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

Л.В. Городняя

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 2
Языки низкого уровня

Препринт
173

Новосибирск 2015

Препринт является второй частью серии «Парадигмы программирования», посвященной исследованию основных парадигм программирования. Представлены результаты анализа особенностей языков низкого уровня, рассматриваемых как предшественники языков высокого уровня. Для иллюстрации использованы фрагменты ряда языков машинно-ориентированного императивного программирования, макротехника и языки управления заданиями. Содержание препринта представляет интерес для системных программистов, студентов и аспирантов, специализирующихся в области системного и теоретического программирования, и для всех тех, кто интересуется проблемами современной информатики, программирования и информационных технологий.

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

L.V. Gorodnyaya

PROGRAMING PARADIGMS

Part 2
Low Level Programming Languages

Preprint
173

Novosibirsk 2015

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

ВВЕДЕНИЕ

Стремительный рост числа языков программирования (ЯП) приводит к проблеме их лаконичной спецификации, освобождающей от необходимости в слишком большом объёме документации при выборе практичных средств разработки программ. Основой такой спецификации может служить парадигматическая классификация языков программирования, показывающая свод семантических методов, поддерживаемых системой программирования (СП).

Парадигмы программирования (ПП) различаются нишей в жизненном цикле программ, приоритетами при оценке качества программ, выбором инструментов и методов обработки данных [6,7]. Часть таких различий может быть выражена в определении операционной семантики (ОС) ЯП. Другие исторически сложившиеся различия отражают реализационную прагматику (РП) достаточно известных СП, наследуемую в новых проектах. Так, например, разработчики нового языка Scala наследуют байт-код языка Java [20,21].

В препринте [6] описана методика определения парадигматической характеристики ЯП. В данном препринте эта методика иллюстрируется на материале языков низкого уровня (ЯНУ), дающих примеры парадигматической характеристики языков низкого уровня, дающих удобные примеры моно-парадигматических систем обработки данных, вошедших в основные парадигмы языков высокого уровня (ЯВУ).

Программирование или кодирование на ЯНУ ассоциируется с одноуровневыми структурами данных, обусловленными архитектурой и оборудованием[1]. 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

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

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 [3, 5, 10, 12, 13, 15]. 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 [1]. 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, 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.

4. 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 возможно-

стям 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. Возникло автоматное программирование до появления ЯВУ [8].

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 [4, 16].

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.

ассемблер = (Текст → {Код | Адрес}): Пам [ячейка] → 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 ... Kn] → [W1 ... 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.

Таблица 1

Конкретизация понятий в языках ассемблера

Понятие	*Ассемблер*
Адрес	Указатель
Регистр	Специально адресуемое слово
Сумматор	Регистр для вычислений
Состояние процесса	Регистр для управления
Машинное слово	Ячейка
Память	Адрес → слово
Структура	блок, стек, ссылки, косвенность, индексы
Переменная	Ид → Адрес (Знач) Регистр
Значение	|Адрес| – в определённых границах |Число| – в пределах машинного слова

	|4 байта| – код строки, размещаемый в ячейке |Код команды|
Выражение	Адрес Метка Одна операция, над адресами (Индексная арифметика)
Команда	Операция Управление цикл-ветвление
Операция	Арифметические операции Индексная арифметика пересылки
Условие/истина	= 0/~=0 Состояние процесса Шкала прерываний
Управление	Значение регистра «счётчик команд»
Прерывание	Шкала прерываний с таблицей обработчиков
Подпрограмма	Адрес Подпрограммы с ПУ в конце для возврата
Вызов подпрограммы	ПУ с точкой возврата
Определение подпрограммы	Ид → Адрес (Подпрограммы)
Обработчик	Подпрограмма для обработки прерываний
Идентификатор	Метка = Адрес

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, основные из которых:

- неявная – команда сама «знает», где и что она обрабатывает, где берет данные и куда разместит результат (сумматоры, регистры);
- непосредственная – операнд расположен непосредственно в команде;
- прямые адреса – код адреса размещен в поле операнда;
- индексируемые адреса – один из операндов используется как индекс при вычислении адреса других операндов;
- базируемых (по регистру) – указан базовый регистр для пересчета адресов операндов;
- относительные (по текущей позиции) – адресация учитывает адрес размещения команды;
- косвенные (через промежуточное слово) – операнд указывает на слово, хранящее адрес значения;
- модифицируемые (по значению-регистру) – один операнд указывает на слово, хранящее значение, модифицирующее адрес другого операнда;
- стек – операнд, доступ к которому подчинен дисциплине стека – «первый пришел – последний ушел».

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 [11].

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).

Код может быть сформирован в расчёте на использование специальной программы «загрузчик», обеспечивающей применение программы как модуля совместно с независимо подготовленными объектами.

Обычно выделена точка входа в программу из операционной системы. START – типичная метка входа в программу из операционной системы, во многих языках ассемблера символизирующая начальный адрес исполнения кода программы.

Абстрактный синтаксис (АС) языка ассемблера сводится к представлению фрагментов с фиксированным числом реализационно простых позиций:

(МЕТКА Идент)
(КОМ ИР [Адр | Метка])
(КОНСТ [Число | Байты]) = команда «КОНСТ»

Абстрактная машина языка ассемблера, как и любого ЯП, определяется парой из основных регистров конкретной машины и её системы команд.

AMA= <RA, SCA>

где RA – набор основных регистров АМ ассемблера, SCA – базовая система команд, т.е. фиксированный словарь, не изменяющийся при выполнении программы.

По тексту программы при ассемблировании формируется код программы, выполняющий пошаговое преобразование памяти. Шаги преобразования затрагивают небольшое число регистров, обычно от одного до трёх, хотя встречаются исключения. Например, пересылка может затрагивать указанный диапазон соседних ячеек.

Асс = (текст → код) : Пам → Пам'

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 SCA может быть задана над тройкой RA = <S, C, M >, в которой S и C – простые регистры, а M – вектор, представляющий общую память.

<сумматор: значение, указатель_на_текущую_команду: Адр[Ком] , память: Адр-Знач >
S C M → S′ C′ M′

Пример системы команд:

Таблица 2

Пример системы команд ассемблера.
Пересылки – это изменение части состояния общей памяти.
Следующая команда расположена по соседству или задана командой передачи управления

SCA	*Описание команды*	*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	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 3

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	
(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. 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.

