ЛЕНИНГРАДСКИЙ ОРДЕНА ЛЕНИНА И ОРДЕНА ТРУДОВОГО КРАСНОГО ЗНАМЕНИ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

В.А.Тузов

ЯЗЫКИ ПРЕДСТАВЛЕНИЯ ЗНАНИЙ

Учебное пособие

Ленинград 1990

Печатается по постановлению
Редакционно-издательского совета
Ленинградского университета

Тузов В.А. Языки представления знаний: Учебное пособие. Л., 1990. 120 с.

Методы процессно-ориентированного программирования, в основе которых лежит принцип активизации исходной информации, применяются для построения класса языков представления знаний. Принцип активизации позволяет стереть грань между их декларативным и процедурным представлением. Предлагается конкретный язык этого класса и его реализация на языке Форт. Обсуждаются перспективы использования русского языка как языка представления знаний.

Для студентов и аспирантов, изучающих теоретические основы программирования, для специалистов в области информатики, для всех, кто интересуется перспективами развития вычислительных машин и языков представления сложно-организованной информации.

Рецензенты: канд.физ.-мат.наук В.В.Клюев (Ленингр.ун-т), канд. физ.-мат.наук С.Р.Мшагский (Ленингр.отд.Научно-исслед.ин-та связи)

ВВЕДЕНИЕ

В основе любой теории лежит ряд фундаментальных понятий, которые невозможно точно определить через другие понятия, но которые можно описать, перечислив основные наиболее характерные для них черты. Говоря о языках представления знаний, необходимо прежде всего описать ряд фундаментальных понятий, имеющих к ним непосредственное отношение.

I. Знания. Понятие "знания" выделилось в самостоятельное из составного понятия "база знаний", появление которого было вызвано необходимостью противопоставления понятию "база данных". В базе данных хранится пассивная информация, обработка которой в основном ограничена операциями выборки и переупорядочения, выполняемыми по запросам пользователя. База знаний содержит активную информацию. Таким образом, знания - это специфический тип информации.

Понятие "информация" является фундаментальным понятием, которое вряд ли можно свести к каким-либо более фундаментальным понятиям. Оно является объектом исследования многих научных дисциплин. В данной работе это понятие рассматривается с позиций информатики.

Информатика - бурно развивающееся научное направление, возникшее из практики общения человека с ЭВМ. В основаниях она совпадает с математикой, цели и приложения те же, что и у кибернетики. Это наука об информационных процессах, которая изучает методы и средства управления информацией. Информационный процесс - это сложноорганизованный, автоматически протекающий очень динамичный процесс, связанный с хранением, переработкой и передачей информации.

С точки зрения информатики знания представляют собой особый тип данных. Специфичность этого типа данных определяется четырьмя основными свойствами. Первое свойство - активность, т.е. способность знания самостоятельно обрабатывать информацию. Второе свойство - уникальность, т.е. каждое знание требует уникальной процедуры обработки, или само является уникальной процедурой. Два других свойства характеризуют совокупность знаний в целом. Одно из них - динамичность знаний - указывает на то, что совокупность знаний в процессе вычислений постоянно самоизменя-

ется, порождая новые знания и уничтожая старые. Другое - взаимозависимость знаний - предполагает наличие тесных связей между отдельными знаниями.

Таким образом, знания в отличие от обычных данных обладают некоторой "двуликостью": они сами могут обрабатывать информацию, одновременно подвергаясь постоянному изменению, вызванному как информационным взаимодействием с другими знаниями, так и внешним воздействием со стороны других знаний. Такая "двули - кость" знаний предъявляет особые требования к форме их представления в вычислительной машине. Они должны иметь свернутую форму, пригодную для внешней обработки, и развернутую форму, готовую для выполнения. В языках программирования первая форма - пассивная - может быть представлена в виде строки, вторая - активная - в виде процедуры. Результатом работы каждой процедуры является другая процедура в пассивной или активной форме. Операцию преобразования пассивной формы знания в активную назовем активизацией знания, обратную операцию - сверткой знания.

Специфичность знаний определяет основные черты языка их представления в ЭВМ. Во-первых, программа на этом языке должна представлять собой достаточно большой набор, вообще говоря,мелких информационных процессов. В ходе ее выполнения постоянно возникают новые и исчезают старые процессы, многие процессы выполняются одновременно, каждый из них имеет возможность обмениваться информацией с другими процессами, приостанавливать одни и запускать другие - ранее приостановленные процессы. Во-вторых, язык должен содержать средства активизации информации, т.е. средства ее преобразования в информационный процесс. Здесь первостепенное значение приобретает форма представления информационного процесса.

Небольшая по времени. но бурная история развития языков программирования продемонстрировала многообразие форм представления информации, в частности,информационных процессов. Но многообразие форм лишь обнажило единую сущность любого информационного процесса: каждый процесс является множеством взаимодействующих суперпозиций функций. Если основу живой материи составляет живая клетка, то основу информационного процесса - суперпозиция функций.

2. Язык. Язык есть средство описания информационных процессов. Предложение языка, совокупность предложений или текст - это застывшие формы процессов, превращающиеся в активный процесс во время выполнения человеком или вычислительной машиной.

В общем случае информационный процесс разбивается на совокупность одновременно протекающих (параллельных) элементарных процессов. Каждый элементарный процесс представляет собой суперпозицию функций. Класс всех используемых в языке функций разбивается на два подкласса. К первому относятся функции, заранее предопределенные в языке (базисные функции языка), ко второму - функции, определяемые во время выполнения текста.

Любой объект как в языке программирования, так и в естественном языке можно рассматривать как функцию, а обращение к нему - как обращение к функции. Любая переменная есть функция без аргументов. Отличие переменной от тех функций, которые обычно используются в языках программирования, заключается в том, что тело функции не меняется в процессе выполнения программы, а переменная меняет свое значение. Однако этим свойством обладают лишь функции в компилируемых языках. Но даже и в этом классе языков требование неизменяемости описаний функций соблюдается не всегда.

Структура (запись) с n полями представляет собой одноаргументную функцию, определенную на конечном множестве из n элементов, n-мерный массив является n-арной функцией, аргументы которой - целые числа из ограниченного диапазона, дерево есть суперпозиция функций, граф - связка рекурсивных функций и т.д.

Функциональная интерпретация объектов упрощает концептуальную основу теории и, главное, адекватна практике. Если говорить о простых объектах, таких, как структуры или массивы, то не столь важно, как их интерпретировать - как пассивные или активные, т. е. как функции. При переходе к более сложным объектам ситуация существенно меняется. Чем сложнее объект, тем больше в его структуре содержится информации о том, как он должен функционировать.

Предложение языка является записью суперпозиции функций. Существует огромное разнообразие форм представления суперпози-

ций, однако наиболее удобной для выполнения является бесскобочная инверсная запись, в которой аргументы и имена функций разделены пробелами. Вызов n-арной функции f с аргументами x_1, x_2,...,x_n имеет вид $x_1 x_2 \ldots x_n f$.

а произвольная суперпозиция записывается так: $f_1 f_2 \ldots f_N$.

где f_i - аргумент, имя функции или символ базисной операции. Например, выражение (a + b) * c - d в бесскобочной записи:

a b + c * d -

Произвольной суперпозиции можно дать имя, что позволит в дальнейшем использовать ее при построении новых суперпозиций. Именование суперпозиций дает возможность определять рекурсивные функции.

При вычислении значения суперпозиции существенную роль может играть способ ее выполнения, который определяется типом вызова фактического параметра. Например, выполнение суперпозиции $f(x,g(y))$ может начинаться с вычисления функции f (функциональный вызов второго аргумента функции f) или с вычисления функции g (вызов по значению). Результаты вычисления могут быть различными, если функция g не определена в точке y. Особое значение способ выполнения приобретает при определении рекурсивных функций. Продемонстрируем это на примере фразы "x говорит, что y всегда лжет". Этой фразе соответствует функция:

$f(x,y) = \text{говорит}(x, \forall z(\text{говорит}(y,z) \rightarrow \neg \text{cor}(z)))$.

Здесь cor - базисная функция семантического языка: cor(z) - z соответствует действительности.

При $x \neq y$ правая часть определяющего равенства не зависит от левой. При $x = y$ (x говорит, что он всегда лжет) получим: $f(x,x) = \text{говорит}(x, \forall z(\text{говорит}(x,z) \rightarrow \neg \text{cor}(z)))$.

Проблема заключается в том, что функция $f(x,x)$ является частным случаем второго аргумента суперпозиции, которая ее определяет. Подставив вместо z выражение $B = \forall z(\text{говорит}(x,z) \rightarrow \neg \text{cor}(z))$, получим уравнение, которому должна удовлетворять функция $f(x,x)$:

$f(x,x) = \text{говорит}(x, f(x,x) \rightarrow \neg B)$.

Если $f(x,x)$ определена, то это уравнение эквивалентно равенству:

$f(x,x) = \text{говорит}(x, \neg B)$.

Объединив его с исходным определением функции $f(x,x)$, получим

$f(x,x) = \text{говорит}(x, B \wedge \neg B)$,

что соответствует (в зависимости от способа выполнения) следующим предложениям русского языка: "x говорит нечто противоречивое, т.е. $B \wedge \neg B$", "x говорит нечто неопределенное", "x говорит" (а что - неизвестно). Эти следствия получены при предположении, что функция $f(x,x)$ определена. Но $f(x,x) = \omega$ (ω - всюду неопределенная функция) также является решением исходного уравнения - минимальным по количеству информации. Если $f(x,x) = \omega$, то фраза "x говорит, что он всегда лжет" на прагматическом уровне эквивалентна пустому тексту. Однако из множества всех решений, как правило, выбирается максимальное. В данном случае максимальному решению соответствует фраза: "x говорит нечто противоречивое, а именно, что он всегда лжет, и что он иногда не лжет".

Грамматика языка определяет формы обращения к базисным функциям и способы организации этих обращений в линейную последовательность. Иначе говоря, грамматика является средством построения суперпозиций. Это - узкое толкование понятия грамматики (для него иногда используется термин "синтаксическая грамматика). Полная грамматика включает в себя синтаксическую и содержит описания базисных функций. Между синтаксической и полной грамматиками лежит целый спектр частичных грамматик, различающихся степенью полноты описания функций. Знать язык - значит знать полную грамматику его.

Таким образом, каждая грамматика состоит из двух разнородных, но тесно связанных между собой частей - синтаксической грамматики и совокупности описаний функций. Первая определяет синтаксическую структуру (синтаксис) языка, вторая - его семантическую структуру (семантику). Синтаксис языка определяет способ представления текста, семантика - способ выполнения текста. Языки с одинаковой семантикой могут отличаться друг от друга лишь именами функций и формой обращения к ним, откуда следует, что все свойства синтаксиса в значительной степени предопределены семантикой языка.

Кроме синтаксической и семантической структур в языке выделяют прагматическую структуру (прагматику). Это наиболее сложная и наименее изученная структура языка. Прагматика связывает язык с той или иной предметной областью: язык программирования - с кругом тех реальных задач, на решение которых он ориентирован, естественный язык - с той частью реальной действительности, которую с его помощью можно описать. Прагматика языка предопределяет синтаксис и семантику языка.

