковые переменные, конструкторы уникальных имен, моделирующие иерархию модулей или параметризирующие зависимость вариантов программы от целевых архитектур.

Определения	*Примечание*
#DEFINE THEN #DEFINE BEGIN { #DEFINE END ;}	**«Синтаксический сахар»:** Можно определить ключевые слова для рамочных конструкций;

IF (I > 0) THEN
BEGIN
A = 1;
B = 2
END
	затем написать текст, похожий на Pascal-программу.

#DEFINE MAX(A, B) ((A) > (B) ? (A) : (B))	**Макрос:** Такое определение обеспечивает функцию «максимум», которая реализуется как последовательный код, а не вызов функции.
X = MAX(P+Q, R+S);	**Макровызов:**

	При правильном объявлении аргументов такой макрос будет работать с любыми типами данных.
X = ((P+Q) > (R+S) ? (P+Q) : (R+S));	**Результат макроподстановки:** Нет необходимости в различных видах MAX для данных разных типов, как это было бы с функциями

Пример 11. Пример макросов языка С

Определения	*Примечание*
(defun MF (NF AF BF) (list 'DEFUN NF AF BF))	Макрос
(eval (MF MNF (A B) (cons B A)))	Макровызов
(MNF 1 2)	Применение результата

Пример 12. Пример макротехники на языке Lisp

Основные отличия АМ макрогенератора от АМ ассемблера и стековой машины связаны с переходом от обработки простых значений, размещаемых в словах фиксированного размера, к открытой обработке строк произвольной длины и с использованием программируемых определений, расширяющих исходную систему команд и поддерживающих локализацию переменных.

Таблица 21

Парадигматическая характеристика макропроцессора

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Главное предназначение макросов в системах программирования – достижение гибкости и переносимости текстов программ, применяемых в разных условиях. В системах программирования макротехника применяется на двух уровнях: препроцессоры обычно формируют входной текст для компилятора; макроассемблеры выполняют сборку кода на уровне генерации ассемблерной программы или её объектного

	кода.
Регистры абстрактной машины	E C E – вектор определений и параметров. C – программа, модифицируемая согласно макровызовам. Результат работы макрогенератора – новая форма текста программы.
Категории команд абстрактной машины	Засылка определений. Сцепление фрагментов в строку. Копирование параметров. Применение определения.
Реализационная прагматика	Открытая подстановка без контроля границ стыковки фрагментов.
Парадигматическая специфика	Макротехника близка продукционному стилю программирования, языкам разметки и системам переписывания текстов, в настоящее время активно развивающимся как языки гипертекстов для разработки сайтов и информационных сервисов. Макротехника характерна для ЯВУ, поддерживающих открытую подстановку параметров, вызовы по необходимости при организации отложенных вычислений и специальных функций, использующих пост-обработку параметров.

3.4. Языки управления процессами

Рассмотрим базовые средства для решения проблем обработки информации на уровне функционирования операционных систем (ОС), исполнения отдельных задач и разработки информационных систем, активно использующих понятие «файл». Нередко представление процессов и файлов может быть унифицировано, что видно по сопоставлению команд по обработке файлов и манипулированию процессами.

Таблица 22

Параллель между командами над файлами и процессами

Действие	*Файлы*	*Процессы*
Вывести список	Ls	Ps
Сменить статус	Chmod	Bg Fg
Удалить	Rm	Kill
Сцепить последовательно	Cat	| конвейер
Показать контекст/задания	Env	Jobs
Создать файл/процесс	Cp – в новый файл	Fork
Сменить директорию/процесс	Cd	Exec
Проверить/Ждать	Test	Wait
Переход к результату	Echo	Eval
Формат файла/процесса	Таблица строк/записей	Список команд с параметрами

Описание процесса начинается с определения класса событий, представляющих интерес для участвующих в нем объектов. Множество имен событий, используемых при описании процесса или объекта, обычно предопределено.

Первая абстракция при моделировании процессов – исключение времени, т. е. отказ от ответов на вопрос, происходят ли события строго одно за другим. Это обеспечивается следующими договоренностями:

– элементарные действия исполняются мгновенно;
– протяженное действие: всегда пара событий – начало и конец;
– нет точной привязки действий к моменту времени;
– определены отношения «раньше – позже», «одновременно», «независимо»;
– совместность событий понимается как отношение «синхронизация»;
– одно событие из независимых возникает в любом порядке, без причинно-следственной связи.

На уровне операционной системы (ОС) информационная обработка выглядит как семейство взаимодействующих процессов, выполняемых по отдельным программам – заданиям или сценариям, размещенным в файлах. Языки для ОС работают с очередями, которые могут быть представлены как строки или файлы. В памяти хранится контекст задания и его сценарий.

Контекст содержит перечень доступных файлов. При управлении процессами выполнения заданий используются условия готовности и вырабатываются сигналы, символизирующие успех выполнения действий. Сигналы также хранятся в контексте. Действия могут быть организованы в конвейеры или последовательности и обусловлены успехом предшествующих действий. Очередь может быть пополнена.

Функционирование ОС обеспечивает следующие явления и критерии:

– порождение новых файлов и процессов по ходу дела;
– время жизни файлов и процессов произвольно – нет гарантий;
– неограниченная динамика событий;
– содержание может быть незавершенным;
– изменение содержания и состава;
– очередь процессов с условиями готовности.

Построение модели языка управления заданиями требует дополнительных операций по работе с очередями, что может быть устроено как «ленивый» список, в конец которого можно встраивать новые элементы функцией Conc. В качестве опорного языка рассмотрен Bash, абстрактная машина которого может быть определена как <E, C, D>, где:

E – контекст_процесса – вектор записей Имя: Данные + stdin stdout;
C – текущий_процесс – строка из команд;
D – очередь_отложенных_процессов – вектор записей Имя: Данные.

Команды интерпретатора ОС выполняют обмен данными между процессами и контекстом, обработку очереди, файлов и контекста, проверку ряда условий над данными, контекстом и очередью, выработку сигналов, включая установку сигнала о завершении процесса.

Таблица 23

Типичный набор команд языка управления процессами

SCQ	*Описание команды*	*Примечание*
ECHO	Вывод аргументов	Визуализация
PWD	Выводит название текущего рабочего каталога	
LS	Список файлов в текущей директории	
CAT | MORE	Просмотр содержимого текстового файла	
CD	Сменить директорию	Манипуляции с файлами
CP	Копировать файлы	

MV	Переместить или переименовать файл	
RM	Удалить файлы	
LN	Создать символическую ссылку	
EVAL	Конструирование команды на лету и ее выполнение	Вычисления
EXEC	Вызов другого процесса	
LET	Вычисление выражений	
READ	Ввод значения переменной	Установка значений
SET	Изменяет значения внутренних переменных скрипта	
TEST	Проверка условия	
TRUE	Возвращает код успешного завершения = ноль	
FALSE	Возвращает код завершения, свидетельствующий о неудаче	
Контроль процессов		
PS	(print status) список текущих процессов с их IDs (PID)	Визуализация
FG	Активизировать фоновый или приостановленный процесс	Управление активностью процессов.
BG	Сделать процесс фоновым. Обратная функция от fg.	
WAIT	Ждет выхода из дочернего процесса	
KILL	«Убить» процесс. PID «убиваемого» процесса даёт PS	

Обозначения:

stdin – стандартный ввод. То, что набирает пользователь в консоли.
stdout – стандартный вывод программы.
stderr – стандартный вывод ошибок.
(Expr) – результат вычисления выражения или успех выполнения процесса.
$ – переменная для кода успеха/результата процесса.
$* – все аргументы переданные скрипту(выводятся в строку).
$! – PID последнего запущенного в фоне процесса.
$$ – PID самого скрипта.
NN(d) – список номеров и имён элементов очереди
[<Num, Name, Text>, …] – очереди процессов.
NULL – пустой файл.
H(d) – голова очереди, точнее – процесс с наивысшим приоритетом.
T(d) – хвост очереди, остаток после удаления головы. d = H(d) • T(d).
PN – имя текущего процесса.

Таблица 24

Спецификация команд управления процессами

RQ	RQ’	*Примечание*
e (ECHO String . c) d	→ (e[stdout] | String) e c d	Вывод на стандартное устройство
(e[PWD]=DName) (PWD . c) d	→ (e[stdout] | DName) c d	
(e[PWD]=DName) (LS . c) d	→ (e[stdout] | [DName]) c d	
e (CAT Fname . c) d	→ (e[stdout] | [Fname]) c d	
e (CD New. c) d	→ (e[PWD] := New) c d	Установка переменной
(e[F1]=Datum) (cp F1 F2. c) d	→ (e[F1]=Datum; e[F2]=Datum) c d	Ссылка на копию файла
(e[Fname]=Datum) (RM Fname . c) d	→ (e[Fname]=NULL) c d	Ссылка на пустой файл
e (EVAL T1 ... TK . c) d	→ e (T1 | ... | TK . c) d = e ($* . c) d	Выполнение скрипта
e (EXEC Fname . c) d	→ e ([Fname] c) d	Выполнение файла
e (TEST Expr . c) d	→ (e[$] := (Expr)) c d	Проверка успешности
e (TRUE . c) d	→ (e[$] := 0) c d	Установка признака успешности
e (FALSE . c) d	→ (e[$] := 1) c d	
e[stdin=Text] (READ X . c) d	→ (e[X] := Text) c d	Прием текста
e (PS . c) d	→ (e[stdout] | NN(d)) c d	Номера процессов
e (BG . c) d	→ e H(d) (T(d) | <$$, e[PN], c >)	Смена статуса процесса
e (FG Num . c) (d[Num]=Text)	→ e (Text | c) (d[Num]:=NULL)	
e (KILL Num . c) (d[Num]=Text)	→ e c (d[Num]:=NULL)	

Поддержаны рамочные конструкции для построения многоярусных условий и циклов. Более подробно со средствами управления процессами на уровне ОС можно ознакомиться в книгах по ОС.

Фрагменты	*Примечание*
$ ls doc[c-d]	Перечень файлов с именами, соответствующими маске
$ set 0 noclobber	Установка системной переменной
$ cat newletter1 newletter2 >! Oldletters	Управление потоком данных
$ at 8:15 jobs	Назначение времени запуска работ

Пример 13. Пример программы управления заданиями

Внешне языки управления процессами выглядят как нечто среднее между макроассемблерами и языками высокого уровня. Различие проявляется в понимании данных, подвергаемых обработке, и командах, к которым сводятся процессы обработки:

– роль данных выполняют файлы – объекты, обладающие собственным поведением и подверженные влиянию внешнего мира. Существование файлов в период обработки не всегда очевидно. Файлы могут участвовать одновременно в разных процессах;
– выполнение команды рассматривается как событие, которое может быть как успешным, так и неудачным. Кроме того, существуют внешние события;
– реакция на событие программируется как обработчик события, выполняемый независимо от других обработчиков (это отдельный процесс);
– программа процесса может быть нацелена не на получение результата за конечное время, а на обеспечение непрерывного обслуживания заданий на обработку объектов;
– процесс может быть активным или отложенным;
– процессы могут конкурировать за общие объекты;
– возможна синхронизация процессов и порождение подчиненных процессов;
– программа процесса выглядит как объект и создается как элемент данных, а потом может применяться равноправно с командами;
– последовательное расположение команд в программе не считается основанием для их выполнения в точно том же порядке. Выполнение команды может занимать ряд интервалов времени, между которыми выполняются другие команды.

На уровне ОС основная работа сводится к управлению заданиями, нацеленными на эффективную загрузку общего оборудования и других ресурсов. Доступ к общим ресурсам обычно регулируется с помощью очередей запросов на обслуживание имеющихся устройств и ресурсов, не только процессора. Обслуживание носит асинхронный характер. Основной критерий качества – возможность продолжить выполнение заданий без принципиальных потерь информации.

Любая программа при разработке и отладке выполняется на фоне операционной системы, управляющей процессами ввода-вывода данных ради демонстрации хода обработки данных. Поэтому минимальный контекст отлаживаемой программы – стандартный ввод-вывод, доступный по умолчанию. На уровне языка управления процессами активно используются умолчания, раскрываемые в терминах текущих значений или системных переменных. Основное отличие – укрупнение данных, переход от ячеек и строк к долгоживущим файлам. Кроме того, смягчается зависимость от последовательности вызова процессов, времени их инициирования. Переход к проблемам управления процессами влечёт радикальное изменение понятия «результат». Это не более чем код успеха/провала завершённого процесса. Возможен учёт приоритетов отложенных процессов. Появляются имена, локализованные внутри скриптов.