Зависимость ассемблера от конкретной машины или архитектуры создает значительные трудности при переносе программ на новые компьютерные системы. Существуют методы преодоления такой зависимости, подходы к повышению эффективности программ в тех случаях, когда ценой независимости является чрезмерно избыточный расход памяти или времени, и средства достижения высокой производительности программирования. [16]

Встраиваемые в СП ассемблеры при кодировании подпрограмм должны учитывать стандартные правила доступа к параметрам процедур и размещения выработанного ими результата. Таким же правилам подчиняется и компилируемый код программы, использующей ассемблерные вставки. Это позволяет формально считать равноправными встроенные операции, ассемблерные подпрограммы и функции, программируемые на ЯП. Особенности процесса компиляции достаточно сложны даже для простых ЯП. Такой подход полезен для решения ряда технологических проблем разработки эффективных программных систем (мобильность, надежность, независимость от архитектур и т.п.)

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) [13]

Примеры применения ассемблера LAP в системе программирования для языка Lisp 1.5

В первых Lisp-системах для реализации ядра и встроенных операций использовался специальный Lisp-ассемблер LAP, описание которого можно рассмотреть в качестве иллюстрации взаимодействия ассемблера с СП [22]. LAP проектировался специально для нужд Lisp-компилятора, но он применялся и для низкоуровневых определений функций, а также для обработки исправлений (patches). Это двухпроходный встроенный, исключительно внутренний ассемблер. Первый просмотр анализирует программу и выясняет взаимосвязи между ее частями и Lisp-системой. При ассемблировании на LAP не предусмотрено и не происходит никаких манипуляций с текстами программы в файлах. Ассемблирование кода программы выполняется в оперативной памяти во время второго просмотра, выполняющего сборку кода с установлением фактических адресов. LAP был включен в Lisp-систему как псевдо-функция от двух аргументов. Первый аргумент – листинг программы, представленный в виде списка, второй – исходная таблица символов. Результат – окончательная таблица символов, по формату напоминающая таблицу атомов.

(LAP листинг таблица)

Началом листинга ассемблерной программы является строка, сообщающая ассемблеру, с какой позиции стартовать, и предстоит ли полученный в результате код программы встраивать в интерпретатор как Lisp-функцию.

Предикат GREATER наводит некоторый канонический порядок среди атомов.

Фрагмент программы	*Примечание*

`(LAP ( (GREATER SUBR 2)`
`(TLQ (* 3))`
`(PXA 0 0)`
`(TRA 1 4)`
`(CLA (QUOTE *T* ) )`
`(TRA 1 4) )`
`NIL )`
`)`
	Имя «GREATER» дано подпрограмме от двух параметров

Пример 2. Lisp-функция [22]

По завершении работы ассемблера индикатор «Тип» будет размещен в списке свойств атома «Название» вместе со специально сконструированной структурой данных, хранящей адрес построенного кода, парность и команду для вызова этого кода как подпрограммы из Лисп-интерпретатора. Тип – это обычно SUBR или FSUBR, отражает разницу в методах обработки параметров обычными и специальными макрофункциями – EXPR и FEXPR, соответственно. При ассемблировании атомарных форм таблица атома проверяется на вхождение индикаторов SUBR, FSUBR.

В качестве модели ассемблера можно использовать определение абстрактной машины SECM-машины (без регистра E), приведённое в препринте «Парадигмы программирования. Часть 1» [6]. Результатом работы программы формально считается состояние памяти.

Ассемблерный стиль программирования для целей обучения программированию может быть воспроизведен на языке высокого уровня.

Фрагмент программы	*Примечание*
program MAX2;	Объявлено имя программы.
var A,B,S : int;	Описаны типы переменных для хранения данных и промежуточных результатов
begin	Начало блока
A := 2;	Заданы значения сравниваемых чисел
B := 3;	
S := A – B;	Разность чисел
if (S < 0) then goto MA;	Передача управления в случае отрицательной разности
goto MB;	Передача управления
MA: S := A;	Меткой «MA» помечен результат – число «A»

goto ME; MB: S := B; goto ME; ME: (writeln S) end.	Передача управления на завершение Меткой «MB» помечен результат – число «B» Передача управления на завершение Меткой «ME» помечено завершение программы с выдачей результата

Пример 3. Pascal. Выбор большего из двух чисел

Фрагмент программы	*Примечание*
(Prog (A B S) (setq A 2) (setq B 3) (setq S (- A B)) (cond ((< 0 S) (go MA))) (go MB) MA (setq S A) (go ME) MB (setq S B) (go ME) ME (return (print S)))	Вычисляется императивная форма с тремя рабочими переменными Заданы значения сравниваемых чисел Разность чисел Передача управления в случае отрицательной разности Передача управления Метка «MA» Результат – число «A» Передача управления на завершение Метка «MB» Результат – число «B» Передача управления на завершение Метка «ME» Выдача результата и завершение формы «PROG»

Пример 4. Lisp. Выбор большего из двух чисел. (Форма Prog)

Низкоуровневое программирование успешно при детальном знакомстве с архитектурой компьютера, что в конечном счете приводит к ее пониманию. Принципы и способы программирования в общем не зависят от языка. Главным требованием является умение мыслить логически.

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. 4

Парадигматическая характеристика ассемблера

Параметр	*Конкретика*
1. Эксплуатационная прагматика ЯП	Ассемблер – средство эффективного программирования при решении задач доступа к полному спектру возможностей оборудования
2. Особенности системы понятий	Комплекс непосредственно представимых и неявно подразумеваемых понятий, индуцированных архитектурой, сводится к фиксированному набору команд над регистрами определённого размера
3. Перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Метка, команда, адрес, константа
4. Базовые средства	Команды, наиболее часто используемые в шаблонах компиляции программ на ЯВУ
5. Семантические расширения	Варианты адресация и передач управления, разновидности арифметических операций в зависимости от размера и типа хранимых значений, прерывания, порты, буфера обмена данными
6. 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
7. 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
8. 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.
9. Парадигматическая специфика	Классический язык императивного машинно-ориентированного программирования.

5. 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 Форте сопровождается систематической сверткой понятий, синтаксис применения которых созвучен польской записи. Можно сказать, что хорошая программа на Форте – 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.

Таблица 5

Конкретизация понятий в машинно-ориентированном языке Forth

Понятие	*Forth*
Атом (Элементарное)	Слово
Структура	Стек последовательность
Переменная	позиция в стеке Имя → Ид (Знач) в словаре
Значение	|код|
Выражение	Постфиксные
Действие/Операция	над стеком ариф
Условие/истина	0 на вершине стека
Функция/подпрограмма	Имя → Ид последовательность слов в словаре
Аргумент	позиция в стеке
Вызов Фн/подпр	результат в стеке
Определение Фн/подпр	Имя → строка занесение в словарь
Идентификатор	номер в словаре

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. В качестве элементарных данных выступают литеры, целые без знака и со знаком, неотрицательные целые, адреса, двойные слова и др. типы данных.

Имеются константы TRUE и FALSE, символизирующие логические значения.