Если семантика определяет способы вычисления значения (как результата выполнения текста), то прагматика - способы его интерпретации в окружающем мире. Вне этой интерпретации значение, будучи оторванным от действительности, не имеет реального смысла. Семантика языка и семантика (смысл) конкретного текста - существенно различные понятия. Семантическая структура языка хотя и является отражением окружающего мира, тем не менее полностью от него абстрагирована и может функционировать в сознании человека или в вычислительной машине вне зависимости от реальной действительности. Смысл текста есть результат взаимодействия семантики языка, информации об окружающем мире, которой обладает носитель языка, и текущего состояния внешней среды - области интерпретации значений.

Не существует точных границ, разделяющих синтаксис, семантику и прагматику языка. При описании даже простейших языков могут возникнуть вопросы, считать ли то или иное свойство языка синтаксическим или семантическим. Например, при описании арифметических выражений приоритетность операций (операции типа умножения имеют больший приоритет, чем операции типа сложения) можно считать синтаксическим свойством, и тогда синтаксическая грамматика будет иметь вид

```
E :: E + T | T;    T :: T * F | F;
F :: (E) | I
```

(Здесь E - выражение, T - терм, F - множитель, I - идентификатор.) В противном случае различие между выражением, термом и множителем стирается, и грамматика становится проще:

```
L :: L + L | L * L | (L) | I ,
```

но при этом усложняется описание семантических функций.

Еще более размыта граница между семантикой и прагматикой. Какое-либо новое явление действительности может не иметь даже собственного имени и поэтому с позиции языка является строго прагматическим. Но со временем оно может занять свое место в языке, пополнив его семантическую структуру. Аналогично при создании пакетов прикладных программ используется прагматическая информация, и каждая программа имеет прагматический смысл, однако пользователь пакета может рассматривать его как часть семантической структуры расширенного языка.

Размытость границ между структурами ставит под сомнение деление языка на структуры. Но отсутствие четких границ между частями целого - типичное явление реальной действительности, и если его принимать во внимание, то любое понятие окажется под сомнением.

3. Понятие вычислимости. Каждая функция является частично определенной. Понятие определенности-неопределенности функции - фундаментальное понятие, лежащее в основе формирования принципов вычислимости. Неопределенность значения при вычислении конкретного текста означает бессмысленность этого текста с точки зрения семантики языка. На прагматическом уровне она означает отсутствие информации: такой текст эквивалентен пустому тексту. Пустой текст не может быть ни истинным, ни ложным. С другой стороны, противоречивый текст не может быть выполнен и его смысл не определен, следовательно, он также эквивалентен пустому тексту.

Информационный процесс поставляет новые знания лишь тогда, когда он определен. При аварийном прерывании процесса (если оно не было специально запрограммировано) возникает неопределенная ситуация, которая может служить причиной альтернативного выбора, но которая, как правило, не несет нового знания. Таким образом, понятие знание-незнание тесно связано с понятием определенности-неопределенности. Доопределение функции является источником получения нового знания.

Одним из основных утверждений классической теории алгоритмов является утверждение о принципиальной невозможности доопределения функций: существует частично-рекурсивные функции, которые невозможно доопределить до всюду определенных вычислимых функ-

ций. В основе подобных утверждений лежит признание потенциальной неограниченности вычислительных ресурсов - времени вычислений и объема памяти. Принцип потенциальной неограниченности приводит к такой идеализации понятия вычислимости, которое не только неадекватно реальной действительности, но не имеет с ней ничего общего. В то же время прямое отрицание этого принципа, состоящее в том, чтобы ранее фиксировать конечную границу объема памяти или числа шагов вычислений, равносильно утверждению о невозможности построения теории вычислений. Однако существует компромиссное решение. С одной стороны, однажды начатое вычисление функции f в точке x представляет собой реальный процесс f_T, и как всякий реальный процесс он ограничен и в пространстве, и во времени некоторым числом T. С другой стороны, запись функции f в языке L для каждого x определяет множество реальных процессов, которое неограничено ни в пространстве, ни во времени. Значением функции f в точке x следует считать $\lim_{T\to\infty} f_T(x)$.

Каковы бы ни были вычислительные ресурсы, почти для любой частично-рекурсивной функции f существует такое N, что для всех $n>N$ функция $f(n)$ невычислима при заданных ресурсах. Поэтому функция f_T определена лишь на конечном множестве значений n. Предельная функция $\lim f_T(x)$ практически не вычислима, но, являясь идеализацией множества практически вычислимых функций $f_T(x)$, она может служить основным объектом теории вычислений и с точки зрения теории является вычислимой функцией. Будем называть такие функции реально вычислимыми функциями. Ясно, что любая частично-рекурсивная функция может быть доопределена до всюду определенной реально вычислимой функции.

Теория реально вычислимых функций существенно отличается от классической теории алгоритмов. В рамках этой теории невозможно доказать существование алгоритмически неразрешимых проблем. Как правило, два алгоритмических языка не эквивалентны по мощности друг другу. По любому языку L можно построить реально вычислимую функцию F, которую невозможно записать в языке L. Тем самым, по любому языку L можно построить более мощный язык L'. Множество всех реально вычислимых функций не счетно. Классическая теория степеней неразрешимости во многом адекватна начальной иерархии языков реально вычислимых функций (если термин "степень неразрешимости" заменить на термин "степень сложности вычислений"). Кванторы общности и существования эквивалентны предельному переходу и поэтому повышают и степень сложности, и мощность языка.

Если заменить понятие истинности-ложности на понятие определенности-неопределенности, то предикат $\forall x f(x)$ ($\exists x\ f(x)$) эквивалентен утверждению, что функция f - всюду определенная функция (функция f определена хотя бы в одной точке). Предикат $\forall x \exists y\ P(x,y)$ эквивалентен утверждению, что существует функция $f(x)$, такая, что суперпозиция $P(x, f(x))$ определена тогда и только тогда, когда определена функция $f(x)$. Это одна из уточненных формулировок аксиомы выбора Цермело. Эта аксиома утрачивает абсолютный и приобретает относительный характер: если функция $P(x,y)$ вычислима относительно языка L (т.е. имеет запись в языке L), то функция $f(x)$, вообще говоря, принадлежит более мощному языку L', который можно построить по языку L. Такое уточнение позволяет избежать многих парадоксов теории множеств.

4. Принцип активизации. Обладая определенной спецификой, языки представления знаний навязывают особый - функциональный стиль программирования. В основе функциональных методов программирования лежит принцип активизации, сущность которого заключается в том, что исходные данные решаемой задачи отображаются не на пассивные типы данных (строки, списки, деревья и т.п.), а на активные - функции, суперпозиции функций, наборы функций и т.п. Эти методы наиболее эффективны при построении алгоритмов, предназначенных для обработки сложноорганизованной информации. Их эффективность тем выше, чем сложнее структура исходных данных. Если исходные данные не разложимы на составные части (например являются числами, символами, логическими или байтовыми значениями), то функциональные методы вырождаются в обычные методы программирования.

Чтобы избежать терминологической путаницы, будем различать термины "функциональное программирование" - стиль программирования, который навязывают лиспоподобные языки, и "функциональные методы программирования". Их различие сравнимо с различием меж-

ду языками Лисп и Форт. В языке Лисп каждая функция представима в виде списка, но не каждый список является функцией. В языке Форт ситуация обратная: любой список может быть представлен в виде функции, но не любая функция является списком. При функциональном методе программирования существует один тип данных - функция. Не разложимые на составные части данные являются простейшими нульарными функциями. Таким образом, понятие функции здесь используется в самом общем (математическом) смысле этого слова. В самом общем, но не в самом абстрактном: функция - это запись процесса, готового к выполнению. Обращение к функции активизирует процесс.

Списки, деревья, графы, тексты - это объекты достаточно сложные, поэтому в вычислительной машине их следует представлять в виде функций, готовых к выполнению. Формальными параметрами этих функций являются другие, более простые функции. Тем самым класс всех функций разбивается на иерархически упорядоченные подклассы. Нижние уровни этой иерархии занимают элементарные функции, которые будем называть операциями. Операции используют в качестве аргументов данные простейших типов. Иерархия является основой структурирования исходной задачи, которая автоматически распадается на ряд более простых подзадач.

Набор операций обладает эффектом комбинаторного взрыва: их небольшое число позволяет строить огромное количество необходимых функций. Такая возможность существенно упрощает задачу модификации построенных функций.

Общая схема применения функциональных методов заключается в следующем. Во-первых, класс X исходных объектов отображается на класс Σ параметризованных функций: $\Psi: X \to \Sigma(Y_1, Y_2, \ldots, Y_N)$, где Y_i - формальные параметры данного класса объектов. Во-вторых, каждой функции $F(x_1, x_2, \ldots, x_n)$ $(x \in X)$ ставится в соответствие набор операций A1,A2,...,AN , которые являются фактическими параметрами, определяющими n процессов: $x_1(A1, A2, \ldots, AN), \ldots$ $\ldots, x_n(A1, A2, \ldots, AN)$. Параллельное выполнение этих процессов вычисляет значение функции F . Их запуск и завершение осуществляется главным процессом, описание которого и является описанием функции F :

:F (: A1 A2...AN :) < инициализация > < запуск > < завершение > ;

Здесь (: и :) синтаксически ограничивают параметры, а фактически являются функциями, которые осуществляют передачу параметров A1,A2,...,AN, связывая их с формальными параметрами $Y_1, Y_2, \ldots, Y_N$.

Процесс построения функции Ψ будем называть процессом активизации исходного класса объектов. Функция Ψ преобразует исходный объект из пассивной формы в активную. Передача фактических параметров A1 конкретизирует объект, настраивая его на вычисление функции F . Для точного определения процессов активизации и конкретизации необходим язык, обеспечивающий их адекватное описание. Трудно представить себе список, произвольный текст или граф как функции на языке ПЛ/I (или любом другом, подобном ему языке). Мало подходят для этой цели и такие языки как Лисп, Планер, Пролог. Для исследования функциональных методов программирования вполне подходящим оказался язык Форт. Поэтому этот язык был выбран в качестве языка представления объектов достаточно сложной природы - списков, графов, деревьев, автоматов, текстов и грамматик. Функциональное представление этих объектов занимает в памяти ЭВМ примерно вдвое, а иногда и втрое, больше места, чем их свернутая форма. Это недостаток функциональных методов, но он окупается двумя преимуществами: функциональные методы обеспечивают, во-первых, существенное снижение трудоемкости процесса программирования, во-вторых, эффективность по быстродействию получаемых программ.

Продемонстрируем функциональный стиль программирования на примере перевода целых двоичных чисел в десятичные. Двоичное число отображается на суперпозицию функций, содержащую две (пока неопределенные) функции: 0 и 1. Например, двоичное число x = 11001 будет записано в виде: x 1 1 0 0 1 ; , т.е. в виде суперпозиции в бесскобочной записи. Функции 1 и 0 можно определить следующим образом:

: 0 2 * ; : 1 2 * 1 + ;

где + и * - операции десятичного сложения и умножения. Обращение 0 x оставит в стеке десятичную запись числа x .