Таблица 25

Парадигматическая характеристика языков управления процессами

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Абстрагирование от аппаратуры, обслуживание запросов от программных инструментов, обеспечение бесперебойной эксплуатации и функционирования оборудования.
Регистры абстрактной машины	E C D E – контекст_процесса; C – текущий_процесс; D – очередь_отложенных_процессов. Результатом является код успеха или неудачи завершения процесса.
Категории команд абстрактной машины	Копирование файлов. Изменение статуса файла или его места в иерархии файлов. Установка или ввод значений переменных. Проверка условий. Запуск процесса. Вычисление выражений. Конструирование команд «на лету». Управление активностью процесса. Ожидание при взаимодействии процессов.
Реализационная прагматика	Автомат управления процессами требует реализации структуры данных для очередей, регулирующих доступ к объектам. Чаще всего используются две модели – супервизор, контролирующий взаимодействие семейства процессов, или автомат, способный тиражировать себя при ветвлении процессов. И в том и в другом случае функционирование автомата сводится к бесконечному циклу анализа происходящих событий, появление которых влечет включение обработчиков, соответствующих событиям. Проблема остановки решается вне языка на уровне базовых средств или внешним образом через прерывания.
Парадигматическая специфика	Любая программа при разработке и отладке выполняется на фоне операционной системы, управляющей процессами ввода-вывода данных ради демонстрации хода обработки данных. Поэтому минимальный контекст отлаживаемой программы – стандартный ввод-вывод, доступный по умолчанию.

Определение парадигм для ЯНУ не вызывает затруднений – в них явно видна ключевая идея, а семантические системы сравнительно изолированы в определении языка. Основные различия сосредоточены на конкретизации понятия «значение» и спектра средств укрупнения осмысленных единиц при подготовке программы.

Функциональные модели ЯНУ достаточно просты. По уровню сложности они проще SECD или SECM, т.к. не гарантируют защиту контекста. Реализационная семантика ЯНУ, как правило, требует введения дополнительных понятий (очередь, логика, словарь, точка возврата, позиция в стеке, шкала прерываний и т.п.), возникающих на уровне схем программ и программисткой терминологии.

Механизмы представления и обработки данных, накопленные в ЯНУ, в значительной мере унаследованы методами реализации ЯВУ, что позволяет локализовать изучение таких механизмов. Практика программирования на ЯНУ имеет образовательное значение. Ценящие подготовку высококвалифицированных программистов вузы, готовящие победителей международных чемпионатов по программированию, включают в начальное обучение программирование на ассемблере и управление процессами на Linux.

Таблица 26

Наследование методов ЯНУ парадигмами ЯВУ

Механизм	Парадигма	*Примечание*
Ассемблер	ИП ООП	Императивный стиль программирования, как на ассемблере, можно устроить на любом процедурно-императивном или объектно-ориентированном языке, ограничив их средства по сложности выражений и исключив иерархию структур данных и классов объектов
Стековая машина	ЯВУ	Стековые машины служат в большинстве ЯВУ основным механизмом поддержки выражений, функций и процедур с вычисляемыми параметрами
Макротехника	ЛП ФП препроцессоры	Продукционное программирование в стиле макротехники унаследовано логическим программированием, функциональное программирование переносит её на уровень работы со структурами данных. Многие системы программирования на ЯВУ используют такую технику в препроцессорах и как внутренний инструмент при кодогенерации.
Управление процессами	ООП	Механизм управления процессами наследуется парадигмой ООП, использующей взаимодействия объектов с помощью сообщений. Он существенно используется и парадигмой параллельного программирования для описания асинхронных процессов

ЯЗЫКИ ВЫСОКОГО УРОВНЯ

ЛЕКЦИЯ 4. ИМПЕРАТИВНО-ПРОЦЕДУРНОЕ 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, включая размещение локальных данных в стеке. Выполняется частичный контроль доступа к переменным и совместимости операций и операндов по типам данных с вычислением значений константных выражений (констант, переменных, элементов структур данных, результатов операций и вызовов функций), возможны оптимизирующие манипуляции по управлению вычислениями:

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

4.1. Особенности представления программ на С

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

Язык С предполагает, что программа собирается из набора файлов, содержащих фрагменты программы и библиотеки функций, подготовленные, возможно, независимо. При этом 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 изменять 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. Это оказалось серьёзным препятствием к повышению производительности труда.

–При конструировании кода программы С-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 С.

– 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 С 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. Лекция 2):

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, 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 14. Программа распечатки параметров командной строки и переменных среды на языке Си.

4.2 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.

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:

- 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.

4.3.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. Необходимо лишь четко уяснить цену такого дополнения и его преимущества, которая обычно

связывается с наследованием решений и привлечением пользователей. В первых реализациях Lisp-а 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 средство такого рода, 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 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» (Algol-like), т. к. они появились за десять лет до Паскаля. Теперь более известен Pascal).

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.):

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

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

Пример 15. Описание алгоритма на естественном языке

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

Pascal	*Примечание*

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

Пример 16. Представление программы на языке Pascal

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

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

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

Пример 17. 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 SE*T*, 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.[14] 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.

[14] Clisp – действуют на любом уровне.

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`.

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.[15]

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);	Меткой «B» помечена сборка результата.
x := cdr (x);	Шаг продвижения по списку.
goto A;	Переход на метку «A» для дальнейшего реверсирования.
end;	

Пример18. Представление программы на языке Pascal. Функция `rev` обращает все уровни списка так, что `rev` от `(A ((B C) D))` даст `((D (C B))A)`.

[15] В Clisp все ветвления работают так.

Lisp	*Примечание*

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

Пример 19. 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.

4.4. Спецификация

Таблица 28

Парадигматическая характеристика языков, поддерживающих процедурно-императивную парадигму программирования

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Программирование решений задач, обладающих точной постановкой задачи и практичным алгоритмом без претензий на тщательное исследование.
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.
Категории команд абстрактной машины	Засылка в стек. Вычисления над стеком. Пересылки в глобальную и локальную память. Передачи управления. Ветвления. Вызов процедуры. Возврат из процедуры.
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.
Парадигматическая специфика	Процедурно-императивный стиль программирования обы дополнен простыми средствами функционального программирования.

ЛЕКЦИЯ 5. 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».

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, что 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 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.

5.1. 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-исчисление 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 29

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-FRM)	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)	Специальная мультифункция над произвольным числом аргументов, каждый из которых является списком из представлений предиката и	Метод организации частичных вычислений.

	-- -- (T Z))	соответствующей ему ветви. При пустом 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 30

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 X) = X Результат совпадает с

		PRINT с побочным эффектом, заключающемся в размещении эквивалентного её аргументу текста на экране или в заданном файле.	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 «строка»	Представление самоопределимой функции без аргументов, результатом которой является её собственное представление.	Результаты такой функции хранятся непосредственно 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.	(+ 1 2 3 4) = 1+2+3+4 (* 1 2 3 4) = 1*2*3*4

		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
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 определяющему 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 кратностью вычисления тела цикла.	Поддержка традиционных схем представления программ.
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,	Оптимизация отлаженных функций по скорости выполнения.

		оттесняющего её символьное определение.	
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) = (CONS (CAR X) Y) (CONC AL BL) = (APPEND AL BL) (MAPCON FN AL) = (MAPLIST FN AL)	Поддержка синхронизации независимых участков программы, возникающих при вычислении рекурсивных функций, независимых параметров, выполнении итераций и т. п.

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. По степени общности 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. 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 Lisp-процессоры, доказавшие, что неэффективность функционального программирования обусловлена характеристиками оборудования, а не стилем программирования. Функциональные мини-языки хорошо показали себя и при решении задач аппаратного уровня. Все это превращает ФП в практичный и перспективный инструментарий. Такая схема подтверждается самой историей развития диалектов языка Lisp и родственных ему языков программирования.

Изучение функционального программирования начинается с овладения техникой работы с так называемыми «чистыми», строго математическими, идеальными функциями в области константных вычислений. 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 обучения такому стилю программирования на языке Lisp был создан язык Pure Lisp и определён его интерпретатор. Концептуально близкие идеи «структурного программирования» были сформулированы более чем через десять лет.

Если в качестве данных допускать не только значения, но и символьные формы для вычисления 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).

Функции могут быть частично определёнными, отображающими часть множества, и многозначными, хотя бы одному значению аргумента соответствует два или более значений функции. Прежде всего, понятие «функция» обогащается представлением о псевдо-функциях, которые используются с целью предоставления аппаратных, зависимых от устройств действий (ввод/вывод, сообщения, рисование и т. п.). Они фактически осуществляют 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.

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. Функциональные программы могут играть роль спецификации обычных императивно-процедурных 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 для этого не требуется даже цикла, 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 встречаются 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 является не простым делом. 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)	Сведение к вызову вспомогательной функции.
где	
F(x, y, z) =	Вспомогательная функция.
если (x = 1) то 0	Объявление значения от 1.
иначе если ((y +1) = x) то z	Проверка достижения предыдущего числа.
иначе F(x, y +1, z +1)	Шаг рекурсии с наращиванием параметров.

Пример. 20. Выражение «-1» через «+1»

Решение получилось через введение формально усложненной функции F со вспомогательными параметрами, которое противоречит интуитивному стремлению к монотонности при движении от простого к сложному. Возможны ограничения на типы данных, допускаемых в качестве аргументов, в таком случае речь идет о частичных функциях. Эти 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:

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

5.2. Функциональные ЯП

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.

Языки ФП достаточно разнообразны. Существует и активно применяется более трехсот диалектов Lisp-а 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 различных языков программирования, в числе которых Haskell, Sml, Scheme, Mondrian, Mercury, Perl, Oberon, Component Pascal, Разработан F# – новый язык ФП. 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.

5.3. Отображения и функционалы

*Выразительная сила ЯФП наиболее результативно проявляется в пр*ограммировании 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. Так работает любое введение обозначений – от знака 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:

– где размещается множество полученных результатов;
– чем отличаются нужные результаты от полученных попутно;
– 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 называют функционалом. 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 отображения может быть разбито на части (функции и функционалы) разного назначения, типичного для многих схем информационной обработки. Это позволяет упрощать отладку систем определений, повышать коэффициент повторного использования

отлаженных функций. Можно сказать, что отображения – эффективный механизм абстрагирования, моделирования, проектирования и формализации крупномасштабной обработки информации. Возможности отображений в информатике значительно шире, чем освоено практическим программированием, но их применение требует дополнительных пояснений.

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, представления 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) [16]	Следующие числа
(map-el #'CAR xl)	«головы» элементов = CAR
(map-el #'length xl)	Длины элементов

Пример 21. Отображение элементов списка с помощью заданной функции

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

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

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 голов AL 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 22. 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 23. 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.

5.4. Отложенные действия

Императивная организация вычислений по принципу немедленного и обязательного выполнения каждой очередной команды не всегда результативна. 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 ядре 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 31

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 32

Дополнение АМ для эффективного выполнения отложенных действий

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] [17]
([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)

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))`
	Вырезка начального отрезка

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

```
         )  )       )
   )       )     )
(def endless (LAMBDA (m)
   (cons m (lambda ()
                    (endless (cons m m)) ))
)             )
(beg '(a b c) (endless 'A))
          ; = (A (A . A) ((A . A) A . A ))
```

Пример 24. Конструирование последовательности произвольной длины из бесконечного списка

5.5. 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 Lisp-интерпретатора, написанном на Lisp-е.

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`.

5.6. Гибкий интерпретатор

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

Ради удобочитаемости здесь уменьшена диагностичность, нет пост-вычислений и формы PROG. Lisp 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)))) (ASSOC e al)))) ((EQ (CAR e) 'QUOTE) (CAR (CDR e)))	 Диагностика Однократность вычисления значения

((EQ (CAR e) 'FUNCTION) (LIST 'CLOSURE (CADR fn) al)) ((EQ (CAR e) 'COND) (EVCON (CDR e) al)) (T (apply (CAR e)(evlis (CDR e) al) al))))	Замыкание функционального аргумента

Пример 25. 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 26. 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 27. 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 28. Вычисление параметров

5.7. 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 29. 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 макрофункций нет необходимости в вычислении аргументов

Пример 30. Специальные макрофункции 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 31. 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 32. 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 ещё и 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.

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))))	Элементарные функции
)	Программируемые

((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)))))	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 33. Самоописание Lisp-интерпретатора, предложенное Дж. Маккарти в начале 1960-х годов

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 использование 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 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.

5.8. Спецификация

Таблица 32

Парадигматическая характеристика языков, поддерживающих функциональное программирование

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Исследование потенциального максимума возможностей решения сравнительно новой задачи.
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.
Категории команд абстрактной машины	Как в Pure Lisp.
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.
Парадигматическая специфика	Система программирования обычно поддерживает механизмы других парадигм, возможно выделенные в отдельные подсистемы (монады в языке Haskell).

ЛЕКЦИЯ 6. ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ

Исследовательская и проектная работы обычно проходят 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:

$$H :- B1_1, \ldots, BN_n,$$

понимаемую как логическое следование

$$\text{if } (B1_1 \text{ and } \ldots \text{ and } BN) \text{ then } H$$

или

$$B11 \ \& \ \ldots \ \& \ BN \rightarrow H$$

H называют головой правила, а $B1_1, \ldots, BN$ – 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 макротехникой. Для повышения эффективности программ программисту следует знать особенности поведения механизма вычислений и границы вычислимости используемых выражений.

6.1. Операционная семантика

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

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.

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.

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

```
s e c d r → s' e' c' d' r'
```

Таблица 33

Расширение абстрактной машины для выполнения альтернатив ЛП

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

Таблица 34

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

6.2. 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. Однако эти понятия не очень усложнятся, 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 варианта или собрать все беступиковые варианты. Второе можно сделать, используя отображения (map), а первое допускает метод первого подходящего, чо можно реализовать как слегка модифицированный evcon с добавочной ловушкой на прерывание при достижении успеха.