Новые слова можно вводить в форме

: имя опр;

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

(КОМ Арг1 Арг2 … АргК) | Текст

Абстрактная машина AMS = <SCS, RS>, где RS = <S, E, C>

Спецификация команд абстрактной машины для языка Forth представляется тройкой

<Стек, словарь, указатель_на_текущее_слово_в_программе>
S E C → S’ E’ C’

где 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 6

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 7

Спецификация команд стековой машины языка Forth. N – длина стека

RS	RS	*Примечание*
s e (LDC X . c)	(X . s) e c	Подготовка стека к предстоящим операциям.
(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 Форт содержит пару «интерпретатор – компилятор», причем техника компиляции весьма эффективна. Система использует единый порядок представления данных и команд в программе – все 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 Форт по сложности сравним с автоматом для ассемблера. Основные различия таковы:

- словарь Форта 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.

Работа со стеком достаточно просто моделируется на списках интерпретатором, подобно упрощённому SECD.

На практике язык используется с расширенной системой команд. Кроме обычных арифметических операций, имеются эффективно реализуемые:

1+ 1- 2+ 2- 2/ половина

Слово стека занимает 16 разрядов, но имеются операции над двойными словами – в 32 разряда:

2drop 2dup 2over 2rot 2swap

Есть вариант умножения чисел без потери точности – результат пишется в двойное слово:

UM* (a,b → cc)

Система программирования использует при работе ряд стеков:
R – стек возвратов
C – стек компиляции
F – стек чисел
S – стек словарей

Имеются операции, работающие с адресами и памятью:
@ (adr → x) чтение-разыменование
! (x adr →) запись

ALLOCATE выделить память
FREE освободить память
RESIZE заменить размер
IOR код завершения операции

ALLOT (N → адр) резервирование памяти в словах
HERE (→ A) адрес слова
, компиляция со стека
UNUSED (→ N) остаток памяти

Операции заполнения памяти предусматривают следующие вариации кодов росписи:

FILL char
ERASE 0
BLANC пробел

Возможна пересылка блоков памяти:

CMOVE
CMOVE>
MOVE

Арифметические операции:
+ – * / (n1 n2 → n3)
*/ (n1 n2 n3 → n4) n1 * n2/n3 «рабочая лошадка», эффективно выполняющая часто встречающееся сочетание операций.

Слова система хранит в словарях, образующих контекст исполнения программы. Непрерывная среда разработки программ содержит средства отладки, средства управления разработкой и методами обработки текста программы, включая гибкое сочетание интерпретации и компиляции.

Состояние системы программирования можно исследовать на уровне системных переменных. Различается состояние компиляции (создание кода программы) и интерпретации (непосредственное исполнение программы). Если STATE = 0, то система ведет исполнение программы.

Системная переменная BASE задает систему счисления.

Средства размещения слов в словарь:

IMMIDIATE (→) немедленное исполнение в любом состоянии

FORGET убрать из словаря

Обозначение систем счисления:

DECIMAL
HEX
LITERAL

Богато представлена логика и операции сравнения:

and or xor not
< = > <> 0<> 0< 0= 0>

Манипулирование стеками:

n STACK s – новый стек S размера N

ws PUSH – запись в стек
s POP – перенос в стек данных
s ST @ – скопировать
s ST ! – сброс стека

Возможна работа на уровне обычного ассемблера:

CODE ... ENDCODE

KEY – ввод с клавиатуры
KEY? – есть ли символ?
ACCERT [=EXPECT] – ввод строки
PARSE – адр слова в буфере и его длина

При работе со словарем, рассматриваемым как список слов, можно применять отношение порядка:

ORDER – в порядке поиска

Имеются средства работы с событиями, исключениями (0 – успех) и прерываниями:

CATCH - THROW

Можно создавать мета-определения:

CREATE DOED>

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

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

Пример 5. 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) DUP (A B → A B B) SWAP (A B B → A B**2 → B**2 A) DUP * (B**2 A → B**2 A A → B**2 A**2) + (B**2 A**2 → B**2 + A**2) ; (конец определения нового слова)	Объявление имени укрупнённого действия Дубль верхнего элемента Перемножение и перестановка Дубль и квадрат Сумма – результат в стеке

Пример 6. Введение нового слова SQ для подсчета суммы квадратов
(A B → A**2 + B**2)

Для удобства программирования определений функций привлекается моделирование переменных и констант:

```
VARIABLE  имя                    – переменная  в словарь
знач имя !                       – присвоить = запись в переменную
имя @                (a → x)     – чтение в стек из переменной
знач CONSTANT имя    (X →)       – создание константы по стеку
имя                  (→ x)           – константа → стек
знач VALUE имя       (X →)           – переименование
```

Средства управления процессами ветвления: условное выражение и переключатель.

```
усл IF часть-то THEN
усл IF часть-то  ELSE часть-иначе THEN

зн-0   CASE
зн-1  OF ветвь1 ENDOF
---
---

[ветвь-иначе] ENDCASE
```

Моделирование циклов:

```
кон-зн нач-зн DO тело-цикла LOOP
кон-зн нач-зн DO тело-цикла шаг +LOOP
          ?DO - м.б. ни разу

I - текущее значение счетчика
J - внешний цикл

UNLOOP сброс счетчика
LEAVE выход из цикла

... BEGIN ... AGAIN бесконечный цикл
       ... усл WHILE ... REPEAT ...

... BEGIN ... усл UNTIL
```

Язык Forth – пример организации вычислений над стеком, что 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 [1].

Появление в 70-х годах микропроцессорных средств с малым объемом памяти и сравнительно невысоким быстродействием вызвало очевидные трудности в применении привычных языков высокого уровня. Не удивительно, что поиск подходящих средств программирования сменил приоритеты в оценке свойств языков и систем программирования. На передний план вышли машинно-ориентированные языки-оболочки, поддерживающие достижение эффективности и компактности программ при достаточном уровне абстрагирования от конкретики процессоров, сравнимом с абстрагированием в языках высокого уровня. Особо ярким явлением в эти годы стал язык Forth, сохранивший многочисленных приверженцев и в наши дни, когда пресс технических характеристик изрядно помягчал [1].

Наиболее убедительное применение Forth получил как средство разработки специализированных систем, создаваемых по методике раскрутки, начиная с тщательно минимизированного ядра с рядом последовательных шагов расширения в рамках единой оболочки. Сам язык устроен по такому же принципу, так что такая технология пошагового программирования впитывается при изучении идей и средств системы программирования на базе Forth-а. Кроме общеизвестных примеров про системы управления астрономическими телескопами и лазерами, следует упомянуть IBMCAD – аналог системы инженерного AutoCAD и систему автоматизации издательского дела МРАМОР. Для последней именно на Форте была создана предельно компактная операционная система объемом около 4 килобайт. Форт был успешно применен при реализации языка PostScript, до сих пор используется при разработке видеоигр и систем начальной загрузки ОС, чувствительных к скорости срабатывания. При значительном внешнем несходстве Форт обладает концептуальным родством с языком Лисп, что побудило в 80-е годы к разработке языка ФоЛи, объединяющего достоинства обоих языков. [9]

Программирование на Форте 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 Форте. Ус-

пех достигается явным максимализмом в тщательной отладке и способностью видеть задачу программирования в развитии.