В этом примере как отдельные цифры - 0 и 1 , так и число x представлены в активной форме. Функционирование активизированной информации существенно зависит от того, в каком окружении

она выполняется. Например, изменив функции + и * , можно построить перевод в любую другую систему счисления.

5. Смешанные вычисления. Процесс активизации имеет ряд сходных черт со смешанными вычислениями. Действительно, пусть $P(x,y)$ - произвольная функция и мы хотим запрограммировать ее, активизируя аргумент x . Тогда функция fx , в которую должен быть преобразован аргумент x , должна удовлетворять равенству:

$$fx(y) = P(x,y) ,$$

т.е. fx есть остаточная программа, которая может быть получена из $P(x,y)$ в результате смешанных вычислений.

Смешанные вычисления представляют собой функциональную операцию $S(P^3,x_0)$: используя запись P^3 тела функции P и конкретное значение x_0 аргумента x , операция S строит тело новой функции Px_0 . Поэтому развитый функциональный язык должен содержать средства, поддерживающие процесс смешанных вычислений. Но если активизация аргументов позволяет из меньшего получить нечто большее, то смешанные вычисления из большего получают нечто меньшее. Кроме того, реализация смешанных вычислений не сравнима по сложности с реализацией обычных операций, таких, как CAR или CDR в языке Лисп, или с реализацией, например, механизма возвратов в Прологе или Плэнере. Поэтому прежде чем приступать к реализации смешанных вычислений, необходимо ответить на вопрос, на каком классе задач применение смешанных вычислений позволит окупить издержки на их реализацию.

Процесс смешанных вычислений в классе арифметических задач очень сложен сам по себе и, как правило, не упрощает исходную программу. Практическая бесконечность множества арифметических значений и слабая структурированность входных данных - основные причины, из-за которых смешанные вычисления практически не применимы к этому классу задач.

Смешанные вычисления в классе булевских функций очень просты и, как правило, существенно упрощают исходную программу. То же самое можно сказать о всем классе функций, определенных на конечных множествах. Если представить такую функцию (с n аргументами) n-мерной таблицей, то смешанные вычисления по любому аргументу понижают размерность этой таблицы на единицу, т.е. в среднем уменьшают объем описания функции вдвое. Применив процесс смешанных вычислений к таблице решений, можно построить соответствующую ей блок-схему. Каждая блок-схема определяется порядком аргументов, по которым ведутся смешанные вычисления.

Другим классом задач, при решении которых смешанные вычисления не бесполезны, является класс так называемых текстовых задач, т.е. задач, связанных с обработкой текстов произвольной длины. Все задачи этого класса можно разбить на две группы: задачи, для решения которых необходимо учитывать синтаксическую и семантическую структуры текста (например перевод, интерпретация текста), так как именно они определяют процесс обработки, и задачи чисто механической обработки текстов (например разбиение текста на строки, абзацы, страницы). Смешанные вычисления могут быть эффективно использованы по крайней мере при программировании задач первой группы.

Наконец, третий класс задач содержит задачи, связанные с обработкой сложноорганизованной информации, т.е. информации, представленной в виде списков, деревьев, графов, сетей и т.п. С точки зрения смешанных вычислений этот класс очень близок к классу конечных функций. Отличие лишь в том, что в этом классе приходится иметь дело с циклическими или рекурсивными алгоритмами, что технически усложняет процесс смешанных вычислений, который, как правило, порождает связку рекурсивных функций. В классе конечных функций рекурсия или циклы исключены.

6. Недетерминированные и параллельные процессы. Частичная определенность функций является средством управления множеством информационных процессов. При выполнении единичного процесса неопределенность какой-либо функции вызывает аварийное прерывание. При выполнении множества процессов прерывание одного из них может инициировать запуск или приостановку других процессов. В этом заключается сущность управления при помощи частичной определенности функций.

Запуск множества процессов предполагает либо выполнение всех процессов, либо выполнение хотя бы одного процесса. В первом случае говорят о параллельных процессах, во втором - о недетерминированном процессе. Хотя между этими двумя классами процессов имеется много общего, а с теоретической точки зрения недетерминированные процессы являются частным случаем парал-

лельных процессов, однако на практике частный случай часто оказывается особым случаем, для исследования которого требуется особый подход. Эти два класса процессов объединяет и противопоставляет классу детерминированных процессов множественность выполняемых процессов, но между собой они находятся в такой же зависимости как кванторы общности и существования. Операция отрицания, которая обеспечивает эквивалентность кванторов, связана с доопределением частичной функции - с точки зрения классической теории алгоритмов проблемой неразрешимой, а с точки зрения практики программирования - проблемой достаточно сложной. В этом заключается первооснова различий между этими двумя классами процессов.

Если недетерминированный процесс имеет имя, то в языке программирования он представляется в виде недетерминированной функции, в противном случае - в виде альтернативного оператора, частными случаями которого являются условный оператор и оператор выбора. Недетерминированная функция это функция, которая имеет несколько экземпляров своего описания. При обращении к ней управление передается первому экземпляру, при возврате - второму и т.д. Часто требуется не просто выполнить функцию, но и построить суперпозицию из успешно выполненных экземпляров недетерминированных функций: однажды найденный маршрут может служить основой для решения множества конкретных задач.

Функция $F(x_1, x_2, \ldots, x_N)$, значениями аргументов которой являются объекты сложноорганизованной структуры, порождает N взаимодействующих между собой параллельных процессов. В организации параллельных процессов главную роль играют механизмы синхронизации. В языках ПЛ/I, Алгол-68 для целей синхронизации используются семафоры, в языке Ада - механизм рандеву, некоторые языки используют сети Петри и т.д. Каждый из этих механизмов описывает некоторый класс взаимодействий одновременно выполняющихся процессов. Функциональные методы программирования однозначно определяют свой специфический способ синхронизации. Оператор вызова функции может оказаться разорванным и различные его части могут находиться в разных процессах. Процесс останавливается тогда и только тогда, когда происходит обращение к отдельной части оператора вызова. В простейшем, но важном

случае, когда оператор вызова содержит лишь имя функции, различные части этого имени разбросаны по разным процессам. Каждая такая часть является обращением к неописанной функции. Останавливая процесс, она попадает в управляющую память. В момент, когда управляющая память будет содержать все части полного имени, происходит их конкатенация и обращение к функции с этим именем. После ее выполнения (или после ее приостановки) все процессы, остановленные частями ее имени, запускаются на выполнение. Такова общая схема синхронизации параллельных процессов, порожденных в результате активизации аргументов функции F .

7. Принципы формализации ЕЯ. Естественный язык (ЕЯ) обладает многими свойствами, необходимыми для представления знаний. Но для того чтобы использовать его как средство представления знаний в ЭВМ, необходимо решить сложнейшую проблему содержательного (семантического) анализа текстов на ЕЯ. Вопросам формализации отдельных сторон ЕЯ посвящено немало работ как у нас в стране, так и зарубежом. Однако никем еще не предпринималась попытка формализовать ЕЯ в целом. Более того, претензия на научно обоснованное использование ЕЯ в общении с машиной многими воспринимается как дурной тон. Этот вывод - следствие многочисленных неудач в области машинного перевода, автоматического реферирования, аннотирования и т.п. Но неудача - не повод для пессимизма, а стимул для дальнейших исследований.

Формализовать язык значит задать машине информацию о языке так, чтобы машина могла выполнять любые тексты на этом языке. Поверхностное понимание процесса формализации ЕЯ часто приводит к искаженному представлению, что формализованный язык - абстрактная, полностью оторванная от содержания схема с простой логической структурой. Адекватная формализация не отрывает исследуемый объект от содержания. Наоборот, главное ее предназначение - служить средством точного и эксплицитного представления содержания, т.е. такого представления, которое позволяло бы выделять различные его части и исследовать динамику их взаимодействия. Применительно к языку это означает, что адекватное описание языка есть, главным образом, описание его семантической структуры.

Семантика определяет синтаксис. Синтаксическая структура является отражением (внешним проявлением) семантической струк-

туры. Но даже признавая эти утверждения верными, не следует пренебрежительно относиться к синтаксическим аспектам языка. Семантика оставляет отчетливый отпечаток на синтаксисе, и этот факт дает возможность исследовать многие языковые явления на синтаксическом уровне.

Тем не менее, степень формализации языка определяется степенью формализации его семантики. По отношению к естественным языкам это утверждение приводит к глубоким последствиям. Так как в основе семантической структуры ЕЯ лежит модель окружающего нас мира, то возможность точного описания ЕЯ прямо зависит от возможности построения точных моделей реальной действительности. Ясно, что реальная действительность неформализуема, следовательно, казалось бы абсурдна даже постановка вопроса о возможности формализации ЕЯ. Но это неправильный вывод. Дело в том, что для формализации семантики ЕЯ достаточно языковой модели мира, а эта модель содержится в словарном составе ЕЯ. Тем самым, можно говорить о формальной модели естественного языка, не выходя за его рамки. Прагматика языка (что машина должна делать, когда фраза понята) остается за этими рамками.

Таким образом, одной из важнейших проблем формализации ЕЯ является проблема извлечения модели реального мира из словарного состава ЕЯ. Извлечение - не совсем точное слово: словарь лишь поставляет материал для построения модели. Модель сама является языком, причем языком формальным. (Поэтому в дальнейшем вместо термина "модель" будем использовать термин "семантический язык".) Этот язык может быть построен по правилам, аналогичным правилам построения, например, языков программирования, и его основные понятия можно пояснить, сопоставляя им наиболее близкие программистские понятия.

В основе семантической структуры любого языка, в том числе и ЕЯ, лежат две подструктуры: структура данных и структура управления. Структура данных ЕЯ представляет собой пару (M,P), где M - семейство множеств значений, P - набор базисных операций, каждая из которых определена на декартовом произведении некоторых множеств из M со значениями в одном из множеств семейства M. Структура управления ЕЯ определяет способы хранения информации в памяти и доступа к ней, в частности, способ идентификации данных.

Предложение на семантическом языке представляет собой суперпозицию функций, которая строится из базисных операций. На этом уровне базисные операции остаются неопределяемыми. При задании конкретной предметной области они могут быть определены как функции на множествах объектов этой предметной области, реализуя связь семантики языка с его прагматикой. Предложение на ЕЯ также является суперпозицией (внешних) функций f_i, в результате выполнения которой вычисляется соответствующее ей предложение на семантическом языке.

Каждая функция f_i связывается с конкретным словом языка. На поверхностном уровне функциональность слова проявляется в его многозначности. В общем случае каждое свое конкретное значение слово приобретает только в предложении. Нельзя говорить о значении, например, функции sin , можно говорить лишь об области ее значений. Значение любой функции определяется лишь после задания конкретных значений ее аргументов. Рассматривая слово как символ функции, мы получаем возможность вычислять его значение (но только после задания значений его аргументов). В частном случае слово может быть нульарной функцией, т. е. константой.