Более сложно обеспечить равновероятность выбора вариантов. Наиболее серьезно возможность такой реализации рассматривалась в проекте языка *SETL*. 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.

6.3. 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.

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.

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 34. Вычисление факториала

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 35. Быстрая сортировка

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 36. Отображение

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 37. Расширяемый скелет Prolog-интерпретатора.

Определение	*Примечание*
provable(true, _) :- !.	Доказуемость

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 38. Выводимость цели. Обобщение интерпретатора

В определении функции `provable(Goal, Defs)` вырабатывается истина, если цель `Goal` выводима по отношению к `Defs`, представленным списком клауз в форме *Head-Body*. Полная реализация бектрекинга требует детерминированного накопления результатов. Отдельная альтернатива представляется как список целей и ветвей, которые могут использоваться как список альтернатив.

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

Пример 39. Бэктрекинг. Возвраты при неудачном выборе клаузы

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. Однако 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, поэтому 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.).

6.4. 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.

Уточнённую схему 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 40. 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 41. 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 42. 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 форма для накопления всех результатов.

Пример 43. 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 44. Пересечение множеств A и B

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

Пример 45. `b` вычисляется лишь при истинном a, что результативно, но не всегда соответствует интуитивным ожиданиям (логика, предложенная в свое время Маккарти, 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 46. 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 47. `(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 48. a+b+c = (a+b)+c = a+(b+c) = (a+c)+b – профилактика переполнения

6.5. Модели недетерминизма

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 49. Механизм обработки событий throw-catch как модель недетерминизма вариантов

Следует отметить неисчерпаемый ряд задач, при решении которых удобно используются недетерминизм.

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.

В истории ЛП можно выделить 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.

6.6. Спецификация

Таблица 35

Парадигматическая характеристика языков логического программирования

Параметр	*Конкретика*
эксплуатационная прагматика ЯП	Неопределенные постановки задач. Эмпирическое накопление рецептов, достаточное для решения некоторых практических задач.
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.
категории команд абстрактной машины	Кроме типов команд, характерных для ФП: - выбор варианта; - восстановление контекста при переборе вариантов.
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.
парадигматическая специфика	Расширение класса решаемых задач использованием недетерминированных моделей.

ЛЕКЦИЯ 7. 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.

Прежде всего, необходимость в целостном решении полной задачи «с нуля» теперь встречается достаточно редко. 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. Обычно сложные задачи декомпозируются на модули, разрабатываемые отдельно, но функционировать модулям предстоит совместно. Кроме того, снижение трудозатрат на создание программ достигается программированием многократно используемых компонентов, 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 обработки 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 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.

7.1. 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.

Однако 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 Lisp-е 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 развитии операционной системы UNIX как работоспособный принцип декомпозиции программ. Но существуют задачи, например, реализация систем программирования, в которых прямое следование такому принципу может противоречить требованиям к производительности. Возможен компромисс «отдельная работа программируется отдельно, а выполняется взаимосвязано с другими работами», 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.

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.

7.2. 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>, где

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.

Обозначения:
<[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 36

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

Формат команд АМ имеет вид:

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

Таблица 37

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 связан с упрощением процесса отладки, который 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 мета-надстройка над ООП [48]. Отдельный ряд трудностей вызывают приаппаратные 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.

7.3. 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 ацикличного 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. Важно принять во внимание следующее:

- 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 50. Программа на языке С++

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 51. 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 52. 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 C выглядят как появление ряда новых спецификаторов, с помощью 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.

Фрагмент	*Пояснение*

class complex {
  double re, im;
 public:
  complex(double r, double i) { re=r; im=i; }
  friend complex operator+(complex, complex);
  friend complex operator*(complex, complex);
};
	Объявление класса объектов с закрытыми полями и общедоступными конструкторами и перегрузкой операций.

Пример 53. Перегрузка арифметических операций для их использования в выражениях над комплексными числами

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*

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 обработка чисел с помощью перегруженных операций.

Пример 54. 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 55. 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 данных новой категории – объектов.

7.4. Функциональные модели ООП

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. 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. П. Грем приводит 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 или функцией категории FSUBR.

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 просто класс с одним-единственным предком. При переходе к векторной реализации разделение на классы и экземпляры становится реальным, также становится невозможным превращать экземпляры в классы простым изменением свойства.

Более прозрачная модель ООП на базе обычных списков, иллюстрирующая глубинное родство ФП и ООП, реализована в системе CLOS. Показанный ниже пример 56 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. Рассмотрим примеры использования *к*лассов и экземпляров 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)))	Метод разбора выражений с триадными операциями.

(SETF (ft x) (CAR tl)) (SETF (sd x) (cadr tl)) (SETF (td x) (caddr tl)) (SETF (td x) hd)) (:documentation "распаковка выражения"))	
(defmethod compl ((hd (eql 'IF)) (tl expr) N) (let ((then (list (compl(sd tl)N) '(JOIN))) (else (list (compl(td tl)N) '(JOIN)))) (append (compl(ft tl)N) (list 'SEL then else)))(:documentation "сборка кода"))	Метод компиляции выражений с триадными операциями.
(defmethod parh ((x expt)) (let (ftx (ft x)) (COND ((ATOM ftx) (spread 'ADR ftx)) ((member (CAR ftx) '(QUOTE CAR CONS + IF LAMBDA LABEL LET)) (spread (CAR ftx) (CDR ftx)) (T (spread 'OTHER ftx))))(:documentation "шаг разбора"))	Метод разбора произвольных выражений.

Пример 56. OO-определение Lisp-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 Lisp-а. ООП – это одна из вещей, которую Lisp изначально умеет делать. Для функционального стиля программирования в переходе к ООП нет ничего революционного. Такой переход практически не расширяет пространство решений. Это просто небольшая конкретизация механизмов перебора ветвей функциональных объектов. При переходе от императивного стиля ООП компенсирует избыточную целостность представления отлаживаемых программ, смягчает жесткость зависимости компонентов программы от потока информационных процессов.

Более интересный вопрос, что же еще может дать функциональный стиль и традиция реализации функциональных систем программирования? – Похоже, что это средства освоения новых архитектур и технологий.

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.

7.5. Спецификация

Таблица 38

Парадигматическая характеристика парадигмы ООП

Параметр	*Конкретика*
Эксплуатационная прагматика ЯП	Практичное программирование, нацеленное на разумный компромисс в пространстве противоречивых критериев с приоритетом критериям сферы приложения программ.
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.
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).
Реализационная прагмати	Сочетание статического представления методов с динамикой размещения объектов, включая автоматизацию повторного использования памяти. Представление сигнатуры для динамического выбора конкретного метода обработки объект.
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.

7.6. Мультипарадигмальные 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 монопарадигматического 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, макрогенераторов или организации выхода на уровень операционной системы.

Таблица 39

Сравнение прагматических особенностей ПП

ПП	*Иерархическая декомпозиция программы*	*Укрупнение конструкций*	*Память и результат*	*Реализационная прагматика*
ИП	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 программировани я	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 40

Специфика базовых семантических систем
в разных парадигмах программирования на ЯВУ

ПП	*Вычисления*	*Память*	*Управление*	*Структуры*
ИП	Скаляры = слова СД-ТД предвычисления	Статическое распределение := Области видимости - динамика вызовов - статика вложенности текста - глобалы: 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 Динамика вызовов – БД (атомы) Списки свойств Let	Quote-eval Lazy Cond Lambda Defun Evalquote Ловушки	Списки Файлы Строки Потоки Модели массивов, таблиц,

		Declare	Мультизначения Параллелизм Необязательные параметры	вариантов
ЛП	Варианты с возвратами Шаблоны в строке Подстановка	Накопление рецепто	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

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

– 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:
– 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, C 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.

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

Изначально императивно-процедурный C 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:

– полный свод обычных механизмов функционального программирования, включая функции высших порядков; каррирование и сопоставление с образцом;
– динамическое связывание;
– ленивые и энергичные вычисления;
– замыкания функций и мемоизация;
– квотирование выражений;
– конструирование выражений, частичное применение функции и мета-компиляция;
– разреженные матрицы;
– хеш-таблицы;
– mutable-переменные (изменяемые);
– 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:

– 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 типозависимых 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.

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.

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

ЛЕКЦИЯ 8. ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ

Прогресс производства аппаратуры намного опередил развитие технологий программирования. Параллельные архитектуры теперь позволяют принципиально повышать производительность программ решения многих задач. Методы автоматического распараллеливания программ способны обеспечить значительное ускорение вычислений, явно сводимых к комплексу независимых процессов обработки элементов векторов, но отступают перед программами более широкого класса. Перспектива выражения языковыми средствами параллелизма на уровне постановки новой задачи приторможена проблемами оптимизирующей верифицирующей компиляции, обеспечивающей аккуратный выбор эффективной и надежной схемы параллелизма. Кроме того, трудоёмкость повторного программирования и отладки параллельных алгоритмов обуславливает рассмотрение готовых последовательных программ в качестве эталонов при оценке результатов параллельных вычислений, что сдерживает развитие языков параллельного программирования и методов их реализации.

В настоящее время осознана актуальность формирования парадигмы параллельного программирования, вызванная расширением и развитием системы базовых понятий, необходимых для рациональной разработки систем управления процессами на современной аппаратуре. Парадигмы в программировании характеризуются стилем мышления при решении задачи, системой используемых понятий и особенностями их реализации. Можно констатировать, что стиль мышления и система понятий для параллельного программирования уже сложились в процессе эволюции языков программирования, но типовая их поддержка в системах программирования ещё не сформировалась

«Ничем не скроешь фундаментальную трудность параллелизма» – так прозвучало заявление Мартина Одерски (Martin Odersky) в его приглашенном докладе на 20-й Международной конференции «Конструирование компиляторов», состоявшейся в марте 2011 года в Германии под председательством Дженса Кнопа (Jens Knoop) [21]. Несмотря на многочисленные призывы и конкурсы, пока не появилось общих идей по новому поколению языков и систем программирования (ЯСП), в которых параллелизм имел бы самостоятельное значение, а не рассматривался бы (по понятным причинам) как пристройка к традиционному программированию.

8.1. Пространство решений

Рост интереса к параллельному программированию в наши дни связан с переходом к массовому производству многоядерных архитектур. Ознакомление с понятиями и явлениями параллельного программирования в данном материале нацелено на профилактику прочного привыкания к принципам традиционного последовательного программирования. Такая профилактика необходима для специализации в области разработки распределенных информационных систем, а также приложений для суперкомпьютеров, многопроцессорных конфигураций и графических процессоров.

Современное развитие ЯСП на практике ориентировано на решение задач параллельного программирования. Как правило, новые ЯСП включают в себя библиотечные модули, обеспечивающие организацию процессов, или подъязыки, допускающие многопоточное программирование. Это не исключает реальную практику ручного распараллеливания ранее отлаженных обычных программ, приведения их к виду, удобному для применения производственных систем поддержки параллельных вычислений. Значительная часть таких работ носит технический характер и заключается в систематической реорганизации структур данных, изменении статуса переменных и включении в программу аннотаций, сообщающих компилятору об информационно-логических взаимосвязях. Существенным ограничением результата ручного распараллеливания является не только опасность повторной отладки алгоритма, но и его жесткая зависимость от характеристик целевой архитектуры.

Разнообразие моделей параллельных вычислений и расширение спектра доступной архитектуры следует рассматривать как вызов разработчикам ЯСП, способных решать проблему создания методов компиляции многопоточных программ для многопроцессорных конфигураций. Язык должен допускать представление любых моделей параллелизма, проявляемого на уровне решаемой задачи или реализуемого с помощью реальной архитектуры, причем такое представление требует лаконичных форм и конструктивных построений, гарантирующих сохранение свойств программ при их реорганизации. Не менее важна расширяемость языка по мере развития средств и методов параллельных вычислений, темп которого превышает скорость осознания специалистами их возможностей.

Рассматривая задачу формализации языков параллельного программирования как путь к решению проблемы адаптации программ к различным особенностям используемых многопроцессорных комплексов и

многоядерных процессоров, мы видим, что решение этой проблемы требует разработки новых методов компиляции программ, особенно масштабируемой кодогенерации, и развития средств и методов ясного описания семантики языков параллельного программирования.

8.2. Параллельные алгоритмы

Обзор решений, встречающихся при организации параллельных процессов для реализации параллельных алгоритмов, даёт некоторое представление о разнообразии многопроцессорных архитектур и систем:

- процессоров может быть много, и они могут обладать разными системами команд;
- процессоры могут работать автономно или сообща выполнять общую работу;
- отдельные процессоры могут функционировать вхолостую, не выполняя полезной работы;
- процессоры имеют свою локальную память и могут работать в общей памяти;
- распределение действий по процессорам может быть предписано программой, а может осуществляться супервизором, поддерживающим определенную модель вычислений.