Язык Форт предложен Чарльзом Маури в 1968 году. К середине 70-х Форт стал третьим по популярности после Бейсика и Паскаля, завоевав свои позиции при освоении микропроцессорных средств. По технике программирования Форт весьма похож на макроассемблер, только вместо системы команд над машинными словами в нем используется система операций на стеком.

Все это позволяет утверждать, что Форт – это и технология, и философия, и язык программирования, обеспечивающий управляемый компромисс между сложностью разработки программ и удобным программированием в терминах задачи.

Хорошо написанные на Форте программы обладают гибкостью, универсальностью, компактностью, расширяемостью и простотой. Удобочитаемость программ не входит в число достоинств этого языка, но при определенных лингвистических навыках потенциально достижима. Возможна выработка стиля программирования на Форте типа мимикрии под другие языки.

Массовое распространение Форта по времени совпало с интенсивным обновлением микропроцессорных средств, расширением их возможностей и эксплуатационных характеристик. Это повлияло на приспособленность языка к обеспечению совместимости и переносимости программ на разные версии Форта своими средствами, а также существенную открытость для программиста.

Традиционно отмечаемые недостатки Форта:

- сложность записи программ для новичков;
- необходимость понимания всех процессов исполнения программы;
- трудность отчуждения программы от системы программирования;
- нестандартность языка: это не типовой язык программирования;
- строго последовательное исполнение потока операций;
- слабый контроль ограничений на оперативную память;
- неполный цикл разработки не приспособлен к производству.

Активный популяризатор Форта Мур отметил: «Форт не уравнитель, а усилитель!».

Современные реализации Форта предоставляют средства работы с очередями, управления компиляцией, локализации переменных, обработки структур данных. Системы программирования на Форте содержат препроцессор CREATE, средства работы со статическими динамическими метка-

ми, объявления отношений, связывания имен, подключения альтернативных словарей и использования системных вызовов.

Основное отличие программирования на языке Forth от программирования на ассемблере заключается в размещении операндов и концентрации результатов в стеке. В любой момент известно, где размещены промежуточные результаты и где будет итоговый результат. Такой механизм неявно присутствует в реализации любого языка высокого уровня (ЯВУ), использующего при организации вычислений выражения, функции и процедуры. Нередко системы программирования для ЯВУ поддерживают явную работу со стеком с помощью процедур PEEK (чтение головного элемента) POP (удаление) и PUSH (добавление), которые можно и реализовать над вектором или списком.

Фрагмент программы	*Примечание*
var ST : array [1000] of integer; PS : integer = 0;	Описание СД для реализации стека.
func PEEK (P:integer, S: array of integer): integer; begin PEEK := S [P] end:	PEEK (чтение головного элемента)
proc POP (var P:integer, S: array of integer); begin P := P- 1 end:	POP (удаление)
proc PUSH (E:integer, var P: integer; S: array of integer); begin P := P + 1 ; S[P] :+ E end:	PUSH (добавление),

Пример 7. Реализация стека на базе вектора. Pascal

Фрагмент программы	*Примечание*
(λ(st)(car st))	PEEK (чтение головного элемента)
(λ(x st)(cons x st))	PUSH (добавление)
(λ(st)(cdr st))	POP (удаление)

Пример 8. Реализация стека на базе списка. Lisp

Фрагмент программы	*Примечание*

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; }
};
	Шаблон с параметром для класса
	Границы стека и его размер
	
	Конструктор стека с заданным размером памяти
	Деструктор стека
	
	Метод пополнения стека
	Метод сокращения стека
	Метод доступа к размеру стека

Пример 9. Реализации стека на языке C++. 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.

Таблица 8

Парадигматическая характеристика языка Forth

Параметр	*Конкретика*
1. Эксплуатационная прагматика ЯП	Язык Forth – пример организации вычислений над стеком, что можно рассматривать 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
2.	Особенности системы понятий	Вместо системы команд над машинными словами в нем используется система операций на стеком, в котором концентрируется и общий результат работы программы. Лаконизм, постфиксный стиль
3.	Перечень понятий, распознаваемых на уровне абстрактного синтаксиса	Операции и значения
4.	Базовые средства	TRUE, FALSE, словарь команд, манипуляции со стеком, пополнение словаря.
5.	Семантические расширения	Слова разных размеров, модели средств ЯВУ, множественность стеков и словарей, управление компиляцией и временем исполнения частей программы
6.	Регистры абстрактной машины	S E C S – стек операндов и результатов. E – словарь определений. C – программа – поток операций и значений
7.	Категории команд абстрактной машины	Преобразование содержимого стека. Вычисление над стеком. Применение программируемых определений
8.	Реализационная прагматика	Стек – вектор заданного системой размера. Элементы стека – слова фиксированной разрядности. Система программирования использует пару интерпретатор-компилятор
9.	Парадигматическая специфика	Механизм реализации выражений в большинстве ЯВУ. Форт обладает концептуальным родством с языком Лисп

6. ПРОДУКЦИОННАЯ МАКРОТЕХНИКА

Макротехника дает весьма мощные, но не вполне безопасные, средства повышения выразительности ЯП. Рассмотрим устройство ряда макропроцессоров, используемых при обеспечении гибкости кода программ. Кроме обще известных ассемблерной макротехники и препроцессоров систем программирования [2, 3], достаточно интересны макропроцессоры GPM, Trac и макрогенераторы нестандартных языков программирования, применявшиеся при факторизации текстов программ, разрабатываемых одновременно на разные архитектуры. Макротехника обладает родством с методами конструирования регулярных выражений, техникой сопоставления данных с образцом в языках логического программирования (Snobol, Prolog), системами переписывания и современными языками разметки (xml, TeX и др.).

Хотя макротехника применяется к любым форматам данных, традиционно макросом называют средство замены строки на другую, полученную из исходной по заранее заданным правилам. Такую замену осуществляет макропроцессор, управляемый системой макросов. Макропроцессор перерабатывает текст, содержащий вызовы макроса и новые макроопределения, пополняющие систему макросов. Различают общие и локальные макросы, воздействующие на всю текущую программу или на часть ее текста. Системы макросов могут организовываться в библиотеки.

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. Возможен стиль нумерации позиций – это 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 GPM, TRAC [3], а также два макропроцессора из системы подготовки программ на языке SETL.