Признание функциональной природы слова приводит к необходимости признать индивидуальность, неповторимость каждого слова. А это означает, что каждое слово должно иметь свое собственное семантическое описание. Тем самым возможны по крайней мере два принципиально различных подхода к реализации ЕЯ: либо создавать единую процедуру обработки текстов, либо рассматривать каждое предложение как единичную процедуру, выполнение которой обеспечивает его обработку.

Любой языковый факт связан со словом или словосочетанием (а во фразеологизмах - с предложением), поэтому каждое слово следует рассматривать как активную процедуру. Это является программистским отражением простой истины, что несмотря на жесткие рамки языка, которому принадлежит слово, оно обладает ему лишь одному присущей индивидуальностью, имеет свое собственное лицо. Говоря иначе, семантика языка выражается через лексическое значение слова, а лексика слова определяет грамматику языка. Отсю-

да следует, что не существует главных и второстепенных членов предложения. Такие понятия как подлежащее, сказуемое, дополнение и т.п. в какой-то степени отражают грамматическую структуру ЕЯ, но с их помощью можно излагать лишь некоторые элементарные факты, ни в малейшей степени не затрагивая сущности языка. (Неадекватность традиционной грамматики, основанной на этих понятиях, аналогична неадекватности римской системы счисления, которая не позволяет формализовать даже операцию сложения целых чисел.)

Для того чтобы представить хотя бы в общих чертах, чем является процедура, связанная с конкретным словом, рассмотрим проблему построения программного обеспечения с точки зрения теоретической вычислимости. Теоретически любой процедурный механизм может быть представлен одним алгоритмом (например нормальным алгоритмом Маркова). Допустим, что такой алгоритм P построен. Будем подавать ему на вход синтаксически правильные предложения, содержащие какое-то конкретное слово. Например: A разбивает B ломом (C). На выходе будем получать некоторые семантические структуры $P_{\text{разбивает}}(A,B,C)$. Процедура $P_{\text{разбивает}}$ является частичной программой, которая получается из алгоритма P в результате смешанных вычислений при неполностью заданном входе. Эта процедура $P_{\text{разбивает}}$ и связывается со словом 'разбивает'. Индивидуальность, неповторимость каждого слова обеспечивает простое и естественое разложение алгоритма P на множество P_w процедур-функций, каждая из которых соответствует некоторому слову w . Сложным предложениям будет соответствовать суперпозиция нескольких процедур.

Вынося из алгоритма P все процедуры P_w, получим алгоритм P_c, который уже не зависит от словарного состава языка. Алгоритм P_0 обеспечивает только программную поддержку процесса выполнения суперпозиций функций.

Механизм смешанных вычислений пронизывает всю структуру ЕЯ. Он позволяет получать из более общих функций частичные функции, которые могут использоваться тогда, когда значения некоторых аргументов неизвестны или нерелевантны сообщению. Например, для глагола 'покупает' (кто-что x, кого-что y, у кого z , за что v) первый и второй аргументы обязательны. Если значение второго аргумента отсутствует, то используется более конкретная функция -

'тратит'= 'покупает'(x,y,#,v) , которая не имеет аргумента z вовсе, а аргумент y для нее не обязателен. Из общей функции 'перемещается' (кто-что x, куда y, откуда z, по чему v, как w) путем подстановки конкретных значений некоторых аргументов можно получить 82 частичных функции, связанных с глаголами движения.

Хотя эти общие рассуждения, конечно, не дают ясного представления о том, как описывать функции, связанные со словами ЕЯ, однако они уже сейчас позволяют сформулировать ряд преимуществ, которыми обладает функциональный подход (с точки зрения построения программного обеспечения обработки текстов) по сравнению со всеми ранее предлагавшимися методами описания ЕЯ. Во-первых, общая процедура обработки текстов - очень сложная и даже необозримая для достаточно большого фрагмента языка - распадается на сотни и даже на тысячи (для словаря порядка 100 тыс. слов) мелких процедур, которые строятся независимо друг от друга. Во-вторых, программное обеспечение оказывается независимым от словарного состава языка и, обеспечивая лишь поддержку процесса выполнения суперпозиций, может быть использовано на всех уровнях обработки текста - морфологическом, синтаксическом и семантическом. В-третьих, иерархическое описание функций, многие из которых получаются из более общих функций за счет сужения областей определения, в частности, путем подстановки конкретных значений аргументов, адекватно отражает практику построения толковых словарей и, по существу, является формальным толковым словарем. В-четвертых, описание каждого слова в виде функции есть основная часть описания перевода (другой частью является синтаксический анализатор) с внешнего языка на семантический язык.

Любой естественный язык - столь сложный объект, что разработка методов или подходов его формального описания всего лишь первый шаг на длинном пути построения формальной модели конкретного ЕЯ. (В дальнейшем в качестве конкретного ЕЯ будем рассматривать только русский язык.) Основными этапами этого пути являются:

- разработка семантического языка, который не зависит от национальных особенностей ЕЯ;
- функциональное описание слов русского языка;
- построение синтаксического и морфологического анализаторов.

Подавляющая часть этой работы носит чисто лингвистический характер и аналогична работе по созданию толкового словаря, но словаря математически точного, без "дурной" рекурсии, со строгим разграничением определяемых и неопределяемых понятий. В результате этой работы должен быть создан машинный фонд русского языка.

Морфология, синтаксис и семантика русского языка неразрывно связаны между собой, поэтому анализ (морфологический, синтаксический и семантический) различных аспектов языка необходимо выполнять одновременно. Этим обеспечивается детерминированность процесса анализа, исключается перебор возможных вариантов разбора предложения, что существенно повышает эффективность работы анализатора. Однако описания трех уровней языка не только могут, но и должны быть независимы.

В заключение следует отметить, что название работы не соответствует реальному состоянию теории или практики программирования: языков представления знаний нет. Системы искусственного интеллекта и в особенности появившиеся в последнее время экспертные системы используют определенные средства представления знаний - семантические сети, порождающие правила типа "ЕСЛИ, ТО", фреймы и т.п., однако это еще лишь предыстория. История языков представления знаний еще не началась, но будущее за ними. Их ростки - в современных языках программирования. Непросто по росткам определить их дальнейшую судьбу. Каким может и должен быть язык представления знаний? Дать пусть пока не точный и не полный ответ на этот вопрос - основная цель этой работы. Эта цель в какой-то степени оправдывает ее название.

В рамках общей теории два класса языков - языки программирования и естественные языки - образуют единый класс языков, который можно и нужно исследовать с единых позиций. Расхождение между языками этих классов велико, но это расхождение только в их "возрасте". Однако языки программирования столь быстро "взрослеют", что это различие начинает стираться (по крайней мере в теории). Развитый язык представления знаний должен обладать алгоритмическими возможностями мощного языка программирования, а также гибкостью и выразительностью естественного языка.

Глава I. КОНЦЕПТУАЛЬНАЯ ОСНОВА ЯЗЫКА ПРЕДСТАВЛЕНИЯ ЗНАНИЙ

В концептуальной основе языка представления знаний лежит принцип активизации, общая схема применения которого описана во введении. В данной работе этот принцип применяется для решения ряда задач обработки сложноорганизованной информации.

Данные делятся на пассивные и активные. Пассивные данные предназначены для обработки некоторыми внешними по отношению к ним процедурами. Активные определяют некоторый информационный процесс (или множество информационных процессов) и сами способны обрабатывать информацию. За первым типом данных сохраним термин "данные", второй тип данных будем называть "знаниями".

Классический подход к программированию определяется соотношением: Данные + Алгоритм = Программа. В рамках искусственного интеллекта был выработан новый подход, сущность которого выражает соотношение: Знания + Вывод = Система. Подход, который предлагается в данной работе, также может быть охарактеризован в столь же лаконичной форме соотношением: Данные + Активизация = Программа. Основная цель работы состоит в том, чтобы показать, что первые два подхода являются вырожденными частными случаями третьего. Достижение этой цели позволит ответить на главный вопрос: каким должен быть язык представления знаний.

В японском и английском проектах машин пятого поколения предлагается взять за основу будущих языков программирования языки Лисп и Пролог, а для их аппаратной поддержки создать соответственно функциональную машину и машину логического вывода. Кроме того, предлагается создать еще четыре класса машин: машину реляционной алгебры, машину абстрактных типов данных, машину потока данных и обновленную машину фон Неймана. В этой коллекции есть все, кроме здравого смысла. Конечно, лучше шесть разнородных машин, чем одна машина фон Неймана. Но еще лучше - один класс машин, построенных на единой концептуальной основе с единым языком программирования (или группой однородных языков). Однако выбор единственного пути из десятка возможных оправдан лишь тогда, когда он правилен. Можно гарантировать правильность такого выбора, если показать, как построить единый язык, который, во-первых, перекрывает возможности всех названных выше языков и машин, во-это-

рых, является достаточно простым в реализации, в-третьих, обеспечивает необходимое быстродействие написанных на нем программ, и, наконец, в-четвертых, содержит средства программной поддержки систем обработки информации на естественном языке.

Ни один из современных языков программирования не удовлетворяет первому и четвертому условиям. Среди всех языков, которые в какой-то степени удовлетворяют этим условиям, прежде всего следовало бы назвать язык Форт. Этот язык выделяет из группы других широко распространенных языков программирования то, что он поддерживает процесс активизации исходных данных, обеспечивая тем самым возможность применения принципиально новых методов программирования. Для того чтобы точно сказать,о какой принципиальной новизне идет речь, необходимо хотя бы кратко охарактеризовать существующие методы программирования.

§ I. Классификация методов программирования

Алгоритм решения задачи разрабатывается человеком. Этот алгоритм может быть сформулирован на любом в общем случае сколь угодно абстрактном языке (АЯ). На этот язык не накладывается никаких ограничений кроме одного: он должен быть понятен самому разработчику алгоритма. Сущность программирования заключается в построении отображения алгоритма на абстрактном языке в алгоритм (или программу) на конкретном языке программирования (ЯП). Общие принципы и методы построения отображения лежат в основе той или иной методологии программирования.

Три составные части языка определяют его сущность: структура данных ($S_{ЯП}$), структура управления ($U_{ЯП}$) и логистика ($L_{ЯП}$). Каков бы ни был абстрактный язык, на котором первоначально записывается алгоритм решения задачи, он также содержит эти три составные части: $S_{АЯ}$, $U_{АЯ}$, $L_{АЯ}$. В основе процесса программирования лежит построение отображения структур абстрактного языка на структуры языка программирования. В принципе каждая из трех структур абстрактного языка может быть отображена на любую из структур языка программирования. Таким образом, получаем девять типов отображений:

T1 :	$S_{АЯ}$	$S_{ЯП}$	T5 :	$S_{АЯ}$	$L_{ЯП}$
T2 :	$U_{АЯ}$	$U_{ЯП}$	T6 :	$U_{АЯ}$	$S_{ЯП}$
T3 :	$L_{АЯ}$	$L_{ЯП}$	T7 :	$U_{АЯ}$	$L_{ЯП}$
T4 :	$S_{АЯ}$	$U_{ЯП}$	T8 :	$L_{АЯ}$	$S_{ЯП}$

T9 : $L_{АЯ}$ $U_{ЯП}$

Каждое из этих отображений определяет свой собственный стиль программирования, или,как иногда говорят, технологию программирования.