Такое пространство допускает естественное развитие в направлении моделирования асинхронных процессов, а также помогает уяснить, что существуют контексты исполнения программ, от которых зависит выполнимость команд. При анализе результатов программ обнаруживаются разные категории ошибок управления и выделяются невыполнимые действия, несмотря на то, что все образующие их команды выполнимы. Проявляется роль частичного упорядочения действий в обеспечении работоспособности программ средствами управления вычислениями, отражающего неимперативность последовательного исполнения действий при организации параллельных процессов. Оптимизирующие компиляторы любую программу нередко рассматривают как мультипрограмму.

Переход к параллельным алгоритмам влечёт пересмотр содержания многих понятий и введение новых терминов, отражающих разного рода явления и эффекты, не имевшие особого значения для обычных последовательных алгоритмов. Прошли времена, когда параллельное программирование называли неалгоритмическим. Теперь термин «параллельный алгоритм» связан с пониманием того, что алгоритм может выполняться как одним, так и группой исполнителей. Такое понимание

резко расширяет пространство решений задач, меняет подходы к реализации решений, использующих параллелизм, и в некоторой мере сказывается собственно на постановке задач и планировании жизненного цикла программ решения задач, ориентированных на использование параллелизма.

Прежде всего, следует прояснить следующие вопросы, связанные с формулировкой постановки задачи:

1. Насколько изменится трудоёмкость жизненного цикла программы решения задачи с помощью параллельного алгоритма?
2. В какой мере при постановке задачи следует учитывать модель параллелизма?
3. Как обосновать и измерить выигрыш от разработки параллельного алгоритма?
4. Насколько изменяется постановка задачи при переходе к параллельным алгоритмам?
5. Что даёт парадигма параллельного программирования на уровне разработки параллельного алгоритма?
6. Какими средствами представляются разрабатываемые параллельные алгоритмы решения задачи на этапе, предшествующем разработке программы?

За полвека традиционного последовательного программирования отлажено колоссальное количество программ, аккумулированных в системы программирования и стандартные библиотеки. Изменение постановок задач, уже имеющих готовые отлаженные программные решения, ради учёта допустимого параллелизма чревато повторным программированием и, что гораздо более трудоёмко, повторной отладкой.

Основной аргумент – целесообразность учета естественного параллелизма на уровне постановки задачи, утрачиваемого при решении задачи посредством обычных алгоритмов. Число языков параллельного программирования, удобных для реализации параллельных алгоритмов год от года растёт, хотя и их применение решает не все проблемы организации параллельных вычислений.

Итак, параллельный алгоритм может быть реализован по частям на множестве различных устройств с последующим объединением полученных результатов и получением целевого результата. Возникают практические вопросы:

- Каким образом в определении алгоритма выделены части, выполняемые отдельными устройствами?
- Обязана ли реализация алгоритма использовать в точности представленный в его определении набор устройств?

– Можно ли последовательный алгоритм рассматривать как параллельный, исполняемый на одном устройстве?

Следующая обойма вопросов касается категории «время» и связана с проблемами синхронизации:

– Могут ли части параллельного алгоритма обладать своим независимым или централизованным отсчетом времени?
– Может ли синхронизация частей алгоритма противоречить его информационным связям и логике управления?
– Можно ли синхронизацию частей алгоритма рассматривать как частный случай асинхронности?

Особые сложности параллелизма вызывают вопросы доступа к памяти:
– Каким образом взаимодействующие части параллельного алгоритма обмениваются данными?
– Могут ли части параллельного алгоритма изменять состояние общей памяти и памяти других частей?
– Может ли часть параллельного алгоритма воспрепятствовать использованию своей памяти другими частями?

Кроме того, части алгоритма могут быть определены в разных моделях вычислений и над разными структурами данных. Оценка результата разработки параллельного алгоритма, кроме оценки сложности вычислений и объёма данных, осложнена целесообразностью оценивать выполнение разного рода трудно формализуемых критериев, часть которых, однако, поддаётся современным средствам верификации на моделях.

И всё же, сколь ни сложен мир параллелизма, программистам предстоит его понять и освоить!

Интересно отметить появление новых архитектур, обладающих полным набором команд с условным исполнением.

Исследования в области информатики в настоящее время претерпевают изменение системы понятий, используемой в практических ИТ. В этом процессе расширяется пространство решений сложных задач, модернизируются методы развития информационных систем (ИС) на основе компьютерных сетей и многопроцессорных комплексов.

Интересно рассмотреть перспективы развития парадигм программирования, обусловленных изменением условий эксплуатации современных информационных систем, особенно связанных с повсеместным распространением сетевых технологий, меняющих критерии

оценки качества программ и методы обеспечения надежности и производительности программирования. Две основные линии такого развития – разработка распределенных информационных систем (РИС) и компонентное программирование (КП).

При практичном решении проблемы разработки языков параллельного программирования и подходов к их реализации в центре внимания вопросы выбора лаконичных форм представления программ, обеспечения конструктивных построений, гарантирующих сохранение правильности программ при их реорганизации, и поддержки расширяемости языка по мере развития средств и методов параллельного программирования.

Описание парадигмы параллельного программирования отражает прагматические различия в условиях реализации и применения изобразительных средств, используемых в жизненном цикле программ. Парадигмы параллельного программирования занимают нишу, связанную с реализацией программ выполнения вычислений на многопроцессорных системах для организации высокопроизводительных вычислений. Эта ниша обременена резким повышением трудоемкости отладки программ, вызванной комбинаторикой выполнения фрагментов асинхронных процессов.

Варьирование правил функционирования сетей допускает как асинхронную, так и синхронную организацию срабатывания действий, включая дозирование нагрузки и специализацию процессоров и распределение действий по потокам выполнения. Использование иерархических, многоуровневых, структурированных и расширяемых сетей обеспечивает моделирование практически любых, накопленных в языках программирования, техник программирования и представления структур данных.

8.3. Практичные системы программирования

При измерении производительности суперкомпьютеров и в экспериментах с распараллеливанием программ активно используются задачи научных расчётов, преимущественно реализующих алгоритмы векторной обработки данных, удобно распараллеливаемых с помощью MPI. Практические задачи современного параллельного программирования обычно выглядят как приведение больших отлаженных ранее программ на C или Fortran-е к форме, дающей выигрыш от распараллеливания с помощью штатных средств, включаемых в доступные системы программирования. В целом такая работа сводится к следующим видам работы:

– разметка участков программы, допускающих автоматическое распараллеливание;
– анализ участков и причин, препятствующих распараллеливанию программы;
– выбор участков программы, допускающих их техническое приведение к форме, пригодной для распараллеливания;
– изобретение рецептов полуручного преобразования текста программы с целью расширения возможностей распараллеливания;
– приведение текста программы к предельно распараллеливаемой форме;
– прогон распараллеленной версии программы для оценки выигрыша от параллелизма;
– частичное перепрограммирование и отладка фрагментов программы для исключения или смягчения эффектов, препятствующих достижению нужных характеристик производительности;
– установление частичной функциональной эквивалентности исходной программы и результирующей её версии.

Обычно компилятор поддерживает оптимизации, обеспечивающие устранение неиспользуемого кода, чистку циклов, слияние общих подвыражений, перенос участков повторяемости для обеспечения однородности распараллеливаемых ветвей, раскрутку или разбиение цикла, втягивание константных вычислений, уменьшение силы операций, удаление копий агрегатных конструкций и др.

Рассматривается зависимость ускорения вычислений от числа процессоров и объема общей и распределенной памяти. Выполняется систематическая замена рекурсии на циклы. Предпочитается однородное пространство процессоров, общая память, быстрые обмены, соседство, гарантирующие улучшение производительности систем для высокопроизводительных вычислений.

Заметно влияние дисциплины работы с памятью на характеристики параллельных процессов. Используется защищенная и размазанная память. Различны решения, принятые в разных языках программирования, по работе с многоуровневой и разнородной памятью (доступ, побочный эффект, реплики, дубли и копии). Обработка транзакций становится одной из типовых семантик работы с памятью в языках программирования.

Предлагаются средства и методы типизации управления процессами, удобными для подготовки программ, ориентированных на исполнение с помощью Open MP или MPI. Идеи языков программирования по сетевому представлению типов управления и дисциплины работы с памятью ещё не получили удобной системной поддержки.

Компонентно-ориентированная разработка ПО может следовать единому сетевому определению семантики языков программирования и процесса разработки программ.

Некоторую поддержку работ по подготовке программ, использующих параллельные процессы, обеспечивает созданная фирмой Microsoft технология .NET, позволяющая применять общие системные библиотеки в программах на разных языках программирования.

Лидирующие в области поддержки параллелизма фирмы Intel и Microsoft предлагают разработчикам высокопроизводительных вычислений новые решения, направленные на преобразования программ в процессе их создания и распараллеливания:

– помощник по параллельному программированию (Parallel Advisor XE);
– компиляторы-отладчики и библиотеки (Parallel Composer XE2011);
– анализатор потоков и памяти (Parallel Inspector XE);
– профилировщик производительности (Parallel Amplifier XE).

Помощник по параллельному программированию анализирует исходный текст программы на языках Fortran или C до ее компиляции и отмечает узкие места, препятствующие распараллеливанию.

Анализатор потоков и памяти исследует полученный в результате компиляции объектный код и выявляет неудачное распределение ресурсов, снижающее производительность программы.

Профилировщик производительности позволяет наблюдать экспериментальную проверку достигнутой производительности программы.

Технология применения всех этих инструментов предполагает, что программист по ходу дела многократно принимает решения об изменении исходного кода программы и вручную включает в него необходимые прагмы, указывающие компилятору допустимость оптимизирующих преобразований, что делает работу компилятора по распараллеливанию программы проще и надежнее. Возникает проблема автоматизации преобразования программ с целью приведения их к виду, удобному для распараллеливания компилятором и настройки объектного кода на конкретную многопроцессорную конфигурацию. Часть проблем решает схема подготовки параллельных программ с использованием библиотек преобразований исходного и объектного кода.

При демонстрации новых средств поддержки параллельного программирования представители фирм-разработчиков компиляторов показывают, насколько просто выполняется каждое такое преобразование в отдельной позиции программы. Проблемой является то, что таких позиций

в нужной программе может быть не десятки и сотни, а тысячи и десятки тысяч. Отыскать их все без специальных инструментов проблематично. Кроме того, при выполнении таких работ возникает необходимость повторной разработки структур данных и схем управления вычислениями, а также обоснованного выбора средств реализации программного конструктива.

Так, например, возникает необходимость устранения или преобразования ряда механизмов работы с глобальными переменными, общими блоками данных, глубокой вложенности процедур, перехода от рекурсивных вызовов к итерациям со статическим управлением циклом и другое. Кроме того, возникает целый спектр аналитических выкладок по установлению фактических областей видимости имён, выбор между общей памятью и памятью конкретного потока. Выполнение таких работ вручную связано с риском порождения типичных для параллельных программ ошибок, таких как гонки памяти, конфликт доступа, взаимоблокировка и т. п.

Реорганизация векторов для нужд эффективного распараллеливания может идти разными путями: разделение на чётные-нечётные элементы, разбиение на половины или четверти, распределение по частям, требующим разной интенсивности вычислений и т. п.

Иногда такие проблемы можно решить на уровне макропреобразований текста программы, выполняемых препроцессорами, – необходимо лишь разработать систему подходящих конкретной программе макроопределений. Такие системы обычно не обладают универсальностью: они отражают индивидуальный стиль программирования, учитывают особенности мнемоники и неявные границы схем управления вычисления. В результате создаётся многопоточная версия программы, строго говоря, не являющаяся эквивалентом исходной программы, а лишь эквивалентно строящая её основные результаты.

Особый круг проблем связан с навыками учёта особенностей многоуровневой памяти в многопроцессорных системах. Обычное последовательное программирование такие проблемы может не замечать, полагаясь на решения компилятора, располагающего статической информацией о типах используемых данных и способного при необходимости выполнить оптимизирующие преобразования программы.

Следует отметить, что использование языков параллельного программирования в качестве языка представления исходной программы не гарантирует её приспособленность к удачному распараллеливанию.

При анализе пригодности программы к распараллеливанию анализируются следующие потенциальные зоны риска:

– объявление и использование глобальных переменных;
– области видимости переменных и констант;
– определения распараллеливаемых процедур, содержащие внутренние вызовы других процедур;
– рекурсивные определения;
– заголовки циклов, управляющие кратностью выполнения распараллеливаемого тела цикла;
– ветвления и переключатели, дающие развилки потоков;
– обработчики исключений;
– реакции на события и сообщения;
– позиции возможного обмена данными между потоками;
– тиражирование данных в общей памяти и в файлах;
– списки параметров функций, возможно преобразуемые в данные в общей памяти;
– декомпозиция управления циклом и реорганизация соответствующих структур данных;
– границы участков изменения значений переменных;
– переменные для значений промежуточных вычислений;
– многократно выполняемые идентичные вычисления – избыточные;
– инкрементные переменные, динамику изменения которых желательно сохранять для пост-анализа;
– позиции диагностической аранжировки для пост-анализа;
– фактически не задействованные переменные и константные или не используемые вычисления;
– параметризация зависимостей частей многопоточной программы от номера потока.