Таблица 9

Конкретизация понятий в языках макропроцессора

Понятие	*Макропроцессоры*
Атом (Элементарное)	символ – слово
Структура	строка – список,
Переменная	Имя, позиция в списке → Ид (Знач)
Значение	Строка
Выражение	§Макрос (а, в, ...) вызов макроса
Действие/Операция	Ввод определений, подстановка значения переменной
Условие/истина	0 пустая строка
Макрос	Макрос: Имя → Ид (шаблон подстановки)
Аргумент	элемент списка при вызове макроса
Вызов Фн/подпр	замена вызова макроса на преобразуемый шаблон
Определение Фн/подпр	Имя → представление шаблона
Идентификатор	номер в таблице определений – в словаре

Начнем с классификации макропреобразований по мощности допустимых воздействий на обрабатываемые данные, предложенной в 1973 году А. А. Берсом:

- чисто текстовая подстановка, обеспечивающая подготовку текстов по шаблонам и формулярам (GPM);
- условная подстановка, допускающая при формировании текста использование статических параметров (Setl);
- вычисляемые макропреобразования – вычисляются значения некоторых формул, возможно, с использованием средств основного языка программирования (PL/1).

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:

$$\text{Макро} = (\text{Прог} = \text{Текст}): \text{Прог} \rightarrow \text{Прог}'$$

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 АММ может быть задана парой <SCM, RM>, где RM = <E, C>, в которой C – строка, а E – вектор определений.

< Таблица_макросов_с_параметрами, Основной_текст >
e c → e’ c’

Таблица 10

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 11

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, быть автономным инструментом системы программирования, таким как текстовый редактор, оптимизатор или отладчик, или существовать самостоятельно как универсальный инструмент общего назначения.

Обычно макроопределения формируются с учетом границ строк, макровызовы могут выполняться за одни просмотр или до исчерпания при итеративном анализе текста. Возможно управление 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.

Отдельного решения требуют вопросы проявления в программах контекстов без макрообработки, таких как строковые константы и комментарии, выпадающие из общего строя языка программирования.

В текстовых процессорах встречается контекстная замена, контекстное редактирование и регулярные преобразования текста.

Модель макропроцессора может быть определена как композиция функций подстановки текста типа Subst и Sublis из описания Lisp-интерпретатора, дополненная таблицами для хранения значений параметров макросов для GPM и функциями сопоставления образцов для TRAC.

При поддержке динамически возникающих макроопределений традиционно обеспечивают самодостаточность, т.е. возможность развития макропроцессора своими средствами. Обычные требования:

- подстановка аргументов;
- использование библиотек;
- допущение переменных и структур данных;
- присвоение значений переменных;
- ветвления и переходы;
- циклы, иерархия и рекурсия;
- динамика обработки и создания новых макросов.

Макропроцессор получает логическое завершение, поддерживая динамически формируемые макроопределения, возможно используя макровызовы в аргументах. Являясь инструментом расширения средств системы программирования, макропроцессор по своей природе должен быть развиваемым. Традиционная при разработке систем программирования общность и минимизация понятий оказывается не лучшим идеалом для разработки информационных систем массового назначения, потому что решение практических задач всегда сопряжено со множеством тривиальных мелочей, связанных с нелогичными особенностями оборудования или привычками пользователей. Макротехника дает сравнительно недорогой метод учета таких мелочей на уровне специализации системных приложений.

Примеры программ с макросами:

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 10. Рекурсивное макроопределение факториала

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

Пример 11. 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 12. 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 13. 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 14. 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

Пример 15. Работа с шаблонами на языке Trac

Два интересных механизма макротехники были реализованы в проекте языка Setl при попытке его эффективной реализации посредством языка Little [5].

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 16. Представление зависимости от процесса формирования текста

Макропроцессоры – мощный инструмент повышения емкости действий, образующих процессы информационной обработки. Главное предназначение макросов в системах программирования – достижение гибкости и переносимости текстов программ, применяемых в разных условиях. Мно-

гие трудности такого применения макротехники связаны с проблемой контроля типов данных на уровне текста программы. Системы переписывания термов и разметки текстов пока не дали более практичных решений в этой области. Современные информационные системы как правило содержат макропроцессоры в качестве инструмента настройки на различные стандарты подготовки и обработки данных.

Общеизвестно, что макрос легче применять, чем определять. Внешняя простота введения макросов сопряжена с вероятностью 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.

Достаточно распространено определение макросов для нужд одной единственной программы, образующих с ней единое целое. Оно популярно при распараллеливании особо важных программ.

Макропреобразования используются при автоматизации формирования различимых имен, например, меток, подпрограмм или переменных в программе.

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

На практике макроассемблер выполняет роль расширенной системы команд. Такие команды могут обеспечивать специальные модули для обработки файлов с нестандартной информацией, например, распознавать текст с устаревшими или нестандартными шрифтами.

В общем случае интерпретирующие автоматы для макропроцессоров характеризуются возможностью многократной обработки данных, до тех пор, пока не будет получен результат без макровызовов. Соответствующая универсальная функция построения результата макрообработки получается реализационно не очень простой.

Подобные механизмы макрообработки текстов используются препроцессорами в стандартных системах программирования и текстовых процессорах. Иногда встречаются и более специализированные средства, использующие счётчиковые переменные, конструкторы уникальных имен, моделирующие иерархию модулей или параметризирующие зависимость вариантов программы от целевых архитектур.

Концептуально 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.

Специальные функции языка Lisp, определённые с индикаторами FEXPR и FSUBR, по существу работают как макроопределение, т.е. выполняют открытую подстановку аргументов, вычисляя их лишь по мере необходимости с помощью универсальной функции Eval [22].

Определения	*Примечание*
#DEFINE THEN #DEFINE BEGIN { #DEFINE END ;}	Можно определить

IF (I > 0) THEN
BEGIN
A = 1;
B = 2
END
	и затем написать

#DEFINE MAX(A, B) ((A) > (B) ? (A) : (B))	Макрос. Такая возможность обеспечивает «функцию максимума», которая расширяется в последовательный код, а не в обращение к функции
X = MAX(P+Q, R+S);	Макровызов. При правильном обращении с аргументами такой макрос будет работать с любыми типами данных
X = ((P+Q) > (R+S) ? (P+Q) : (R+S));	Результат. Нет необходимости в различных видах MAX для данных разных типов, как это было бы с функциями

Пример 17. Пример макросов языка C

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 18. 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. Результат работы макрогенератора – новая форма текста программы. Макротехника характерна для ЯВУ, поддерживающих открытую подстановку параметров, вызовы по необходимости при организации отложенных вычислений и специальных функций, использующих постобработку параметров (специальные функции в языке Lisp).