Если основное внимание при программировании обращено на отображение T1 структур данных, то основу технологии составляют абстрактные типы данных и принцип модульности. Действительно, для того чтобы построить отображение T1, как правило, необходимо укрупнить типы данных или операции над ними, т.е. повысить уровень структуры данных ЯП до уровня структуры данных АЯ, а процесс укрупнения данных и операций над ними прямо приводит к понятиям модульности и абстрактных типов данных.

Отображение T2 структур управления всегда связано либо с понижением уровня структуры управления АЯ, либо с повышением уровня структуры управления ЯП. В первом случае мы имеем дело с технологией структурного программирования сверху-вниз, во втором случае с разработкой процедурного механизма ЯП. При структурном программировании задача разбивается на подзадачи до тех пор, пока структура управления АЯ не совпадет со структурой управления ЯП. Разработка процедурного механизма с целью повышения уровня управляющей структуры ЯП закономерно подводит к таким понятиям как сопрограмма, планируемый вызов процедуры, взаимодействующие процедуры и т.п.

Отображение T3 лежит в основе логических методов программирования. Подавляющее большинство современных языков программирования либо вовсе не содержит логистической структуры, либо она является несущественной вспомогательной частью структуры данных (например, система приведений в языке Алгол-68). Однако некоторые языки программирования, например Планнер и Пролог, имеют развитую логистическую структуру и ориентированы на поддержку логических методов программирования. Развитой логистической структурой обладают языки искусственного интеллекта. Почти все задачи, решаемые

в рамках искусственного интеллекта, описываются так называемой лабиринтной схемой, которая определяет логику (или логистику - в общем случае) задачи. Отображение этой схемы на логистическую структуру ЯП есть отображение типа Т3.

По существу все эти технологии являются технологиями ручного программирования. И здесь единственный путь автоматизации процесса программирования - повышение уровня ЯП. Таким образом появились языки высокого уровня, затем языки сверхвысокого, сверхсверхвысокого уровня и т.д. Но структура языка сверхвысокого уровня слишком "высока" по сравнению со структурой языка вычислительной машины, что вынуждает заменить процесс компиляции программ на их интерпретацию, а это, в свою очередь, приводит к существенной потере эффективности (по быстродействию) языка. Сейчас становится совершенно очевиден тот факт, что дальнейшее повышение уровня ЯП по крайней мере нецелесообразно. Поэтому практика программирования спонтанно порождает принципиально новые методологии более эффективного построения программ. В основе этих методологий лежат отображения Т4 - Т9. Как часто бывает, новое - это хорошо забытое старое. Многие принципы, определяющие сущность новых технологий, использовались и ранее, но на новом витке развития вычислительной техники и языков программирования они приобретают качественно новую окраску.

Отображение Т4 активизирует пассивные данные, преобразуя их в активные процессы. Это отображение есть другая форма принципа активизации. Принцип активизации был детально исследован при создании систем построения трансляторов (СПТ) и системы обработки сложноорганизованной информации, достаточно подробное описание которой приводится в следующей главе. Опыт построения этих систем дает право сказать, что принцип активизации позволяет резко сократить объем ручного программирования по крайней мере при решении задач перевода и подавляющего большинства задач обработки символьной и сложноорганизованной информации. Отображение Т4 лежит в основе функциональных методов программирования. Кроме того, оно в значительной степени определяет методы объективно-ориентированного программирования, а также ряд других методов, близких к функциональным, например таких, как метод Джексона и метод Варнье.

Если отображение Т4 позволяет по структуре объекта построить

операции, необходимые для его обработки, то отображение типа Т5 дает возможность по совокупности операций построить логистическую структуру, т.е. структуру переходов, которая является необходимой и, как правило, достаточной для автоматического построения алгоритма, решающего исходную задачу. Иначе говоря, отображение Т5 задает спецификацию задачи, достаточную для автоматического синтеза программ. Этот вид синтеза (так называемый структурный синтез программ) нашел применение в системе Приз [8].

Отображение типа Т6 лежит в основе построения всех интерпретаторов и частично в системах, поддерживающих процесс смешанных вычислений. Традиционный процесс трансляции, макрогенерация и компиляция-выполнение (например, при работе текстового интерпретатора языка Форт) являются классическими примерами методов программирования, основанных на отображениях двух типов - Т4 и Т6. Отображение Т6 резко повышает универсальность средств программного обеспечения, существенно понижая, как правило, его эффективность. Однако это в полной мере относится лишь к интерпретаторам и смешанным вычислениям, если их рассматривать с самой общей точки зрения. Что касается макрогенерации, то поскольку она носит двойственный характер, ее эффективность и универсальность зависят от соотношения отображений Т4 и Т6. Процессы трансляции и компиляции в силу своей жесткой ориентации на конкретный язык достаточно эффективны, но не универсальны.

Отображение типа Т7 пока не нашло широкого применения в методах и системах программирования, но в будущем следует ожидать появления очень интересных - с теоретической точки зрения - и очень полезных для практики программирования систем, основанных на отображениях этого типа. Характеризуя целевую направленность этих систем, уже сейчас можно сказать, что в основном они будут ориентированы на расшифровку смысла программ, на более доступное для понимания описание смысла алгоритма, а в конечном счете - и смысла задачи. Кроме того, системы этого типа могут быть полезны как средство автоматического структурирования программ или в общем случае как средство автоматического преобразования управляющей структуры программы к более простому виду. И, наконец, эти системы могут использоваться для доказательства свойств программ, в частности, их правильности.

Отображение типа Т8 на интуитивном уровне всегда связывается либо с типизацией данных и многоуровневостью языка, в случае если необходимо реализовать логистическую структуру алгоритма более эффективно, либо с интерпретаторами, которые реализуют языки с развитой логистической структурой. В первом случае создаются языки типа Алгол-68 с достаточно развитой системой видов значений (быть может, пока недостаточно развитой) и системой приведений. Во втором случае - такие языки как Планнер (механизм возвратов, вызов процедуры по шаблону) или Пролог, для которого необходим интерпретатор логики предикатов. Отображение типа Т8 служит основой логического синтеза программ.

Отображение типа Т9 используется либо в системах структурного синтеза вместе с отображением типа Т5, либо как средство снятия кванторов существования и общности. При помощи кванторов можно существенно повысить уровень языка, и поэтому они могут входить как составная часть абстрактного языка, но кванторные операторы абсолютно недопустимы в любом языке программирования (разве лишь в узкоспециализированных, ориентированных на конкретный класс задач, языках). Переход от кванторного выражения к эквивалентному алгоритму на языке программирования может быть более просто осуществлен через логистику языка.

Каждое из девяти типов отображений определяет некоторую разновидность технологии программирования. Эти технологии можно назвать "чистыми технологиями". Из сказанного выше видно, что некоторые отображения тесно связаны друг с другом, поэтому на практике иногда трудно отделить один подход к программированию от другого. Более того, некоторые подходы целесообразно применять в комплексе, что может обеспечить гораздо больший эффект. Так, абстракция типов данных, укрупняя обрабатываемые данные и операции над ними, позволяет существенно упростить управляющую структуру алгоритма, тем самым облегчая процесс построения отображения Т2. Технологии программирования, основанные на отображениях Т4, Т5 и Т9, взаимно дополняют друг друга. Если структура обрабатываемых объектов достаточно сложная, то используется отображение Т4, в противном случае - комплекс отображений типа Т5 и Т9. Отображение Т6, являясь обратным к отображению Т4, может эффективно использоваться там, где Т4 не применимо, и наоборот. В идеале можно предста-

вить язык программирования, который поддерживает все девять технологических разновидностей, и этот язык является тем единым языком, о котором говорилось в связи с машинами пятого поколения.

Приведенная классификация методов программирования носит весьма приблизительный характер, так как многие существенно различные методы попадают в один класс. Это объясняется тем, что каждая из трех структур языка состоит из разнородных подструктур и при построении более точной классификации за основу следует взять отображения этих подструктур. Однако приведенная классификация вполне достаточна, чтобы пояснить, почему особое внимание уделяется отображению Т4. Это отображение ставит во главу угла управляющую структуру языка программирования. Для его эффективного использования необходимо, чтобы язык имел развитую управляющую структуру, в частности, он должен содержать недетерминированные и параллельные процессы. Параллельные процессы потому, что отображение Т4 ставит в соответствие n-арной функции n параллельных процессов, а недетерминированные процессы, с одной стороны, являются естественным обобщением детерминированных, а с другой - адекватным, а во многих случаях и более мощным средством описания логистической структуры решаемой задачи. Развитая управляющая структура языка содержит необходимые средства программной поддержки всех девяти типов отображений.

При построении отображения Т1 сложность программирования определяется сложностью построения высокоуровневых операций, адекватных операциям предметной области. Если исходные данные имеют элементарную структуру, то методы, основанные на абстракции типов данных и модульности, приобретают первостепенное значение. Отображение Т4 в этом случае практически не применимо. Но если исходные данные не элементарны, то отображение Т4 позволяет резко сократить объем ручного программирования.

Ясно, что развитая управляющая структура языка, содержащая недетерминированные процессы, обеспечивает программную поддержку отображений Т2, Т3, Т5 и Т9.

Отображение Т6 является обратным к отображению Т4. Это отображение позволяет перейти от описания множества функций к описанию одной функции, что составляет сущность процесса параметризации функций - процесса, обратного процессу активизации. Если про-

вести полную параметризацию всех функций, описывающих решение задачи, то можно получить единую программу, интерпретирующую исходные данные. Однако эффективность (по быстродействию) полученных в процессе параметризации алгоритмов, как правило, существенно ниже непараметризованных алгоритмов. Тем не менее два процесса - активизации и параметризации - взаимно дополняя друг друга, часто позволяют найти разумный компромисс между объемом описания функций и эффективностью их выполнения. (Активизация данных, как правило, требует большего объема памяти.)

Отображение T7 не используется в практике программирования, поэтому оставим его без комментариев.

Сложность построения отображения T8 связана со сложностью построения интерпретатора логистической структуры. Отображение T4 способно существенно повысить быстродействие такого интерпретатора. Кроме того, в этом случае оно служит средством типизации данных и обеспечивает естественный переход от одноуровневого языка к многоуровневому.

§ 2. От задачи - к алгоритму

На прагматическом уровне понятие задачи распадается на две составные части: что дано и что требуется получить. Математикой накоплен огромный опыт, во многих случаях дающий ответ, как по тому, что дано, получить то, что требуется. В наиболее концентрированной форме этот опыт заключен в аксиоме выбора Цермело, которая утверждает, что предикат $\forall X \exists Y\ P(X,Y)$ эквивалентен существованию функции $f(X)$, превращающей этот предикат в тождество. Предикат $P(X,Y)$ является математической формулировкой задачи, функция $f(X)$ - ее решением. В чистой математике часто игнорируется вопрос, как по предикату P построить функцию f. Прикладная математика не может игнорировать этот вопрос. Информатика должна идти дальше: в сферу ее исследований входит не только проблема взаимозависимости предикат-функция, но, главным образом, методы построения информационных процессов, вычисляющих функцию f.