Вытекающие из такого анализа преобразования программы могут быть выполнены в стиле символьной обработки данных с помощью техники лексического и синтаксического анализа, регулярных выражений и макропреобразований. Чисто технические трудности связаны с разнообразием типов преобразований. Практики придерживаются традиционной методики отладки, т. е. разделяют процесс преобразований на шаги, на каждом из которых выполняется ровно один тип преобразований, после которого выполняется прогон программы на имеющихся тестовых данных с целью удостоверения сохранения частичной функциональной эквивалентности.

Таким образом, техника приведения программ к распараллеливаемой форме меняет стиль применения систем программирования, требует программистской работы на уровне грани между разбором программы и

кодогенерацией, а также обустройства библиотек преобразований программы и её оптимизирующих преобразований.

Важно учесть изменение критериев применимости оптимизирующих преобразований. Если устранение избыточных вычислений в последовательной программе обычно оценивается как улучшение, то в параллельной программе оно может оказаться пессимизацией из-за появления лишних взаимосвязей между потоками и доступа к общей памяти.

Следует принять во внимание, что, хотя основная трудоёмкость жизненного цикла программ связана с тестированием и отладкой программ, именно это направление за полвека профессионального программирования практически не получило должного прогресса. Если первый барьер к успеху в параллельном программировании обусловлен последовательным стилем мышления, то второй очевидно зависит от до сих пор не преодолённой трудоёмкости отладки. Основные результаты здесь получены в форме парадигм функционального, структурного и объектно-ориентированного программирования и технологий поддержки непрерывной работоспособности разрабатываемой программы, смягчающих сложность тестирования, развития и версифицирования программного продукта.

Отладочные средства современных визуальных оболочек систем программирования, как и в ранние времена последовательного программирования, предоставляют для отладки параллельных программ средства выполнения в пошаговом режиме, отслеживания значений переменных и установления точек приостановок, для работы с которыми программист должен заранее подготовить комплекты тестов для всех участков программы и контроля корректности значений внутренних переменных. Возможно указание условий, при которых отладчик произведёт приостановку программы и обеспечит воспроизводимость повторного прогона программы.

Специфику параллельного программирования отражает лишь возможность переключения потоков, обнаружения тупиков и потерянных сигналов, совместно используемых (разделяемых) данных, реентерабельных процедур, вызываемых повторно из другого потока до их завершения.

Для целей отладки выполняется компиляция специальной отладочной версии программы, что говорит о необходимости дополнительной отладки программы уже без отладочного сервиса. Впрочем, известны средства фоновой отладки, обеспечивающее слежение за отлаживаемой программой из независимого смежного процесса уровня ОС.

Лишь в последние годы в работах ИСП РАН наметилось более серьёзное отношение к проблемам отладки параллельных программ, что представлено

в форме появления теорий для обработки информации, подверженной искажениям, и методики сравнительной отладки программ, в которой задействовано понятие эталонной программы и схемы распределённого отладчика.

8.4. Модели параллелизма в языках программирования

История языков программирования накопила целый ряд примеров лаконичных форм представления программ, начиная с умолчаний, неявных циклов и операторов ввода-вывода в языке Fortran. С точностью до реализационной прагматики, при разработке языков параллельного программирования можно унаследовать языковые конструкции и механизмы из привычных парадигм программирования и зарекомендовавших себя языков параллельного программирования.

Прежде всего, это алгебраические механизмы распространения функций и операций относительно структур данных, предложенные в первом языке параллельного программирования APL. Дальнейшее упрощение изобразительных средств управления параллелизмом дает предложенный в языке Sisal подход к неявному распараллеливанию циклов на основе построения пространства итераций по пространству обрабатываемых данных.

Ещё острее становятся проблемы разработки, отладки, тестирования и верификации параллельных программ по сложности весьма превосходящие обычное программирование. Именно здесь сосредоточена исследовательская активность современного языкотворчества и системного программирования, поиск средств и методов интеграции удачного опыта параллельного программирования и успешного обучения методам параллельных вычислений. Акцент на анализ подходящих решений в области разработки переносимых программ, средств визуализации процессов, типизации управления и дисциплины доступа к памяти.

Практически в мире параллелизма все базовые понятия программирования претерпевают изменение или расширение (программа, ветвление, цикл, событие, память, результат). Появляется ряд специфических для параллельного программирования понятий (процессор, поток, ожидание, длительность, фильтр, барьер). Программы становятся многопоточными, циклы – параллельными, память обретает копии и реплики, события происходят одновременно в разных синхронизуемых процессах, вычисление результата может не означать завершение процесса. Такая ревизия понятий влияет не только на стиль программирования, но и

изменяет характер компиляции на этапе генерации кода программы. Возрастает роль техники использования многократно используемых компонент схемного уровня, соответствующего средствам типизации управления процессами.

Существуют версии ряда стандартных языков императивного программирования, приспособленные к выражению взаимодействия последовательных процессов в предположении, что в каждый момент времени существует лишь один процесс. При таком подходе в программе выделяются критические интервалы, учет которых полезен при распараллеливании программ. Многие традиционные языки программирования приспособлены к выражению локального параллелизма с помощью специальных расширений или библиотечных функций, обеспечивающих выделение участков с независимыми действиями, пригодными для распараллеливания компилятором.

Большие надежды связаны со строго функциональным подходом к спецификации параллельных программ и типов данных в языке с предпочтением так называемой «ленивой» схемы вычислений. Противопоставление достоинств и недостатков «ленивых» и «энергичных» методов вычислений отчасти смягчается концепцией «монад» в строго функциональном языке программирования Haskell. Особенности определения семантики языковых конструкций по-прежнему не отражают решение проблем обеспечения удобочитаемости программ и их отладки. Табулирование сложных вычислений, называемое «мемоизация», становится популярным практичным инструментом снижения сложности вычислений.

В настоящее время существуют сотни функциональных языков программирования, ориентированных на разные классы задач параллельного программирования. Языки функционального программирования обогатились типовыми средствами практически всех известных подходов к представлению программ и организации вычислительного эксперимента и информационных процессов. Обеспечена организация параллельных процессов. Возможна визуализация данных и программ. Имеются средства стандартного и объектно-ориентированного программирования. Поддержано управление компиляцией и конструирование компиляторов. На сегодняшний день утратили актуальность опасения относительно ресурсно-эксплуатационных трудностей функционального программирования. Функциональное программирования вносит свой вклад техникой рекурсивных определений и отображений, параллелизмом обработки аргументов функций, ленивых

вычислений, а также сведением понятия «условия» в ветвлениях к понятиям «страж» или «образец».

В семействе практичных языков функционального программирования заметное место занимают языки организации распределенных и параллельных вычислений. Практики с большой похвалой отзываются о языке функционального программирования *Erlang* фирмы Ericsson.

Fortran - сопрограммы

Средства представления параллельных вычислений доступны, начиная с первых языков высокого уровня. Языки Fortran II и Fortran IV были достаточно универсальны для представления программ организации параллельных процессов. Механизм сопрограмм, допускающий многовходовые (Entry) процедуры, позволял представлять программы взаимодействующих процессов и декомпозировать программу на управляющую и вычисляющую части, выглядящие как независимые компоненты программы, но это не привело к практике параллельного программирования и постепенно сопрограммы превратились модули, обеспечивающие представление иерархии функций подобно иерархии классов в ООП. Синтаксически средства параллельного программирования выглядят в современных Fortran-программах как разметка текста ключевыми словами и вызовы библиотечных функций.

Lisp – отложенные вычисления

Унификация представления данных и программ позволила пополнить передачу параметров по заранее вычисленному значению или указателю возможностью организовывать передачу представления выражений, вычисление которых можно выполнять по мере необходимости. Такая возможность привела к концепции «ленивых» вычислений и развитию методов динамической оптимизации процессов в отличие от статической оптимизации программ.

APL – векторные операции

Более удачной оказалась идея поэлементной обработки однородных структур данных в языке APL, особенно с появлением векторных архитектур Cray. В 1962 году был предложен интересный механизм реализации многомерных векторов, приспособленный к расширению и распараллеливанию обработки данных. Сложные данные представляются как пара из последовательности скаляров и паспорта, согласно которому эта последовательность структурируется. Такое решение позволяет любое определение функции над скалярами автоматически распространять на

произвольные структуры данных из однотипных скаляров. В настоящее время для большинства специализированных языков параллельного программирования типично, что сложные построения факторизуются с учетом особенностей структуры данных так, что выделяются несложные отображающие функции, "просачиваемые" по структуре данных с помощью функций более высокого порядка – функционалов. В результате можно независимо варьировать структуры данных, функционалы, методы сборки полного результата и набор отображаемых множеств. Целенаправленно выделяются конвейерные процессы, приспособленные к минимизации хранения промежуточных результатов. С 1980-ых годов эта идея наследуется многими языками, поддерживающими параллелизм и в наше время фактически является стандартной.

Algol-68 – критические участки и семафоры

К концу 1960-х годов сложилось значительное разнообразие теоретических моделей параллелизма, при исследовании которых проявилась проблема надежности параллельных вычислений, выразившаяся в неожиданном различии поведения императивной последовательности действий в зависимости от включаемых в нее фрагментов «независимых» процессов, нарушающих императивность. Для профилактики таких эффектов в семантику языка Algol-68 включается идея непрерывно исполняемых критических участков и представления их защиты в терминах семафоров.

Unix и JCL – управление процессами

Независимо идеи явного порождения процессов и организации их взаимодействия через каналы возникают в языках управления заданиями и процессами в операционных системах.

Setl – множества и кванторы

Другой подход к надежности программирования предложен в языке теоретико-множественного программирования Setl, ориентированном на активизацию интуиции грамотных математиков при разработке спецификаций программ в терминах преобразования множеств, естественно подразумевающих возможность параллельной обработки элементов множества, причем в реализационно независимом стиле. Наследование решений из универсальных языков сверх высокого уровня, таких как Setl, абстрагирование данных и процессов в которых приспособлено к гибкому и строгому структурированию, удобно для культивирования доказательных

построений в практике параллельного программирования. В этом плане представляет особый интерес эксперимент по развитию теоретико-множественной семантики языка Setl, в котором весьма общее построение формул с кванторами над множествами погружено в обычную схему последовательного управления процессами. Реализация языка Setl характеризуется богатым полиморфизмом. Для представления множеств используется около двадцати разных структур данных, выбор которых осуществляется системой программирования в зависимости от динамики операций над множествами. В результате программируемые функции слабо зависят от реализационной структуры данных. В практике управления процессами используется понимание команд как позиций независимого порождения процессов. Такое понимание естественно согласуется с идеями теории множеств о независимости элементов множеств и может служить основой архитектуро независимой семантики языка программирования.

SQL – транзакционная память и нормальные формы

Много более заметная проблема параллельной обработки данных независимыми операторами обнаружилась в практике применения общих баз данных, приведшей к выделению языка запросов (SQL) и концепции транзакционной памяти, теперь рассматриваемой как перспективная основа семантики языков параллельного программирования.

Occam – взаимодействие «процесс-канал»

Середина 1970-х годов характеризуется кризисом технологии программирования, выход из которого тогда виделся в массовом переходе к параллельному программированию. Активные исследования разрешимых классов параллельных схем программ показали ряд неудобных, снижающих эффективность распараллеливания, конструкций, таких как ветвления. Э. Дейкстра опубликовал решение этой проблемы в форме защищенных команд, которая нашла свое место в определении языка Occam,предоставляющем для транспьютерного программирования модель взаимодействия CSP процесс-канал. В эти же годы популяризируются идеи структурного программирования, нацеленные на снижение сложности отладки программ, близкие идеям функционального программирования, которое теперь рассматривается как один из универсальных методов представления удобно распараллеливаемых программ.

Ada – «рандеву»

В проект языка Ada предпочли включить механизм «рандеву», сводящий представление взаимодействия процессов к рассредоточенному обмену

сообщениями, подобному сигналам в оборудовании и модели CCS, что можно рассматривать как приаппаратное низкоуровневое средство, несколько диссонирующее с высоким уровнем языка.

БАРС – сетевое управление

В нашей стране разработаны языки БАРС и Поляр с разными концепциями сетевого управления процессами и представления дисциплины доступа к памяти. Программирование на уникальном по уровню средств управления процессами языке БАРС нацелено на обеспечение высокопроизводительных вычислений и организацию асинхронных параллельных процессов. При создании языка БАРС в качестве базового ЯВУ был привлечен популярный язык Pascal, в 1970-е годы перераставший из учебного в производственный язык системного программирования. При сохранении основных принципов семантики вычислений были существенно обобщены средства структуризации данных на основе понятия «мультимножество», приспособленного к именованию элементов структур данных и учета кратности их использования. Работа с именованной памятью (Name-oriented) дополнена возможностью задавать дисциплину доступа к элементам памяти. Идеи более ранних языков параллельного программирования были развиты и обогащены в языке БАРС в трех направлениях:

– в качестве базовой структуры данных были выбраны мультимножества (размеченные множества с кратностью элементов);
– описание элементов памяти сопровождается предписанием дисциплины доступа к памяти;
– средства управления асинхронными процессами включали механизм сетей Петри, координирующих работу независимо созданных функциональных фрагментов.