Таблица 12

Парадигматическая характеристика макропроцессора

Параметр	*Конкретика*
1. Эксплуатационная прагматика ЯП	Главное предназначение макросов в системах программирования – достижение гибкости и переносимости текстов программ, применяемых в разных условиях. Макротехника дает весьма мощные, но не вполне безопасные средства повышения выразительности ЯП. Макропроцессор часто используется в паре с ассемблером (макроассемблер) и другими ЯП. Макротехника применима и к другим форматам данных. В системах программирования макротехника применяется на двух уровнях: препроцессоры обычно формируют входной текст для компилятора, а макроассембле-

	ры выполняют сборку кода на уровне генерации ассемблерной программы или её объектного кода
2. Особенности системы понятий	Для макропроцессора программа – это содержащий макровызовы текст, при обработке которого программа преобразуется в новый текст программы, полученный в результате макропреобразований. 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
3. Перечень понятий, распознаваемых на уровне абстрактного синтаксиса	Определение макроса, вызов макроса, параметр макроса, константа
4. Базовые средства	Определение макроса. Сцепление строк. Размещение строки в таблице. Копирование строки. Применение макроса
5. Семантические расширения	Макропреобразования могут использовать локальные или глобальные переменные, вложенность областей действия определений, рекурсию. Макропроцессор может быть встроен в компилятор, быть автономным инструментом системы программирования, таким как текстовый редактор, оптимизатор или отладчик, или существовать самостоятельно как универсальный инструмент общего назначения. Условная подстановка, допускающая при формировании текста использование статических параметров Вычисляемые макропреобразования – вычисляются значения некоторых формул, возможно с использованием средств основного языка программирования. Для автономных макропроцессоров характерны специальные механизмы регулярного конструирования различимых текстов: Большинство макропроцессоров допускают переменные макропериода – глобальные и локальные макропеременные

6. Регистры абстрактной машины	E C E – вектор определений и параметров. C – программа, модифицируемая согласно макровызовам. Результат работы макрогенератора – новая форма текста программы
7. Категории команд абстрактной машины	Засылка определений Сцепление фрагментов в строку. Копирование параметров. Применение определения
8. Реализационная прагматика	Открытая подстановка без контроля границ стыковки фрагментов
9. 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

7. ЯЗЫКИ УПРАВЛЕНИЯ ПРОЦЕССАМИ

Большинство ЯП избегает средств для решения технических проблем управления процессами, в практике неизбежных. Рассмотрим базовые средства для решения таких проблем на уровне функционирования операционных систем (ОС), исполнения отдельных задач и разработки информационных систем.

Таблица 13

Конкретизация понятий в языке управления заданиями

Понятие	ОС
Атом (Элементарное)	Строка имя файла, номер процесса, устройство
Структура	файл, директория

Переменная	Имя → Ид (Знач)
Значение	Число строка
Выражение	Командная строка
Действие/Операция	над файлами и процессами
Условие/истина	Успех Существование
Функция/ подпрограмма/ сценарий	Имя → Ид (подпр) I/O, Error
Аргумент	поля в командной строке < > | &
Вызов Фн/подпр	Процесс → < протокол, код завершения >
Определение Фн/подпр	Имя → подпр Данное смена статуса файла
Идентификатор	имя файла имя переменной

Нередко представление процессов и файлов может быть унифицировано, что видно по сопоставлению команд по обработке файлов и манипулированию процессами:

Таблица 14

Параллелизм команд над файлами и процессами

	Файлы	*Процессы*
Вывести список	Ls	Ps
Сменить статус	Chmod	Bg Fg
Удалить	Rm	Kill
Сцепить последовательно	Cat	| конвейер
Показать кон- текст/задания	Env	Jobs
Создать файл/процесс	Cp – в новый файл	Fork
Сменить директо- рию/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. [10, 12, 23] 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 «лени-

вый» список, в конец которого можно встраивать новые элементы функцией Conc.

В качестве опорного языка рассмотрен Bash [23].

Абстрактный синтаксис мало отличается от формата командной строки, в которой перечисляются аргументы, размеченные спецсимволами, обозначающими направление передачи данных.

(КОМ A1 A2 … AK)

Спецификация команд абстрактной машина языка управления процессами AMQ = <SCQ, RQ> может быть определена над тройкой RQ = <E, C, D>, где E – контекст процесса, представленный как вектор записей с определениями файлов и переменных, C – строка, представляющая текущий процесс, D – очередь отложенных процессов.

< контекст_процесса, текущий_процесс, очередь_отложенных_процессов>

E C D

контекст_процесса – вектор записей Имя: Данные + stdin stdout
текущий_процесс – строка из команд
очередь_отложенных_процессов – вектор записей Имя: Данные

e c d → e′ c′ d′

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.

Простейшие действия:

Таблица 15

Типичный набор команд языка управления процессами

SCQ	*Описание команды*	*Примечание*
ECHO	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	

stdin – стандартный ввод. То, что набирает пользователь в консоли.
stdout – стандартный вывод программы.
stderr – стандартный вывод ошибок.

Обозначения:
(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 16

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)	

Рамочные конструкции для построения многоярусных условий вида

```
if ...
then ....
else
if ....
then....
else ....
```

для краткости и читаемости кода можно использовать структуру

```
if ..
then ...
elif ...
then ...
elif ...
```

Рамочные конструкции для итерирования

For …
In ..
Do …
done

while …
Do …
done

until
Do …
done

Определение	*Примечание*
case "${x##*.}" in	Переключатель
gz) gzunpack ${SROOT}/${x} ;;	Ветви переключателя
bz2) bz2unpack ${SROOT}/${x} ;;	
*) echo "Формат архива не определен."	
exit	Выход из переключателя
;;	
Esac	Завершение определения

Пример 19. Шаблон определения переключателя

Определение	*Примечание*

#!/bin/bash
 	
myvar="hello"
 	
myfunc() {
  local x
  local myvar="one two three"
  for x in $myvar
  do
  echo $x
  done
}
 	
myfunc
 	
echo $myvar $x
	
	Заголовок функции
	Локальные переменные
	
	Заголовок цикла по строке
	
	Тело цикла
	
	Вызов функции

Пример 20. Шаблон определения функции без параметров

Более подробно со средствами управления процессами на уровне ОС можно ознакомиться в книгах [13, 23].

Фрагменты	*Примечание*

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

Пример 21. 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. Суще-

ствование файлов в период обработки не всегда очевидно. Файлы могут участвовать одновременно в разных процессах.

- Выполнение команды рассматривается как событие. Такое событие 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.

В результате разработка программ для организации взаимодействия процессов отличается от подготовки обычных последовательных программ на весьма глубоком уровне, что показано на модели АМ для языка управления процессами. Такой автомат 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.

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.

Взаимодействие систем программирования для ЯВУ с операционной системой.

Таблица 17

Средства ввода-вывода в языке Pascal

Библиотечные процедуры	*Пример*	*Примечание*
Write	Write (1, x, “str”, 2.3, a+b)	Вывод значений любого типа.
Read	Read (x, y)	Ввод значений соответствующего типа.
Writeln	Writeln ()	Перевод строки при выводе данных.