Рассмотрим вначале два элементарных примера, демонстрирующих переход от математической формулировки задачи к решающему ее алгоритму.

Алгоритм деления. Даны целые неотрицательные числа x и y ($0 \le x, 0 < y$). Получить частное и остаток от деления x на y, т.е. пару чисел (q,r), удовлетворяющих предикату:

$$x = q * y + r \ \&\ 0 \le q \ \&\ 0 \le r < y. \qquad (1)$$

Пара $(0,x)$ является решением равенства

$$x = q * y + r, \qquad (2)$$

но она может не удовлетворять неравенству $r < y$. Если пара (q,r) удовлетворяет равенству (2), то пара $(q+1, r-y)$ также является его решением, если $0 < q + 1 \ \&\ 0 \le r - y$. Отсюда получаем правило перехода от пары (q, r) к паре $(q + 1, r - y)$:

$f1(q,r)$ = если $y \le r$ то $(q + 1, r - y)$.

Это правило оказывается достаточным для построения информационного процесса, который имеет общую форму:

< начальные присваивания > < итерационный цикл > < завершение >,

а на конкретном языке, например, Паскаль может быть записан в виде:

```
begin  q: = 0;   r: = x;
   while r ≥ y do
      begin  q: = q + 1 ;
             r: = r − y
      end
end
```

Наибольший общий делитель. Даны целые неотрицательные числа x и y. Получить наибольшее z, удовлетворяющее системе равенств:

$$z * z1 = x$$
$$z * z2 = y$$

Если $x = y$, то x является искомым решением. Иначе, если $x > y$, то можно получить эквивалентную систему:

$$z * z1 = x - y$$
$$z * z2 = y$$

Отсюда, учитывая симметрию чисел x и y, получаем два правила перехода:

$f_1(x,y)$ = если $x < y$ то $(x, y - x)$,
$f_2(x,y)$ = если $y < x$ то $(x - y, y)$,

которые приводят к программе:

```
begin a :=x ; b := y;
   while  a ≠ b  do
      if a < b then  b := b − a
```

```
        else  a := a - b
    end
```

Эти два примера будут служить иллюстрацией общей схемы перехода от исходного предиката $\exists YP(X,Y)$ к решающему его алгоритму $f(X)$. Четыре шага отделяют предикат от алгоритма.

Первый шаг: построить предикат $Q(X,Z)$, эквивалентный предикату $\exists YP(X,Y) : \exists ZQ(X,Z) \equiv \exists YP(X,Y)$.

Предикат $Q(X,Z)$, как правило, является более простым, чем предикат $P(X,Y)$, часто - его составной частью. Он служит для задания либо начального состояния программы, либо области определения функции $f(X)$. В первом примере: $Q(x,y,r) = 0 \le x \,\&\, 0 < y \,\&\, 0 \le r$, во втором примере: $Q(x,y) = 0 \le x \,\&\, 0 \le y$.

Второй шаг: построить систему $f_1, f_2, \ldots, f_n$ преобразований, сохраняющих инвариантным предикат $P(X,Y)$:

$$P(f_i(X,Y)) \equiv P(X,Y).$$

(Иначе говоря, ни одно из преобразований f_i не меняет исходную задачу.) В первом примере $f_1(y,q,r) = (y, q+1, r-y)$, во втором $f_1(x,y) = (x, y-x)$, $f_2(x,y) = (x-y, y)$.

Третий шаг (который необходим лишь тогда, когда преобразования $f_i(X,Y)$ не зависят от второго аргумента): построить предикат $P_0(X,Y)$, такой, что если (X,Y) удовлетворяет ему, то легко вычислить функцию $Y = f_0(X)$. Более формально это означает, что предикат $P_0(X,Y)$ имеет вид

$$P_0(X,Y) = R(X,Y) \,\&\, (R_1(X,Y) \,\&\, Y = g_1(X) \vee \ldots \vee R_s(X,Y) \,\&\, Y = g_s(X))$$

и по нему легко строится функция $f_0(X)$ (в виде абстрактного условного оператора), превращающая $P_0(X,Y)$ в тождество:

$$\begin{aligned} f_0(X) = \text{ЕСЛИ}\ & R(X,g_1(X)) \,\&\, R_1(X,g_1(X)) \to g_1(X) \ \square \\ & R(X,g_2(X)) \,\&\, R_2(X,g_2(X)) \to g_2(X) \ \square \\ & \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \\ & R(X,g_s(X)) \,\&\, R_s(X,g_s(X)) \to g_s(X) \\ \text{КЕ} & \end{aligned}$$

В первом примере - это предикат $(x = q * y + r) \,\&\, 0 < r < y$, во втором - система двух равенств, когда $x = y$.

Четвертый шаг: доказать, что система преобразований $f_1, f_2, \ldots, f_n$ полна, т.е.

$Q(X,Z_0) \equiv \exists i_1 i_2 \ldots i_k (P(x, [\sigma_{i_1 i_2 \ldots i_k}(X,Z_0)]_2) \equiv P_0(\sigma_{i_1 i_2 \ldots i_k}(X,Z_0)))$, где $[X_1,X_2]_1 = X_1, \sigma_{i_1 i_2 \ldots i_k}(X,Z_0)$ - суперпозиция функций (в польской инверсной записи): $X\ Z_0 f_{i_1} f_{i_2} \ldots f_{i_k}$, (X,Z_0) - начальное состояние.

Связь между исходным предикатом $\exists YP(X,Y)$ и решающим его алгоритмом $f(X)$ выражает теорема.

Если система $f_1, f_2, \ldots, f_n$ эквивалентных преобразований полна, то функция $f(X)$, вычислимая абстрактным оператором цикла (в общем случае недетерминированным):

$$\begin{aligned} f(X) = (X,Z_0)\,[\ & \text{ЦИКЛ}\ \neg P_0(X,Y) \to \\ & \text{ЕСЛИ}\ \exists Z\ Q([(f_1(X,Y))]_1, Z) \to f_1(X,Y) \ \square \ \ldots \\ & \ldots \square\ \exists Z\ Q([(f_n(X,Y))]_1, Z) \to f_n(X,Y) \\ & \text{КЕ} \\ & \text{КЦ}]_2 \end{aligned}$$

является решением предиката $\exists Y\ P(X,Y)$.

В первом примере $Q(f_1(y,q,r)) = 0 \le x \,\&\, 0 < y \,\&\, 0 \le r - y$, во втором $Q(f_1(x,y)) = 0 \le x \,\&\, x < y$, $Q(f_2(x,y)) = y < x \,\&\, 0 \le y$.

Доказательство. Пусть $\exists YP(X,Y)$. Тогда в силу полноты системы $\{f_i\}$ $\exists i_1 i_2 \ldots i_k P_0(\sigma_{i_1 i_2 \ldots i_k}(X,Z_0))$. Существует минимальное k_0, такое, что $\exists i_1 i_2 \ldots i_{k_0} P_0(\sigma_{i_1 i_2 \ldots i_{k_0}}(X,Z_0))$. Это означает, что $\forall j (j \le k_0 \supset \neg P_0(\sigma_{i_1 i_2 \ldots i_j}(X,Y)))$.

Покажем, что $\forall j (j \le k_0 \supset \exists Z Q([\sigma_{i_1 i_2 \ldots i_j}(X,Y)]_1, Z))$. Если $\neg \exists Z Q([\sigma_{i_1 i_2 \ldots i_j}(X,Y)]_1, Z)$ для некоторого $j_0 (1 \le j_0 \le k_0)$, то $\neg \exists YP(\sigma_{i_1 i_2 \ldots i_{j_0}}(X,Y))$. Это противоречит тому, что $f_0(\sigma_{i_{j_0+1} \ldots i_k}(\sigma_{i_1 i_2 \ldots i_{j_0}}(X,Y)))$ является решением предиката $\exists YP(\sigma_{i_1 i_2 \ldots i_{k_0}}(X,Y))$. Следовательно, суперпозиция $\sigma_{i_1 i_2 \ldots i_{k_0}}(X,Y)$ вычисляется программой $f(X)$. С другой стороны, в силу полноты системы преобразований программа $f(X)$ заканчивает работу и, следовательно, для некоторых $i_1 i_2 \ldots i_k$ выполнен предикат $P_0(\sigma_{i_1 i_2 \ldots i_k}(X,Y))$. f_i - эквивалентное преобразование, значит

$[\sigma i_1 i_2 \ldots i_k(X,Y)]_2$ является решением предиката, что и требовалось доказать.

Частными случаями общей схемы перехода являются схема эквивалентных преобразований, схема последовательных приближений (в частности, схема неподвижной точки) и схема полного перебора. Первый пример - деление чисел - является типичным примером метода последовательных приближений, если в качестве предиката P взять предикат $\exists q,r((x = q * y + r) \& 0 \le r < y)$. Второй пример - нахождение НОД - иллюстрирует метод эквивалентных преобразований. Задача поиска заданного числа в массиве чисел может служить примером задачи, для решения которой используется схема полного перебора. Поскольку многие задачи принадлежат классу задач, каждая из которых описывается одной из частных схем перехода, имеет смысл рассмотреть эти частные случаи более подробно.

В схеме эквивалентных преобразований предикат $Q(X,Z)$ не зависит от Z и поэтому не содержит квантора существования. Первый шаг схемы перехода может быть сформулирован в следующем виде: построить предикат $Q(X)$, такой, что $Q(X) \equiv \exists Y P(X,Y)$. Предикат $Q(X)$ задает в явном виде область определения функции $f(X)$. Преобразования $f_i(X,Y)$ не изменяют второго аргумента. Поэтому на втором шаге получаем тождество: $P(f_i(X),Y) \equiv P(X,Y)$. Поскольку аргумент Y в процессе преобразования предиката не получает конкретного значения, требование, чтобы функция $f_0(X)$ была легко вычислимой, становится обязательным. Тождество на четвертом шаге, функция $f(X)$ и вычисляющая ее программа могут быть записаны в виде

$$Q(X) \equiv P(X,f(X)) \equiv \exists i_1 i_2 \ldots i_k P_0(\sigma_{i_1 i_2 \ldots i_k}(X), f_0(\sigma_{i_1 i_2 \ldots i_k}(X))); \quad f(X) = f_0(\sigma_{i_1 i_2 \ldots i_k}(X))$$

```
ЦИКЛ ¬P0(X,f0(X)) →
    ЕСЛИ Q(f1(X)) → f1(X) □ ... □ Q(fn(X)) → fn(X)  КЕ
КЦ f0(X)
```

В схеме последовательных приближений предикат $Q(X,Z)$ задает область начальных приближений. Тождество $\exists Z Q(X,Z) \equiv \exists Y P(X,Y)$ гарантирует, что для каждого X найдется начальное приближение Z_0, обеспечивающее сходимость процесса приближений к искомому решению, если система $\{f_1, f_2, \ldots, f_n\}$ полна. Преобразования

$f_i(X,Y)$ не меняют первого аргумента. Предикат $P_0(X,Y)$ совпадает с предикатом $P(X,Y)$. Поэтому $f_0(X) = f(X)$. Тождество на четвертом шаге, функция $f(X)$ и вычисляющая ее программа имеют вид: $Q(X,Z_0) = \exists i_1 i_2 \ldots i_k P(X, \sigma i_1 i_2 \ldots i_k(X,Z_0))$

$$f(X) = \sigma i_1 i_2 \ldots i_k(X,Z_0)$$

```
(X,Z0) ∃ZQ(X,Z) → [ЦИКЛ ¬P(X,Y) →
                       ЕСЛИ f1(X,Y) → f1(X,Y) □ ...
                       ... □ fn(X,Y) → fn(X,Y)  КЕ
                   КЦ]2
```

Схема полного перебора использует минимум сведений о том, как решать задачу. Преобразования f_i неизвестны, и вместо них используется один перечисляющий значения второго аргумента алгоритм $h(i)$ $(0 \le i \le N)$. Предикат Q (за неимением лучшего) совпадает с P. Поэтому тождество на четвертом шаге, функция $f(X)$ и вычисляющая ее программа имеют вид

$$P(X,Y_0) = P(X,h(h\ldots h(0)\ldots))$$

$$f(X) = h(h\ldots h(0)\ldots)$$

```
(X,0) ЦИКЛ¬(P(X,h(i)) & i ≠ N + 1) → (X,i + 1) КЦ
```

В данном случае полнота системы преобразований означает, что при заданном X существует Y_0, удовлетворяющее предикату P.