Процедуры в таком языке приспособлены к варьированию дисциплины доступа к данным и схемы управления процессами обработки данных. Сети Петри позволяют независимые описания процессов связывать в терминах разметки. Узлы с одинаковой разметкой срабатывают одновременно. Процесс обработки данных рассматривается как распределенная система, находящаяся под сетевым управлением. Узлы такой системы могут сработать в зависимости от условий готовности разной природы: доступность ресурсов, сигналы монитора, внутри сетевые отношения, иерархия сетей, правила функционирования разносортных подсетей. Вычисления, как и в языках APL и Sisal, распространяются со скаляров на сложные структуры. Радикальное продвижение в повышении уровня

программирования, предложенное в языке БАРС, заключается в переносе механизма типизации данных на проблему типизации схем управления.

Sisal – однократные присваивания и пространства итерирования

Авторы строго типизированного функционального языка параллельного программирования Sisal создали интересный прецедент по расширению и уточнению системы понятий программирования для нужд представления и реализации масштабируемых параллельных вычислений. Программа в этом языке строится из участков с однократными присваиваниями, удобными для оптимизирующих преобразований, включая распараллеливание. В структуре цикла выделены позиции для формирования пространства параллельных итераций, фильтрации или сборки параллельно полученных результатов и обработки потоков данных при развитой системе работы с векторами, включая методику распространения скалярных операций на структуры данных.

Произошедшее в конце 1970-х годов отвлечение внимания от кризиса технологии программирования на задачу освоения микропроцессоров не остановило поиск языковых решений для представления программ, обладающих параллелизмом. Появился функциональный язык параллельного программирования Sisal, позволяющий формировать пространства итераций для эффективного распараллеливания циклов компилятором. Название языка функционального программирования Sisal расшифровывается как «Streams and Iterations in a Single Assignment Language». Система вычислений в языке Sisal использует понятие «мультизначение», позволяющее подобно языку APL распространять скалярные действия на данные любой структуры, а их обработку осуществлять на многопроцессорных конфигурациях. Отображение мультизначения рассматривается как обработка его элементов на независимых процессорах. Результаты отображения могут повергнуться свертке или фильтрации. Sisal-программа представляет собой набор функций, допускающих частичное применение, т.е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений. Основное продвижение по технике программирования в языке Sisal – развитие структуры циклов для их реализации на параллельных процессорах. Введено понятие «пространство итераций» и предложена

специальная конструкция для фильтрации мультизначений, получаемых при совмещенном исполнении итераций и участков повторяемости. Формирование мультизначения управляется представлением пространства итераций и учетом зависимостей между одноуровневыми итерациями. Работа с именованной памятью (Name-oriented) освобождена от проблемы побочных эффектов методом локализации участков с однократными присваиваниями – SSA-форм, что делает программы удобными для преобразований, оптимизации и компиляции, включая распараллеливание и масштабирование.

Oberon – обучение управлению процессами

1980-е годы знаменует переход к сетевой обработке данных и признанию потенциала ООП при организации информационных бизнес-процессов. Появляются Oberon, Eiffel, SmallTalk-80, C++, Erlang, Perl и другие языки, отчасти компенсирующие недостаток базовых средств и методов реализации массово используемых императивных языков программирования в новых условиях. В наши дни Вирт позиционирует язык Oberon как кандидат на включение в школьную программу информатики вместо языка Паскаль.

Python и Ruby – разработка распределённых систем

Общий прогресс в эксплуатационных характеристиках оборудования с 1990-х годов резко расширил возможности сборки информационных систем из готовых компонентов и сделал доступными свободно распространяемые программные инструменты конструирования систем программирования, как правило, поддерживающие организацию параллельных процессов, если не собственно на уровне языка, то на уровне библиотечных компонент. Мультипарадигматические языки Python и Ruby показывают хорошие результаты в программировании сетевых процессов для многопроцессорных комплексов и привлекают большое число сторонников. Язык Python зарекомендовал себя как удобное средство разработки распределенных систем и сетевого программирования.

Haskell – «ленивые» вычисления и мемоизация

Кроме того, существуют сотни функциональных языков программирования, ориентированных на разные классы задач параллельного программирования. Языки функционального программирования обогатились типовыми средствами практически всех известных подходов к представлению программ и организации вычислительного эксперимента и информационных процессов. Обеспечена организация параллельных процессов. Возможна визуализация данных и программ. Имеются средства

стандартного и объектно-ориентированного программирования. Поддержано управление компиляцией и конструирование компиляторов. Методы функционального проектирования и программирования обеспечивают технику представления и отладки функциональных моделей, спецификации и верификации программ, исследования их свойств и экспериментального сравнения моделируемых параллельных процессов с моделями и прототипами. Функциональный подход исторически является основой для исследования средств и методов программирования, прототипирования и декомпозиции программируемых систем и развития современных методов параллельного и многоязыкового программирования. Разработан чисто функциональный язык Haskell, предлагающий эффективную модель «ленивых» вычислений с мемоизацией промежуточных результатов по принципу полузабытых методов «математического динамического программирования».

Норма – параллельные вычисления

Система поддержки параллельного программирования для решения задач вычислительной математики.

MPI и Open MP – средства распараллеливания программ

Появляются популярные системы MPI и Open MP, обеспечивающие эффективность параллельного программирования в рамках языков Fortran и C. В MPI взаимодействие процессов обеспечивается через посылку сообщений между процессорами. Open MP предоставляет процессам возможность использовать разные виды памяти, включая быструю общую память, что при удачном ее распределении позволяет достигать высокой эффективности.

mpC – синхронизация семейств процессоров

В мире параллелизма все базовые понятия программирования претерпели изменение или расширение (программа, ветвление, цикл, событие, память, результат). Появился ряд специфических для параллельного программирования понятий (процессор, поток, ожидание, длительность, фильтр, барьер). Программы стали многопоточными, циклы – параллельными, память обретает копии и реплики, события происходят одновременно в разных синхронизуемых процессах, вычисление результата может не означать завершение процесса. Такая ревизия понятий влияет не только на стиль программирования, но и изменяет характер компиляции на этапе генерации кода программы. Возрастает роль техники использования многократно используемых компонент схемного уровня, соответствующего

средствам типизации управления процессами. Так, например, язык mpC предлагает более детальный учет механизмов взаимодействия параллельных процессов в терминах барьеров и специальных категорий переменных, обладающих особым, аппаратно реализуемым поведением.

Java, C#, Scala, F# – библиотеки управления синхронизацией

В новых системах программирования для языков Java, C#, Scala, F# и т.д. существенно повысилась результативность системных решений в области работы с памятью, компиляции, манипулирования комплектами функций и классами объектов, выделение которых по существу обусловлено результатами теоретических работ в области системного статического анализа, основу которых все в большей мере составляет функциональный подход. Получили значительное развитие методы декомпозиции программ в рамках объектно-, субъектно- и аспектно-ориентированных подходов к определению систем программирования на базе многократно используемых компонент. Для современных областей программирования и проектирования характерна интеграция средств и методов из разных парадигм, что может привести к профессиональной консолидации программистского корпуса.

Изучаются средства организации параллельных процессов средствами языка функционального программирования F# с библиотеками .Net. На этом языке рассматривается возможность реорганизация программ, что поддержано в языке специальными средствами типа Quotation – доступ к внутреннему представлению программ в виде структуры данных. Анализируется вклад типизации данных в построение эффективных программ. Использование модифицируемой и защищенной памяти. Прикладные аспекты работы с информацией, отражающей специфику области приложения программ (measure).

C# дает возможность встраивать функциональные построения в контекст привычных императивных программ. Рассматривается стыковка программ на новых языках с производственными системами, разрабатываемыми на базе библиотек .Net. Возможность оперирования деревом разбора программы и ее исполнимым кодом. Вопросы защиты программ и данных. Управление дисциплиной доступа к памяти и представление запросов к базам данных.

CUDA – многопроцессорные видеоплаты

Появление технологии CUDA, объединившей в графических ускорителях достоинства этих ранее сложившихся подходов, выводит параллельное программирование в ранг массово доступных методов

создания программных систем благодаря преодолению стоимостного барьера. Следует ожидать, что развитие парадигмы параллельного программирования приведет к улучшению средств поддержки полного жизненного цикла программ, включая активное использование методов верификации взаимодействия процессов, автоматизацию приведения обычных программ к эффективно распараллеливаемой форме, обеспечение мобильности параллельных программ относительно параллельных архитектур, а следовательно и к повышению эффективности и надежности программ, приспособленных к многократному использованию типизированных решений особо важных наукоемких задач. На повестке дня – разработка методов архитектурно независимой кодогенерации масштабируемых параллельных программ, легко настраиваемых на особенности используемых вычислительных комплексов. Так, например, новый язык программирования OpenCL обеспечивает параллелизм на уровне инструкций и на уровне данных для различных графических и центральных процессоров.

8.5. Языки сверхвысокого уровня

Подготовка программ на базе языков сверхвысокого уровня (ЯСВУ) нацелена на длительный срок жизни запрограммированных решений особо важных и сложных задач. Удлинение жизненного цикла достигается представлением обобщенных решений с определенной степенью свободы по отношению к полным пространствам допустимых смежных компонент, реализованных ранее или планируемых на будущее. Реализационное сужение семейства процессов, допускаемых семантикой ЯСВУ, противоречит его целям или концепциям по следующим прагматическим мотивам:

– высокий уровень абстрагирования программируемых решений;
– решаются задачи, зависящие от непредсказуемых внешних факторов;
– базовые средства и/или алгоритмы вычислений используют параллелизм;
– актуальны прагматические требования к темпу и производительности вычислений;
– эксплуатируются динамически реконфигурируемые многопроцессорные комплексы.

Обычно создатели нового ЯСВУ используют в качестве исходного материала один или несколько базовых ЯВУ и встраивают в них

изобретаемые средства и методы. От базовых ЯВУ наследуются парадигмы удовлетворительного решения сопутствующих задач. В таких случаях парадигматическая характеристика ЯСВУ может формулироваться относительно базовых ЯВУ, хотя внешнее синтаксическое сходство языковых конструкций иногда скрывает совсем другую семантику.

Для ЯСВУ характерно применение регулярных, математически ясных и корректных, абстрактных структур, при обработке которых возможны преобразования данных и программ, использование подобий и доказательных построений. Все это призвано гарантировать высокую производительность вычислений, надежность процесса разработки программ и длительность их жизненного цикла. Типичны алгебраические спецификации, теоретико-множественные построения, параллелизм, модели процессов разработки программ. Изобретаются специальные системные средства, повышающие емкость представлений, их общность и масштабируемость. Естественный резерв производительности компьютеров – параллельные процессы. Их организация требует контроля и детального учета временных отношений и неимперативного стиля управления действиями. Суперкомпьютеры, поддерживающие высокопроизводительные вычисления, нуждаются в особой технике системного программирования, которая еще не сложилась, хотя уже имеется опыт эффективного решения особо важных задач.

Таким образом можно как бы «просачивать» определения функций над простыми данными, распределять их по структурам данных и тем самым распространять простые функции на сложные данные подобно матричной арифметике. Похожие построения предлагаются Бэкусом в его программной статье о функциональном стиле программирования и в языке APL, ориентированном на обработку матриц.

Как правило, языки параллельного программирования включают в себя средства, характерные для разных парадигм. Это определяет целесообразность трансформационного подхода к накоплению правильности программных решений при разработке и модернизации параллельных программ на разных языках в рамках общей системы программирования. Развитие ЯСП в настоящее время ориентировано на решение задач на основе общих библиотечных модулей, обеспечивающих эффективную организацию процессов, или подъязыков, допускающих многопоточное программирование. Это не исключает реальную практику ручного распараллеливания ранее отлаженных обычных программ, приведения их к виду, удобному для применения производственных систем поддержки параллельных вычислений. Значительная часть таких работ носит технический характер и заключается в систематической

реорганизации структур данных, изменении статуса переменных и включении в программу аннотаций, сообщающих компилятору об информационно-логических взаимосвязях. Существенным ограничением результата ручного распараллеливания является не только опасность повторной отладки алгоритма, но и его избыточная зависимость от характеристик целевой архитектуры.

8.6. Высокопроизводительное программирование.

Рассмотрим один из известных ЯП – функциональный язык параллельного программирования ***SISAL*.**

Название языка расшифровывается как «Streams and Iterations in a Single Assignment Language», сам он представляет собой дальнейшее развития языка *VAL*, известного в середине 70-х годов. Среди целей разработки языка *SISAL* следует отметить наиболее характерные, связанные с функциональным стилем программирования:

– создание универсального функционального языка;
– разработка техники оптимизации для высокоэффективных параллельных программ;
– достижение эффективности исполнения, сравнимой с императивными языками типа Fortran и C;
– внедрение функционального стиля программирования для больших научных программ.

Эти цели создателей языка *SISAL* подтверждают, что функциональные языки способствуют разработке корректных параллельных программ. Одна из причин заключается в том, что функциональные программы свободны от побочных эффектов и ошибок, зависящих от реального времени. Это существенно снижает сложность отладки. Результаты переносимы на разные архитектуры, операционные системы или инструментальное окружение. В отличие от императивных языков, функциональные языки уменьшают нагрузку на кодирование, в них проще анализировать информационные потоки и схемы управления. Легко создать функциональную программу, которая безусловно является параллельной, если ее можно писать, освободившись от большинства сложностей параллельного программирования, связанных с выражением частичных отношений порядка между отдельными операциями уровня аппаратуры. Пользователь языка *SISAL* получает возможность сконцентрироваться на конструировании алгоритмов и раз работке программ в терминах крупноблочных и регулярно организованных

построений, опираясь на естественный параллелизм уровня постановки задачи.