Таблица 18

Средства ввода-вывода в языках C/C++

Библиотечные функции	*Пример*	*Примечание*
Printf	printf (“x = %d, y = % s”, x, y);	Вывод целого и строки
Scanf	scanf ("%d%s", &r, &c);	Ввод целого и строки
Cout	cout << “x = ” << x << “, y = ” << y ;	Вывод значений двух переменных ранее заданного типа.
Cin	cin >> x;	Ввод значения переменной

Вывод по библиотеке функций stdio.h	*Примечание*
printf (“x = %d, y = % s”, x, y);	Предварительное объявление типа значения

Пример 22. Форматный вывод данных в языке Си

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.*

Вывод по библиотеке классов iostream.h	*Примечание*
cout << “x = ” << x << “, y = ” << y ;	Тип значения устанавливается операцией «<<»

Пример 23. Потоковый вывод данных в C++/

Таблица 19

Lisp. Функциональный ввод-вывод данных

Псевдо-функции	*Пример*	*Примечание*
Print	(print ‘(1 a (2 b)))	Значение (1 a (2 b)) – равно аргументу
Read	(read)	Значение – данное любого типа, читаемое с клавиатуры

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.

Смягчение зависимости 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.

Учёт приоритетов отложенных процессов.

Появляются имена, локализованные внутри скриптов.

Таблица 20

Парадигматическая характеристика языков управления процессами

Параметр	*Конкретика*
1. Эксплуатационная прагматика ЯП	Абстрагирование от аппаратуры, обслуживание запросов от программных инструментов, обеспечение бесперебойной эксплуатации и функционирования оборудования
2. Особенности системы понятий	Основные понятия унифицировано сведены к понятию «файлы», обработка которых задаётся в формате командной строки. Исключается время, предпочитается отказ от ответов на вопрос происходят ли события строго одно за другим, что приводит к некоторой неимперативности управления процессами. Информационная обработка выглядит как семейство взаимодействующих процессов, выполняемых по отдельным программам – заданиям или сценариям, размещенным в файлах. Активно используются умолчания, раскрываемые в терминах текущих значений или системных переменных. Укрупняются данные – переход от ячеек и строк к долгоживущим файлам
3. Перечень понятий, распознаваемых на уровне абстрактного синтаксиса.	Команда, данное, направление потока данных, событие, переменная, выражение
4. Базовые средства	Успех-неудача, очередь, навигация в иерархии файлов, манипулирование файлами, задание статуса файла, визуализация и информационные сервисы
5. Семантические расширения	Распознавание конфигурации оборудования, умолчания в командной строке, подготовка и отладка скриптов, хранение истории и протоколов, настройки, прерывания, приоритеты, диагностика, порты, разные типы файловых систем и форматы файлов, распределение ресурсов, защита информации, справочная служба
6. Регистры абстрактной машины	E C D E – контекст_процесса, C – текущий_процесс, D – очередь_отложенных_процессов Результатом является код успеха или неудачи завершения процесса

7. 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
8. 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.
9. 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.

8 ДРУГИЕ ЯЗЫКИ НИЗКОГО УРОВНЯ

Машинно-независимый машинно-ориентированный язык Little – пример альтернативного подхода к решению проблемы низкоуровневого программирования [5]. Этот язык поддерживает традиционную процедурную организацию вычислений над структурированными битовыми строками. Основные конструкции аналогичны фортрановским, но с учетом системных решений, упрощающих реализацию компилятора. Ещё один пример – язык Эпсилон, ориентированный на обработку символьной информации,

успешно применявшийся в экспериментальном программировании [15]. Оба языка допускают порождение эффективного кода “хорошо” написанных программ.

Не менее интересна разработка настраиваемого макроассемблера Сигма, нацеленного на перенос программ относительно архитектуры [18], что может пригодиться при обеспечении архитектуро-независимых высокопроизводительных вычислений.

Появление компьютерных сетей и особенно массовое распространение Интернета, электронной почты и информационных сервисов выплеснуло проблематику параллельных процессов на рядового пользователя. Обыденные впечатления от доступа к новостным и рекламным сайтам, обмена фильмами и звукозаписями, пересылки фотографий для их печати в фотоателье, интерактивное общение и дневники - все это формирует интуитивную базу для вполне полноценного понимания средств и методов параллельного программирования. Клиенты всех таких информационных систем и ресурсов сталкиваются с достаточно сложным поведением сетевых конфигураций из разнородного оборудования, функционирование которого складывается не вполне очевидно. Появляется критерий - достижение понятности сетевой обработки информации, что в значительной мере требует представления механизмов взаимодействия процессов. Этот критерий имеет весьма важную составляющую - обеспечение грамотного поведения клиентов, принимающих решения относительно доступа к информационным ресурсам.

В середине 70-х годов активное исследование методов параллельного программирования рассматривалось не только как наиболее перспективное направление повышения эксплуатационных характеристик оборудования, но и как ведущее направление преодоления кризиса технологии программирования. В настоящее время рост интереса к параллельному программированию связан с переходом к массовому производству многоядерных архитектур.

Рассматривая типичные задачи организации процессов, базовые понятия, полезные при обсуждении проблем взаимодействия процессов, приходим к примерам средств и методов представления программ для организации параллельных процессов. С параллельными процессами мы реально встречаемся при работе на уровне ОС, при организации высокопроизводительных вычислений на базе суперкомпьютеров и многоядерных процессоров, при обеспечении сетевой обработки данных, при оптимизирующей компиляции программ и т.д. и т.п. Проблемы подготовки параллельных программ для всех столь разных работ обладают определенной общностью,

но есть и существенная специфика, требующая понимания разницы в критериях оценки качества программ и информационных систем для различных применений на базе моно- и/или мульти-процессорных комплексов, что будет рассмотрено отдельно.

Сложность перехода от навыков программирования обычных последовательных процессов к организации параллельных процессов в значительной мере обусловлена системой обучения, привычкой все упорядочивать, выстраивать в однозначные, безвариантные структуры, не замечая зависимости принятых решений от выбора структур и последовательности действий по их обработке.

По мнению Т. Хоара, «Параллельная композиция действий внешне не сложнее последовательного сочетания строк в языке программирования» [19]. Функциональное программирование на Лиспе и других языках благодаря своей необычности (не смягчившейся за 40 с лишним лет) и более гибкой модели организации вычислений позволяет предоставить простые примеры для показа непривычных идей параллелизма, интуитивное понимание которых не вызывает затруднений [22].

В таком случае реализация процесса – это функция, определяющая ход процесса по начальному событию. Таким событием может быть в частности готовность входных данных. Функциональная модель представления процессов позволяет легко описывать и взаимодействие процессов в виде функционалов, т.е. функций над процессами, представленными как функциональные переменные.