Схему полного перебора можно интерпретировать как частный случай схемы последовательных приближений. Менее очевидно, но однако и схему эквивалентных преоорезований можно рассматривать как частный случай схемы последовательных приближений. С другой стороны, схема последовательных приближений является частным случаем как схемы эквивалентных преобразований, так и схемы полного перебора. Таким образом, все три типа схем оказываются равносильными и, более того, каждая из них равносильна общей схеме перехода. Однако эта равносильность сугубо формальна: сведение схемы одного типа к другому связано с изменением математической постановки задачи. Семантика задачи (предикаты $P(X,Y)$ и $P_0(X,Y)$) однозначно определяет тип схемы.

Схема перехода от математической формулировки задачи к решающему ее алгоритму полностью покрывает "интеллектуальное откровение" Дейкстры [7] и "апостольское деяние" Грина [5], но она ни в коем случае не исчерпывает содержание науки программи-

рования. Что касается работ Дейкстры и Гриса, то они могут служить введением в программирование задач численного анализа. Но в этом классе задач программистские проблемы столь ничтожны по сравнению с общематематическими проблемами, что только использование математической логики в качестве средства формализации семантики языков программирования может сделать их значительными, да и то лишь в глазах начинающего программиста.

Теория Дейкстры излагается вне какой-либо связи с основаниями математики, поэтому такие понятия как слабейшее предусловие, сильнейшее постусловие, преобразователь предикатов производят на неискушенного читателя впечатление некоего "откровения". Однако аксиома выбора Цермело, конкретизированная в виде общей схемы перехода, не только содержит аналогичные понятия, но и способна объяснить место каждого из них в рамках математической постановки задачи. Предикат $Q(X,Z)$ является слабейшим предусловием, и слабейшим его делает естественное требование эквивалентности: $\exists Z Q(X,Z) \equiv \exists Y P(X,Y)$. Этот предикат выражает необходимое и достаточное условие существования решения задачи. Предикат $P(X,Y)$ является инвариантом алгоритма, так как преобразования $f_i(X,Y)$ – преобразователи предикатов - являются эквивалентными преобразованиями. Эквивалентными, так как в процессе решения может произойти подмена одной задачи другой . Предикат $P_0(X,Y)$ является сильнейшим постусловием. Сильнейшим, потому что только импликация $P_0(\sigma_{i_1,i_2,\ldots,i_n}(X,Y)) \Rightarrow P(X,Y)$ гарантирует, что $f(X)$ является искомым решением. Общее решение предиката $P(X,Y)$ в виде абстрактных операторов (условного и цикла) подчеркивает фундаментальное значение этих операторов для любого языка программирования.

Работы Дейкстры [7] и Гриса [5], и более ранние работы Хоора определяют так называемый аксиоматический подход к описанию семантики языков программирования и на его основе - специфические методы программирования сверху-вниз [1]. Принципиальная ошибка всех этих работ связана прежде всего с тем, что в рамках этого подхода происходит подмена понятия "семантика языка" понятием "семантика алгоритма". Описать семантику языка значит описать процесс выполнения каждой конструкции этого языка. Описание семантики алгоритма складывается из описания семантики задачи (что дано и что получить) и описания того, как решается эта задача, т.е. из описания метода решения, сущность которого определяется выбором системы преобразований f_i . Ясно, что семантика алгоритма тесно связана с семантикой языка. Более того, язык часто навязывает метод решения задачи. Именно эта тесная связь и является причиной путаницы, причиной того, что развиваемая в этих работах теория "стоит на голове": исследование ведется не от понятия задачи через язык к программе, а от конструкций языка, на которые по непонятным причинам следует навешивать неизвестно откуда взятые предикаты.

Другая принципиальная ошибка этого подхода связана со слепой верой в то, что каждому вычислительному процессу, записанному в виде программы на реальном языке программирования, можно адекватным образом сопоставить вывод на языке логики предикатов первого порядка. Язык логики предикатов по мощности эквивалентен языку машин Тьюринга, и писать реальные программы на любом из этих языков не более практично, чем обрабатывать крестьянское поле булавкой. Бурбаки [4] проводят доказательство первых теорем теории множеств на языке математической логики, но, понимая всю бесперспективность такого начинания, очень быстро переходит на естественный язык. А ведь реальная программа гораздо сложнее многих теорем теории множеств. Отсюда следует, что язык описания задачи должен превосходить по мощности язык программирования, а доказательство правильности программ следует проводить на естественном языке.

В задачах численного анализа исходный предикат $P(X,Y)$ задается в виде уравнения или системы уравнений с теми или иными ограничениями на исходные данные или на искомую функцию $f(X)$. Предикат $Q(X)$ (если он не зависит от Z) задает область определения функции $f(X)$. Преобразования $f_i(X,Y)$ являются либо эквивалентными преобразованиями исходной системы уравнений к некоторому каноническому виду, решение которой очевидно, либо определяют очередной шаг в последовательности приближений к искомому результату. Предикат $P_0(X,Y)$ задает либо каноническую форму уравнений, достижение которой является конечной целью преобразований, либо точность вычисления. Таким образом, на классе задач численного анализа общая схема перехода по крайней мере в принципе оказывается достаточно простой и универсальной.

В классе задач обработки символьной информации предикат $P(X,Y)$

задается при помощи системы словарных уравнений, частным случаем которой является операция отождествления. Примеры задач символьной обработки можно найти в работе [II]. Постановка некоторых задач (так называемые лабиринтные задачи) не содержит предиката $P(X,Y)$. В этом классе задач задаются преобразования f_i, начальное состояние и предикат $P_0(X,Y)$. Требуется найти последовательность $f_{i_1}, f_{i_2}, \ldots, f_{i_k}$, которая преобразует начальное состояние в состояние, удовлетворяющее предикату $P_0(X,Y)$. Примеры задач и этого класса можно найти в работе [II].

Перейдем к рассмотрению конкретных примеров. Первые четыре взяты из работы Гриса [5].

I. Даны произвольные числа x, y. Найти $\max(x,y)$.

Предикат $P(x,y) = \exists z(z \geq x \,\&\, z \geq y \,\&\, (z = x \vee z = y))$. Предикат $P_0(x,y)$ совпадает с $P(x,y)$, поэтому

$f_0(x) =$ ЕСЛИ $y < x \to x \sqcup x \leq y \to y$ КЕ

Доказательство правильности полученного решения проведем методом разбора различных случаев.

Пусть $y < x$. Тогда $f_0(x) = x$. (Здесь используется знание семантики условного оператора.) Подставляем $f_0(x)$ в предикат $P(x,y)$ вместо z:

$x \geq x \,\&\, x \geq y \,\&\, (x = x \vee x = y) \equiv$ true.

Пусть $x \leq y$. Тогда $f(x) = y$.

$y \geq x \,\&\, y \geq y \,\&\, (y = x \vee y = y) \equiv$ true.

2. Пусть j и k ($k > 0$) удовлетворяют равенству $j = k \bmod 10$. Требуется увеличить k, сохранив равенство.

Формальное описание предиката имеет вид

$R(k,j) = \exists z\,(k = 10 * z + j) \,\&\, k > 0 \,\&\, 0 \leq j \leq 9$,

$P(k,j) = R(k,j) \,\&\, \exists x,y(R(x,y) \,\&\, x = k + 1)$.

После элементарных преобразований предикат $P(k,j)$ принимает форму предиката P_0:

$P(k,j) = \exists x,y(x = k + 1 \,\&\, (j < 9 \,\&\, y = j + 1 \vee j = 9 \,\&\, y = 0))$,

поэтому получаем

$f(k,j) =$ (k + 1, ЕСЛИ $j < 9 \to j + 1 \sqcup$
$j = 9 \to 0$
КЕ)

Доказательство правильности подстановкой $f(k,j)$ в $P(k,j)$ аналогично доказательству в первом примере.

3. Суммирование элементов массива $b[1 : n]$. Это типичный пример, когда используется схема полного перебора по i $(1 \leq i \leq n)$:

$P(b) = \sum_{i=1}^{n} b[i]$; $Q(i,s) \equiv 1 \leq i \leq n$;

$f_1(i,s) = (i+1,\ s+b[i+1])$; $Q(f_1(i,s)) \equiv 0 \leq i < n$.

Начальное состояние в задачах перебора содержит нижнюю границу индекса перебора и значение искомой функции на пустом множестве. Так как сумма элементов пустого массива по определению равна нулю, то начальное состояние в данной задаче – (0,0). Поэтому получаем программу

(0,0) ЦИКЛ $i < n \to (i + 1,\ s + b[i + 1])$ КЦ

или на конкретном языке программирования (например, на языке Алгол-68):

```
i := 0 ; s := 0; while i<n do i := i+1; s := s+b[i] od
```

4. Поиск элемента в двумерном массиве $b[1 : m, 1 : n]$.

$P(b,i,j,x) \equiv b[i,j] = x$

$f_1(i,j) =$ ЕСЛИ $j < n \to (i,j+1)$ $\sqcup$
$i < m \to (i+1,0)$
КЕ

Функция f_1 задает порядок перебора по двум индексам. Искомая программа имеет вид

(1,1) ЦИКЛ $\neg P \to f_1(i,j)$ КЦ

5. Дано произвольное слово x в алфавите $\{a,b,c\}$. Требуется построить слово y, которое получается из слова x после выбрасывания из него всех символов a.

Это типичная лабиринтная задача при $z_0 = x$ описывается так:

$Q(x,z) = (z = x)$

$f_1(y_1 a y_2) = y_1 y_2$

$P_0(x) = \neg \exists z_1 z_2 (z_1 a z_2 = x)$

Преобразование f_1 в общем случае недетерминировано.