Параллельное программирование на языке Sisal опирается на парадигму функционального программирования. Но замысел языка нацелен на создание конкуренции вечно живому языку Fortran и, кроме того, в качестве базового языка был использован язык VAL, в свою очередь многое унаследовавший от языка Pascal. От языка Fortran унаследован ряд идей по обработке и представлению векторов.

Система вычислений в языке Sisal использует понятие «мультизначение», позволяющее подобно языку APL распространять скалярные действия на данные любой структуры, а их обработку осуществлять на многопроцессорных конфигурациях. Работа с именованной памятью (Name-oriented) освобождена от проблем с побочными эффектами с помощью локализации участков с однократными присваиваниями – SSA-форм, что делает программы удобными для преобразований, оптимизации и компиляции, включая распараллеливание и масштабирование. Отображение значений при информационной обработке рассматривается как исполняемое на многопроцессорных конфигурациях. Результаты отображения могут повергнуться свертке или фильтрации. Формирование конфигурации управляется представлением пространства итераций и учетом зависимостей между одноуровневыми итерациями. Программа строится из участков с однократными присваиваниями, что упрощает технику оптимизационных и распараллеливающих оптимизаций.

Основное продвижение по технике программирования – развитие структуры циклов для их реализации на параллельных архитектурах. Введено понятие «пространство итераций» и предложена специальная конструкция для фильтрации мультизначений, получаемых при совмещенном исполнении итераций и участков повторяемости.

Основные виды команд:

– обработка потоков (очередь =стек= список);
– контроль однократности присваиваний (SSA-формы);
– дополнение цикла участком «returns» для оформления значения распараллеленного цикла;
– формирование пространство итераций;
– канальный обмен между итерациями;
– операции по упаковке, свёртке или фильтрации серийных значений.

Фрагмент	*Пояснение*

For
Approx := 1.0;
Sign := 1.0;
Denom := 1.0;
i := 1
 	
while i <= Cycles do Sign := -Sign;
Denom := Denom + 2.0;
Approx := Approx + Sign / Denom;
i := i + 1
 	
returns Approx * 4.0
end for
	% инициирование цикла
	
	% предусловие завершения цикла
	% однократные
	% присваивания
	% образуют
	% тело цикла
	
	% выбор и вычисление результата цикла

Пример 57. Вычисление числа π (пи)

Фрагмент	*Пояснение*
for i in [1..Cycles/2] do val := 1.0/real(4*i-3) – 1.0/real(4*i-1); returns sum(val) end for * 4.0	% пространство параллельно исполнимых итераций % тело цикла, для каждого i исполняется независимо % выбор и свертка результатов всех итераций цикла % вычисление результата выражения

Пример 58. Это выражение также вычисляет число π (пи). Это выражение вычисляет сумму всех вычисленных значений `val` и умножает результат на `4.0`

Фрагмент	*Пояснение*

for i in [1..2] dot j in [3..4] do
returns product (i+j)
end for
	% для пар индексов [1,3] и [2,4]
	% произведение сумм
	% = 24

Пример 59. В for-выражениях операции `dot` и `cross` могут порождать пары индексов при формировании пространства итерирования

Фрагмент	*Пояснение*
for i in [1..2] cross j in [3..4] do	% для пар [1,3], [1,4], [2,3] и [2,4]

returns product (i+j) end for	% произведение сумм % = 600

Пример 60. В for-выражениях операции `dot` и `cross` могут порождать пары индексов при формировании пространства итерирования

Фрагмент	*Пояснение*

for
I := 1
while I < S do
K := I;
I := old I + 2;
J := K + I;
returns product(I+J)
end for
	% значение из предыдущей итерации

Пример 61. Итеративное for-выражение с обменом данными между итерациями

Как это свойственно языкам функционального программирования, *SISAL* язык математически правильный – функции отображают аргументы в результаты без побочных эффектов, и программа строится как выражение, вырабатывающее значение. Наиболее интересна форма параллельного цикла. Она выделяет три части: *for* - генератор пространства итераций, *do* - тело цикла и *returns* - формирователь возвращаемых значений.

SISAL-программа представляет собой набор функций, допускающих частичное применение, т. е. вычисление при неполном наборе аргументов. В таком случае по исходному определению функции строятся его проекции, зависящие от остальных аргументов, что позволяет оперативно использовать эффекты смешанных вычислений и определять специальные оптимизации программ, связанные с разнообразием используемых конструкций и реализационных вариантов параллельных вычислений.

Фрагмент	*Пояснение*
function Sum (N); result (+ (sqw (1 .. N)));	% Сумма квадратов

Пример 62. Сумма квадратов

Для ЯСВУ характерна яркая специфика, связанная с поиском новых средств и методов программирования. Такая специфика может стать основой новой парадигмы.

Ряд ЯВУ, такие как Fortran, Lisp, Algol, Apl, Pascal, используются как базовые при создании новых ЯВУ и ЯСВУ, что позволяет формулировать относительные парадигматические характеристики в лаконичной и легко воспринимаемой форме.

8.7. *Трансформационная семантика*

Трансформационная семантика обеспечивает сведение конструкций языка программирования к его базовым средствам, что позволяет упростить операционную семантику, а также выбрать реализационное ядро системы программирования при его экспериментальной раскрутке. Задача трансформационной семантики – сведение программы к нормализованной форме, удобной для интерпретации программ или генерации масштабируемого исполнимого кода. В случае многопоточных программ преобразование сети потоков может быть нацелено на сведение к однородной системе потоков, однозначно отображаемых на заданный комплекс процессоров – размещение потоков по процессам или назначение процессоров для выполнения потоков.

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

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

При отладке формируется ряд контекстов, на которых демонстрируются отдельные свойства фрагментов, из которых собирается полная программа. Это контексты для отдельных потоков, для пар синхронизованных потоков, для интегрированной из потоков программы, а кроме того контексты для удостоверения наличия-отсутствия информационных связей между фрагментами.

Возможны пользовательские преобразования схем управления процессами, что позволит не только минимизировать «ручную» оранжировку распараллеливаемых программ, но и даст основу для формирования библиотек преобразования схем программ. Теоретически

такие преобразования следует сопровождать доказательствами частичной эквивалентности для понимания границ их применимости.

На этом фоне задача трансформационной семантики языка параллельного программирования – сведение программы к нормализованной форме, удобной для интерпретации программ, их распараллеливания и генерации настраиваемого объектного кода. Похожая техника применяется при сведении грамматик языка к удобной для автоматизации построения анализатора форме и при оптимизации программ.

Направление преобразований программ обычно связано с определенными критериями применимости и оптимальности, учитывающими результаты анализа логических и информационных связей. При организации параллельных процессов такие критерии обладают спецификой, отражающей особенности эксплуатации многоядерных архитектур. В частности, возрастает роль учета времени доступа к разнородной памяти и статического планирования загрузки процессоров наряду с обеспечением обратимости обработки данных и динамического управления производительностью вычислений. Поддержка такой семантики вычислений выходит за границы традиционных решений по реализации языков высокого уровня.

При декомпозиции трансформационной семантики языка параллельного программирования возникает коллекция нормализованных функциональных моделей отдельных аспектов управления процессами. Нормализация моделей направлена на ограничение сложности их анализа, приспособленность к интеграции с другими моделями, поддержку быстрого прототипирования многопоточных программ, верификацию различных свойств программных систем и их факторизацию в процессе разработки и усовершенствования. Такие механизмы могут быть включены в систему параллельного программирования.

Необходимость согласования большого числа разноплановых факторов приводит к многослойному описанию семантики языков параллельного программирования (ЯПП), в котором разделены уровни абстрактных схем управления вычислениями и наполнения схем конкретными вычислениями. Чтобы избежать повторной компиляции при согласовании выбора схемы управления с целевой архитектурой, при формировании внутреннего представления многопоточных программ обычно используются методы смешанных вычислений и техника макрогенерации. Практичность таких методов обуславливается вполне конкретными, не редко диктуемыми аппаратурой, требованиями к фрагментам наполнения, связанными с целесообразностью достижения конечности отладки фрагментов

(целостность действий, однократность присваиваний, одномерные вектора, функции без рекурсии, циклы со статически определенной кратностью, потоки действий, выполняемых по готовности данных – как при вычислении арифметических выражений).

8.8. Абстрактный комплекс

Операционная семантика языка программирования обычно базируется на определении абстрактной машины, которая в случае многопроцессорных конфигураций естественно становится конструкцией из абстрактных процессоров – абстрактным комплексом (АК). В основном определение команд абстрактной машины наследует решения SECD-машины, предложенные Лэндиным и описанные в книге Хэндерсона. Для языка параллельного программирования такой комплекс можно определить следующим образом:

АМ = <Регистры, [АП-1, … АП-n], СК>
АП-i = <Пам-i, СК-i>
Регистры = < стек результатов,
структура общего контекста,
синхросеть потоков/процессоров (активных и пассивных),
очередь барьеров>

СК – система команд, включающая в себя универсальные для всех процессоров команды, типичные для языков управления заданиями.

Появляется пересмотр ряда понятий, а именно переход от АМ к абстрактному комплексу (АК):

АК = <P, R, B, D> , где
P – вектор процессоров,
R – список внешних результатов вычислений,
B – таблица синхронизованных потоков, ждущих достижения барьера другими потоками,
D – набор пассивных потоков.

Элементы P – вектора процессоров содержат номер активного потока, текущий результат его выполнения и собственно список его предстоящих действий (фрагментов):

P = <i, s, c>, где
i – уникальный номер потока, выполняемого на процессоре P;

s – стек текущих результатов i-го потока;
c – список действий i-го потока.

Определена частичная функция T(i), задающая ограничение времени выполнения потока P(i).
Общая система команд АК поддерживает выполнение следующих действий:

LOAD – загрузка произвольного пассивного потока.
F-LOAD – загрузка заданного пассивного потока.
BAR – статическая синхронизация потоков по барьерам.
IF – фильтр по заданному предикату.
WHEN – ожидание истинности предиката.
FORK – развилка.
JOIN – слияние ветвей.
SEND – динамическая синхронизация активных потоков в стиле «рандеву».
WAIT – ожидание сообщения.
MASSAGE – получение сообщения.
KILL – принудительное отключение потока с диагностическим сообщением.
STOP – плановое завершение работы потока с формированием внешнего результата.

Нормальное завершение многопоточной программы происходит при исчерпании регистров B и D и плановом завершении всех активных потоков.

Кроме того, общее завершение работы происходит при невозможности завершить работу каких-либо потоков регистра B – взаимоблокировки.

Для простоты учебной модели здесь не рассматривается учет разнообразия категорий систем команд отдельных процессоров и видов используемой памяти с различной дисциплиной функционирования. Ради удобочитаемости общие команды АК представлены в виде системы переходов:

$$[i\ s\ c, ...]\ R\ B\ D \rightarrow [i'\ s'\ c', \ldots]\ R'\ B'\ D'$$

Естественно, на уровне абстрактной машины поддержаны обычные бинарные операции над данными и дополнительные операции локального управления процессами, возникающие при реализации системы программирования. Определение синхронизации циклов и рекурсий может

быть представлено в терминах индексируемых барьеров. Динамическая синхронизация в стиле ООП несложно выражается с помощью дополнительного регистра для реализации канала обмена сообщениями.

Наполнение многопоточной программы может развиваться независимо от схем управления вычислениями в отдельных потоках, а схемы можно реорганизовывать без дополнительной отладки наполнения. Они играют роль макетов или моделей программ и работают подобно макросам (открытая подстановка), но с контролем соответствия параметров объявленным синтаксическим типам фрагментов. В новых языках программирования, поддерживающих параллелизм, таких как императивный C# и функциональный F#, имеется возможность манипулировать структурами данных, представляющими внутренний код программы.

Выделение схемного уровня упрощает включение в схему разработки программ механизмов верификации (подобие модели или соответствие аксиомам), а на их основе возможна проверка программ на правдоподобие, логический вывод свойств, выполнение индуктивных и дедуктивных построений. Кроме того, техника отладки программ обогащается возможностью привлечения протоколов ранее выполненных вычислений и приведения программ к нормальным формам, удобным для сведения к базовым/стандартным моделям параллельных вычислений.

8.9. Память

На уровне первичного элементарного программирования мало заметна принципиальная разница между хранением данных в оперативной и внешней памяти. Особенности работы с внешней памятью много детальнее изучены в практике организации реляционных баз данных (БД) и применения языка запросов к базам данных SQL.

Характерной особенностью хранимых в БД данных является их соответствие некоторым физическим параметрам реальных объектов. Такие параметры относительно одного объекта можно представлять в виде строки таблицы. Существуют системы управления базами данных (СУБД), обеспечивающие создание, долговременной хранение, использование и уточнение таких таблиц.