Дальнейшее развитие базовых понятий, используемых при организации параллельных процессов, связано с привлечением недетерминизма и общих разделяемых ресурсов. Управление процессами на уровне ОС, как правило, заключается в оперировании заданиями, сводимыми к передаче данных между устройствами и файлами.

Команды, образующие задания, используют такие объекты как переменные среды, потоки данных, протоколы исполнения команд и сценарии. Переменные среды обеспечивают параметризацию зависимости процессов от пользователей, используемых информационных систем и методов доступа к данным.

Основные события – инициализация процессов и систем, назначение стандартных потоков данных (ввод, вывод, ошибки), переключение режимов исполнения команд (приоритеты, фоновый режим), переадресация потоков, выяснение состояния файлов или устройств, задание времени исполнения команды, отмена команды.

Языки управления процессами характеризуются функциональной полнотой, обеспечивающей реализацию эффективных решений по доступу к устройствам и надежности информационной обработки долговременно хранимых данных.

Практичные решения в этой области выглядят как внедрение низкоуровневых средств управления процессами в среду языка высокого уровня.

Для полноты картины следовало бы ещё рассмотреть языки работы с базами данных и языки разметки для работы с сайтами, дающие понятия о транзакциях и дистанционном доступе к данным.

ЗАКЛЮЧЕНИЕ

Определение парадигм для ЯНУ не вызывает затруднений – в них явно видна ключевая идея, и 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 интерпретатора SECD, т.к. не гарантируют защиту контекста. Отличаются составом команд. 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.

Таблица 21

Наследование методов ЯНУ парадигмами ЯВУ

Механизм	Парадигма	*Примечание*
Ассемблер	ПИП 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

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.

СПИСОК ЛИТЕРАТУРЫ

1. Баранов С.Н., Колодин М.Ю. Феномен Форта // Системная информатика. Вып 4. Методы теоретического и системного программирования. – Новосибирск: Наука. Сиб. изд. фирма, 1995. – С. 193–271.
2. Болски М.И. Язык программирования Си. – М.: Радио и связь, 1988. – 96 с.
3. Браун П. Макропроцессоры и мобильность программного обеспечения. – М.: Мир, 1977. – 253 с.
4. Вирт Н. От Модулы к Оберону // Системная информатика. Вып 1. Проблемы современного программирования. – Новосибирск: Наука. Сиб. отд-ние, 1991. – С. 63–75
5. Городняя Л.В. Об одном подходе к синтезу транслятора на примере языка Литтл. // Теория и практика системного программирования. – Новосибирск, 1977. – С. 60–71.

6. Городняя Л.В. Основы функционального программирования. – М.: Интернет-Университет Информационных технологий. – http://www.intuit.ru, 2004. – 272 с. *(проверено 12.11.2014)*
7. Городняя Л.В. Парадигмы параллельного программирования в университетских образовательных программах и специализации // Всерос. научная конф. «Научный сервис в сети Интернет: решение больших задач». – Новороссийск-Москва, 2008. – С. 180–184.
8. Евстигнеев В.А., Касьянов В.Н. Графы в программировании: обработка, визуализация и применение. – С-Пб.: ЕХП-Петербург, 2003, – 1104 с.
9. Зубенко В.В., Протасов А.В. Язык программирования Фоли (Форт + Лисп) // Системная информатика. Вып 2. Системы программирования. Теория и приложения. Новосибирск: Наука. Сиб. изд. фирма, 1993. – С. 183–215
10. Иртегов Д.В. Введение в операционные системы СПб.: БХВ-Петербург, 2008. – 1040 с.: ил. – ISBN 978-5-94157-695-1.
11. Катцан Г. Язык Фортран 77. – М.: Мир. 1982. – 208 с.
12. Колин А. Введение в операционные системы. – М.: Мир, 1975. – 116 с.
13. Парамзин А.В. Введение в программирование на языке Ассемблера. – Новосибирск, 1993. – 180 с.
14. Пентковский В.М. Автокод Эльбрус. – М.: Наука. 1982. – 350 с.
15. Рар А.Ф. История ЭПСИЛОН. – http://www.iis.nsk.su/memories/rar_eps *(проверено 12.11.2014)*
16. Романов Ю. Ностальгия по БЭСМ-6 // Компьютерра. – 2007. – №34 .
17. Сингер М. Мини-ЭВМ PDP-11: программирование на языке ассемблера и организация машины. – М.: Мир, 1984. – 272 с.
18. Степанов Г.Г. Пути обеспечения переносимости программ и опыт использования системы СИГМА // Трансляция и преобразование программ. – Новосибирск: ВЦ СО АН СССР, 1984. – 9 с.
19. Хоар Ч. Взаимодействующие последовательные процессы. – М.: Мир, 1989 – 264 с.
20. Хорстман К. Scala для нетерпеливых. – ДМК пресс, 2013. – 408 с.
21. Кей С. Хорстманн Java SE 8. Вводный курс Java SE 8 for the Really Impatient. – М.: «Вильямс», 2014. – 208 с. – ISBN 978-5-8459-1900-7.
22. Хьювенен Э., Сеппанен Й. Мир Лиспа. – М.: Наука, 1994. – Т. 1,2.
23. McCarthy J. LISP 1.5 Programming Mannual. – The MIT Press., Cambridge, 1963. – 106 p.
24. Ritchie D.M., Tompson K. The UNIX Time-Sharing System // Bell System Technical J. – 1978. – Vol.57, N 6. – P. 1905–1929.

ОБОЗНАЧЕНИЯ

Обозначение	*Расшифровка*
АМ	Абстрактная машина
АС	Абстрактный синтаксис
ЖЦП	Жизненный цикл программ
КМ	Конкретная машина
ОС	
ПП	Парадигма программирования
РП	Реализационная прагматика
СД	Структуры данных
СП	Система программирования
ТД	Типы данных
ЭП	
ЯВУ	
ЯНУ	
ЯП	Язык программирования
AMA	
AMF	
AMM	
AMQ	
BIOS	
LAP	
PID	
RISC	
SECD	Абстрактная машина языка Lisp

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

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

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

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

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

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

[x] – содержимое памяти по адресу x

e[n] – содержимое n-го элемента контекста

A(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

СОДЕРЖАНИЕ

[2] Части 1 и 3–5 – отдельные препринты.

Л.В. Городняя

ПАРАДИГМЫ ПРОГРАММИРОВАНИЯ

Часть 2
Языки низкого уровня

Препринт
173

Рукопись поступила в редакцию 12.02.2015
Редактор Т. М. Бульонкова
Рецензент Ф.А. Мурзин

Подписано в печать 24.02.2015
Формат бумаги 60 × 84 1/16 Объем 3.77 уч.-изд.л., 4.1 п.л.
Тираж 60 экз.

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