В использовании конкретных таблиц может быть заинтересовано множество пользователей для разных сфер применения. Хранение таблиц может быть организовано на базе различных устройств и файловых систем и доступ к ним может реализовываться с помощью различных дисциплин коммуникации. Для обеспечения эффективности использования

аппаратуры и необходимого для практичности пользовательского интерфейса разработаны определённые системные решения, что суммарно приводит к трём уровням проектирования архитектуры БД.

Уровни архитектуры:

- внутренний (организация хранения);
- внешний (логика использования);
- промежуточный (концепции системы).

Важным достижением реализационного похода в этом направлении является понятие нормальных форм (НФ), дающее разработчику БД чёткое руководство по прогнозу эффективности принятых им решений. Число таких форм немного превышает десяток. Соответствие нормальной форме определяет выбор схемы реализации БД, компактность и структуру хранения данных во внешней памяти и скорость доступа к хранимым данным. Причём во многих случаях повышение компактности хранения сопровождается повышением скорости доступа, опровергая расхожее утверждение, что выигрыш в памяти влечёт потери в скорости и наоборот. Нормальные формы образуют упорядоченную последовательность – очередная последовательность наследует свойства предыдущей.

Полноценное решение проблем параллельного программирования требует создания более специализированного инструментария, некоторые механизмы реализации которого могут быть изучены в форме экспериментальной разработки учебного языка параллельного программирования.

ЗАКЛЮЧЕНИЕ

Выбор парадигмы программирования – это выбор концептуальной схемы постановки проблем и методов их решения, инструмент «грамотного» описания фактов, событий, явлений и процессов, выделения частных и общих понятий. Альтернативные подходы к обработке информации, накопленные и сложившиеся в форме языков и систем программирования, принято называть парадигмами программирования. Изучение и чёткая классификация уже сложившихся и новых компьютерных парадигм призваны помочь обоснованному выбору компьютерных языков при формировании новых программных проектов и создании новых информационных технологий.

Естественная классификация по ёмкости абстрагирования конструкций, выделяющая ЯП низкого, высокого и сверхвысокого уровня, дополняется определением парадигм, поддерживаемых ЯСП. В настоящее время число по существу различимо более двух десятков парадигм. Многие языки программирования относят к пяти - восьми парадигмам. Часть упоминаемых в разных источниках ПП можно характеризовать как стили или методики, отражающие поиск путей снижения трудоёмкости программирования и повышения надёжности программ на базе доступных СП. Например, аспектно-ориентированное программирование поддерживается как макро-расширение ООП. Структурное программирование фактически сводится к ряду рекомендаций по стилю представления императивно-процедурных программ. Мета-программирование представляет собой технику компиляции программ в комплекте с типовыми элементами данных. Недетерминированное программирование не более чем частный случай параллелизма.

При определении ПП обычно поляризуются следующие характеристики ЯП:

– программируемые решения представляются в императивной или в декларативной форме;
– обрабатываемые элементы данных позиционируются как адресуемые блоки памяти или независимо размещаемые значения;
– программа может быть защищена от изменений в процессе её выполнения или допускать программируемые модификации по ходу получения результатов вычисления;

– программа может быть целостной или собираться из типовых компонент и шаблонов;
– представленные в программе функции могут быть частичными, типизированными, обрабатывающими значения заданного домена или универсальными, дающими разумную реакцию на любой элемент данных;
– управления вычислениями выполняется последовательно или параллельно;
– порядок действий может быть определённым или недетерминированным;
– вычисления могут быть энергичными или ленивыми;
– области видимости имён могут быть глобальными или локализованными по иерархии конструкций с возможностью восстановления контекста;
– распределение и повторное использование памяти может быть действием в программе или выполняться автоматически СП;
– инициирование памяти первоначально размещаемыми значениями может требовать программируемых действий или выполняться в СП по умолчанию;
– домены значений могут быть независимыми или допускать пересечения;
– результат выполнения программы может быть рассредоточен по ряду переменных или сконцентрирован в специальном регистре;
– контроль правильности может выполняться статически – при анализе текста программы или динамически – при выполнении кода программы.

Выбор между так противопоставленными характеристиками в представлении программы выражается синтаксически или с помощью спецификаторов. В практике программирования признаны основными парадигмы императивно-процедурного, функционального, логического и объектно-ориентированного программирований, поддерживающие механизмы снижения трудоёмкости полного жизненного цикла программ с тенденцией продвижений к параллельному программированию. При оценке образовательного значения ПП выделяют в качестве базовых функциональное, параллельное и императивно-процедурное программирования, а логическое и ООП рассматривают как дополнение, изучаемое в виде расширения базовых парадигм. Мультипарадигматичность долго живущих ЯП и тенденция создания новых мультипарадигматических ЯП

говорит о созревании единой ПП, объединяющей выверенные в практике механизмы результативного программирования.

СПИСОК ЛИТЕРАТУРЫ

1. *Айлиф Дж.* Принципы построения базовой машины. / Дж . Айлиф. – М.: Мир, 1973. – 119 с.
2. *Баранов С. Н.* Феномен Форта / С. Н. Баранов, М. Ю. Колодин. // Системная информатика. Вып 4. Методы теоретического и системного программирования. – Новосибирск: Наука. Сиб. изд. фирма, 1995. С. 193–271.
3. *Браун П.* Макропроцессоры и мобильность программного обеспечения. /П. Браун. – М.: Мир, 1977. 253 с.
4. *Вирт Н.* От Модулы к Оберону. / Н. Вирт. // Системная информатика. Вып 1. Проблемы современнного программирования. – Новосибирск: Наука. Сиб. отд-е, 1991. С. 63–75
5. *Воеводин В. В.* Параллельные вычисления. / В. В. Воеводин, Вл. В. Воеводин. – СПб.: БХВ-Петербург, 2002. 608 с.
6. *Городняя Л. В.* Функциональный подход к описанию парадигм программирования / Л. В. Городняя // Препринт № 152. Надзаг.: ИСИ СО РАН, – Новосибирск, 2009. С. 66.
7. *Дейкстра Э.* Дисциплина программирования. / Э. Дейкстра. М.: Мир, 1978. 275 с.
8. *Иртегов Д. В.* Введение в операционные системы / Д. В. Иртегов. – СПб.: БХВ-Петербург, 2008. 1040 с.
9. *Лавров С. С.* Методы задания семантики языков программирования / С. С. Лавров // Программирование, 1978. N 6. С. 3-10.
10. *Лавров С. С.* Функциональное программирование. Интерпретатор языка Лисп / С. С. Лавров, Л. В. Городняя // Компьютерные инструменты в образовании. – СПб, 2002. №5.
11. *Леман М. М.* Программы, жизненные циклы и законы эволюции программного обеспечения / М. М. Леман. – М.: Мир, ТИИЭР, 1980. Т. 68-9. С. 26-45
12. *Мейер Б.* Основы объектно-ориентированного проектирования. / Б. Мейер. – М.: Интернет-Университет Информационных технологий. URL: http://www.intuit.ru/department/se/oopbases/, 2007 *(проверено 1.12.2014)*
13. *Пратт Т.* Языки программирования. Разработка и реализация. / Т. Пратт, М. Зелковиц; Под общей редакцией А.Матросова. – СПб.: Питер, 2002. 688 с.
14. *Сошников Д. В.* Функциональное программирование на языке F#. / Д. В. Сошников. – М.: ДМК Пресс, 2011.
15. *Хендерсон П.* Функциональное программирование. / П. Хендерсон. – М.: Мир, 1983. 349 с.

16. *Хоар Ч.* Взаимодействующие последовательные процессы. / Ч. Хоар. – М.: Мир, 1989. 264 с.
17. *Хорстман К.* Scala для нетерпеливых. / К. Хорстман. – М.: ДМК Пресс, 2013. 408 с.
18. *Черноножкин С. К.* Методы тестирования программ / С. К. Черноножкин. – Новосибирск: НГУ, 2004. 166 с.
19. *Backus J.* Can programming be liberated from the von Neumann style? A functional stile and its algebra of programs / J. Backus. // Comm. ACM. 1978. V. 21 N. 8. P. 613-641.
20. *Floyd, R. W.* The paradigms of Programming. / R. W. Floyd. //Communications of the ACM 22 (N 8), 1979: p. 455
21. *Knoop J.* Compiler Construction / 20th International Conference, CC 2011Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011 Saarbrücken, Germany, March 26 – April 3, 2011 // Lecture Notes in Computer Science. Springer V2011. V. 6601. 330 p.
22. *McCarthy J.* LISP 1.5 Programming Manual / J. McCarthy. The MIT Press, Cambridge, 1963. 106 p
23. *Russian Lisp User Group.* URL: http://lisp.ru/
24. *Примеры на языке Prolog.* URL: http://www.cs.bham.ac.uk/~pjh/modules/current/25433/examples/l15_example3.html

Приложение

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.

Показатель отлаженности программы – частота обнаружения дефектов в программе или длина интервала между внесением исправлений в программу.

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.

Семантический спуск определения ЯП – исключение из определения ЯП компонентов, сводимых у более простым или общим средствам, что формально не влияет на полноту ЯП.

Список свободной памяти – структура данных, обеспечивающая динамический доступ к памяти.

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*
АК	Абстрактный комплекс
АМ	Абстрактная машина
АС	Абстрактный синтаксис
БД	Базы данных
БНФ	Формы Бэкуса-Наура
БС	Базовые средства
ВС	Вспомогательная семантика
ЖЦП	Жизненный цикл программ
ЗШ	Заочная школа юных программистов
ЗШЮП	Заочная школа юных программистов
ИП	Императивно-процедурное программирование
ИС	Информационная система
ИТ	Информационные технологии
КМ	Конкретная машина
КП	Компонентное программирование
КС	Конкретный синтаксис
КЯ	Концептуальные языки программирования
ЛП	Логическое программирование
ЛШ	Летняя школа юных программистов
ЛШЮП	Летняя школа юных программистов

НФ	Нормализованная форма
ОИВТ	Основы информатики и вычислительной техники
ООП	Объектно-ориентированное программирование
ОС	Операционная семантика
ПЖЦП	Полный жизненный цикл программ
ПИП	Процедурно-императивное программирование
ПП	Парадигма программирования
РБНФ	Расширенные формы Бэкуса-Наура
РИС	Распределённая информационная система
РП	Реализационная прагматика
СД	Структуры данных
СП	Система программирования
СПП	Стандартное прикладное программирование
СУ	Синтаксическое управление
ТА	Таблица атомов
ТД	Типы данных
ТИ	Таблица идентификаторов/имён
ТП	Технология программирования
УФ	Универсальная функция языка программирования
УЯ	Учебный язык программирования
ФП	Функциональное программирование
ШИ	Школьная информатика
ШЮП	Школа юных программистов
ЯНОП	Язык начального обучения программированию
ЭП	Эксплуатационная прагматика
ЯВУ	Язык высокого уровня
ЯНОП	Язык начального обучения программированию
ЯНУ	Язык низкого уровня
ЯП	Язык программирования
ЯПП	Язык параллельного программирования
ЯСВУ	Язык сверх высокого уровня
ЯСП	Язык и система программирования
ЯФП	Язык функционального программирования
AMA	Абстрактная машина.ассемблера
AMF	Абстрактная машина языка Forth
AML	Абстрактная машина языка Lisp
AMM	Абстрактная машина макрогенератора
AMP	Абстрактная машина языка Pascal
AMQ	Абстрактная машина управления процессами
BIOS	basic input/output system - «базовая система ввода-вывода»
CSP	Communicating Sequential Processes — теория взаимодействующих последовательных процессов,

	разработанная Чарльзом Э. Хоаром в 1969 году.
CCS	Calculus of Communicating Systems — исчисление общающихся систем, разработанное Робином Милнером в 1980 году
FDD	Функционально-ориентированное проектирование
LAP	Ассемблер языка Lisp
PID	Идентификатор процесса
RISC	restricted (reduced) instruction set computer — компьютер с сокращённым набором команд
SECD	Абстрактная машина языка Lisp
SECM	Абстрактная машина языка Pascal
SSA	Однократное присваивание
UML	Универсальный язык моделирования UML
XP	Экстремальное программирование

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`.
`([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).
(Expr) – результат вычисления выражения или успех выполнения процесса.
$ – переменная для кода успеха/результата процесса.
$* – все аргументы переданные скрипту (выводятся в строку).
$! – PID последнего запущенного в фоне процесса.
$$ – PID самого скрипта.
NN(d) – список номеров и имён элементов очереди.
[<Num, Name, Text>, …] - очереди процессов.
NULL – пустой файл.
H(d) – голова очереди, точнее – процесс с наивысшим приоритетом.
T(d) – хвост очереди, остаток после удаления головы. d = H(d) • T(d).
PN – имя текущего процесса.

Учебное издание

Городняя Лидия Васильевна

ПАРАДИГМА ПРОГРАММИРОВАНИЯ

Курс лекций

Редактор Т.Ю. Седыченко

Подписано в печать __.__.2015 г.
Формат 60×84 1/16. Уч.-изд. л. 12. Усл. печ. л. ___.
Тираж ___ экз. Заказ №

Редакционно-издательский центр НГУ.
630090, Новосибирск, ул. Пирогова, 2.