Искомая программа

(x) ЦИКЛ $\neg P_0(z) \to f_1(z)$ КЦ

Эта же задача, запрограммированная в функциональном стиле (на языке Форт):

```
: a   ; IMMEDIATE
: b C"b  C,  ; IMMEDIATE
: c C"c  C,  ; IMMEDIATE
```

Символы входной строки должны быть разделены пробелами.

6. Дана КС-грамматика:

```
E → E + T | T
T → T * F | F
F → (E) | I
I → a | b | c
```

описывающая фрагмент арифметических выражений. Требуется построить перевод арифметических выражений в польскую инверсную запись:

$Q(x) = y_1$ "E +T" $y_2 = x \vee y_1$"T * F" $y_2 = x \vee y_1$"(E)" $y_2 =$
$x \vee y_1$"I" $y_2 = x$;

$f_1(x) = y_1$"E + T" $y_2 = x \to y_1$ "E" "T" + y_2 ;

$f_2(x) = y_1$"T * F" $y_2 = x \to y_1$ "T" "F" * y_2 ;

$f_3(x) = y_1$ "(E)" $y_2 = x \to y_1$ "E" y_2 ;

$f_4(x) = y_1$"I" $y_2 = x \to y_1$ I y_2 ;

Программа, решающая задачу перевода:

ЦИКЛ $f_1(x) \sqsupset f_2(x) \sqsupset f_3(x) \sqsupset f_4(x)$ КЦ

На вход этой программы подается строка "x", где x - исходное арифметическое выражение.

Эта же задача, запрограммированная в функциональном стиле (на языке Форт):

```
VARIABLE a VARIABLE b VARIABLE c
: + C" +  OVER  = 0 ER C" *  = OR
    IF C, [ LATEST NAME> , ]  RDROP
    THEN C" + ;  IMMEDIATE
: * C" * OVER  = IF C, THEN C" *  ;  IMMEDIATE
: ( C" ( ;  IMMEDIATE
: ) DUP C" (  = IFNOT C, [ LATEST NAME> , ]  RDROP
                THEN DROP  ;  IMMEDIATE
: 1 + DROP DROP  ;  IMMEDIATE
```

Следует внимательно проанализировать примеры 5 и 6 и правильно понять различие между программами, записанными в виде абстрактных циклов, и соответствующими программами на языке Форт. Если исходные данные имеют элементарную структуру (как в примерах I - 4), то абстрактный цикл легко преобразуется в программу на реальном языке программирования. Если исходные данные не элементарны, то переход от абстрактного цикла к реальной программе не менее сложен, чем переход к ней от исходной формулировки задачи. Преобразование исходного текста в активный процесс позволяет найти кратчайший путь от постановки задачи к реальной программе.

Последние два примера являются первыми аргументами в обосновании общего вывода, что язык представления знаний должен поддерживать функциональные методы программирования, потому что они позволяют найти кратчайший путь от постановки задачи к решающей ее программе на реальном языке программирования.

§3. Алмаз - абстрактный язык представления знаний

Язык Алмаз (АЛгоритмический Метаязык Активных Знаний) является функциональным языком общего назначения. Основным типом данных этого языка является функция. Этот тип данных в языке Алмаз играет такую же роль, подчиняя себе все остальные типы данных, как список в языке Лисп, строка в языке Скобол или реляционная таблица в реляционной базе данных. Метаязыковые возможности языка Алмаз обеспечиваются набором средств, достаточных для описания современных языков программирования. По своей сущности этот язык является конкретизацией математической модели языка [II]. Несмотря на определенную конкретизацию он остается абстрактным языком (некоторые его конструкции не допускают эффективной реализации), предназначенным для того, чтобы служить ориентиром при построении конкретных языков представления знаний.

В том виде, как он здесь описан, язык Алмаз может использоваться как язык проектирования сложных программ. Однако процесс перехода от абстрактного языка к реальному языку программирования (например, к таким языкам, как ПЛ/I, Алгол-68, Паскаль, Ада) в общем случае остается очень сложным. Система обработки информации, описанная в следующей главе, предназначена в частности для того, чтобы обеспечить реальную возможность перехода от программы на языке Алмаз к программе на языке Форт.

Программа на языке Алмаз представляет собой набор описаний функций, макрофункций, модулей, интерпретаторов, грамматик и управляющих систем. Обращение к одной из описанных функций инициирует выполнение программы. Описания одних функций (модулей и т.д.) содержат список формальных параметров, описания других - не со-

держат. Функции, не содержащие формальных параметров, аналогичны функциям языка Форт: входные данные такие функции берут из стека. В отличие от Форта количество стеков здесь не ограничено. Переход на i-й стек осуществляется оператором [i , возврат на предыдущий стек - оператором i] . Тело функции представляет собой суперпозицию функций в польской инверсной записи.

При работе программа на языке Алмаз может находиться либо в режиме выполнения, либо в режиме компиляции. (В языке Форт эти режимы существуют лишь при работе текстового интерпретатора.) Функция *EXEC переводит работу программы в режим выполнения, а функция .COMP - в режим компиляции.

Управляющие операторы. Основу структуры управления языка Алмаз составляют обращения к функциям и модулям. Обращение к функции в общем случае формируется динамически. Это позволяет исключить использование не только операторов перехода, но и условных операторов и циклов. Однако замена управляющих операторов требует введения новых имен функций и часто делает описание функции слишком мелким. Поэтому в языке Алмаз оставлены и условные операторы и циклы.

Детерминированный условный оператор имеет вид

ЕСЛИ f1 → g1 | f2 → g2 | ... | fN → gN КЕ

где fi, gi - суперпозиции функций. Если суперпозиция f1 определена, то выполняется суперпозиция g1 , и ее результат есть результат условного оператора, иначе (если f2) определена, то выполняется g2 и т.д. Если ни одна из функций fi не определена, то значение условного оператора не определено.

Недетерминированный условный оператор имеет вид

ЕСЛИ f1 → g1 □ f2 → g2 □ ... □ fN → gN КЕ

Если суперпозиция f1 определена, то выполняется суперпозиция g1. Если g1 определена, то ее результат является результатом условного оператора. Иначе, т.е. если f1 не определена или f1 определена, а g1 не определена, то выполняется вторая альтернатива f2 → g2 и т.д.

Параллельный условный оператор имеет вид

ЕСЛИ f1 → g1, f2 → g2, ..., fN → gN КЕ

В этом операторе все альтернативы запускаются на выполнение одновременно.

Операторы цикла имеют вид

ЦИКЛ f1 → g1 | f2 → g2 | ... | fN → gN КЦ

ЦИКЛ f1 → g1 □ f2 → g2 □ ... □ fN → gN КЦ

ЦИКЛ f1 → g1, f2 → g2, ..., fN → gN КЦ

Цикл рассматривается как итерация условного оператора. Итерация выполняется пока соответствующий условный оператор определен.

Определение функций. Выполнение описания функции вводит ее новое определение. Заголовок описания содержит различные префиксы, которые определяют различные классы функций.

1) Пустой префикс. Описание функции имеет вид

: F [* X1 X2 ... XM *] F1 F2 ... FN ;

(при M=0 скобки [* и *] могут быть опущены.)

Описание функции аналогично оператору присваивания: тело функции присваивается в качестве значения имени функции. В процессе определения функции ее тело может быть выполнено. Таким образом, описание является действующим оператором.

2) Префикс COMP-Описание функции F имеет вид

COMP : F [* X1 X2 ... XM *] F1 F2 ... FN ;

Префикс COMP означает, что при обращении к функции F ее тело будет компилироваться целиком независимо от режима выполнения.Однако при обработке описания функции F оператор *EXEC переводит режим компиляции в режим выполнения.

Описание функции с префиксом COMP не эквивалентно вызову функции с этим же префиксом.

3) Префикс EXEC.

EXEC : F [* X1 X2 ... XN *] F1 F2 ... FN ;

Наличие этого префикса в описании функции F означает, что функция F будет выполняться и в режиме компиляции. Вызов COMP.F в любом режиме будет компилировать тело функции F.

4) Префикс ALLOT.

ALLOT : F [* X1 X2 ... XM *] F1 F2 ... FN ;

Префикс ALLOT вводит новое определение функции GF не уничтожая ранее выполненных определений этой функции, а лишь временно закрывая к ним доступ. Выполнение оператора FREE F вновь открывает доступ к предыдущему определению. Таким образом, описание этого типа аналогично оператору ALLOCATE языка ПЛ/I.

5) Префикс NDET.

< число> NDET : F [* X1X2 ... XM *] F1 F2 ... FN ;

Этот префикс вводит описание недетерминированной функции. Перед описанием находится целое неотрицательное число. Все описания недетерминированной функции располагаются в порядке возрастания положительных чисел. Описания, содержащие одинаковые числа, располагаются в порядке их выполнения. Описание, которому предшествует число $\emptyset$, располагается последним (динамически). При обращении к функции сначала выполняется первое описание, при возврате - второе и т.д. Если вызов функции имеет вид .ND N F, то выполнение функции F начинается с первого описания, которому предшествовало число N .

Пример

: СМЕРТЕН ЧЕЛОВЕК ;
0 NDET : ЧЕЛОВЕК "ТЬЮРИНГ" ; 0 NDET: ЧЕЛОВЕК "СОКРАТ" ;
: ГРЕК "СОКРАТ" ;
: ? ЕСЛИ РАВНО ¬ \ НЕУСПЕХ ЖЕ ;

Последовательность действий: СМЕРТЕН ГРЕК ? вычисляет, какой грек был смертен.

6) Префикс PAR :

< число> PAR : F [* X1 X2 ... XN *] F1 F2 ... FN ;

С помощью этого префикса списывается набор параллельных функций, имеющих одно имя. При обращении к функции F одновременно запускаются на выполнение все описания с именем F . Каждое описание функции F инициирует параллельный процесс. Число в префиксе позволяет идентифицировать некоторые описания в наборе с одним именем. Обращение i * i F запускает на выполнение лишь те описания функции F , префикс которых содержит число i .

Синхронизация параллельных процессов осуществляется следующим образом. Если в одном из параллельных процессов произошло обращение к неописанной функции F , то имя этой функции засылается в управляющую память, а выполнение процесса приостанавливается. Если к этому моменту в управляющей памяти находятся имена H1,... ...,HK , то имя F конкатенируется с этими именами и если какая-либо комбинация имен Hj и F образует имя X описанной функции, то функция X выполняется, и после ее выполнения (или приостановки) возобновляют работу все те процессы, которые были приостановлены обращением к функциям, имена которых участвовали в

образование имени X , в противном случае - имя F остается в управляющей памяти и ни один из приостановленных процессов не возобновляет работу. Запустить некоторые из них на выполнение может лишь какой-то работающий процесс, обратившись к неописанной функции. Если все процессы приостановлены, то происходит обращение к функции НЕОПР. Эта функция может быть описана, и тогда она определит дальнейший процесс вычислений. Если функция НЕОПР не описана, то происходит прерывание с признаком "неопр".

Имя неописанной функции автоматически попадает в управляющую память и приостанавливает процесс выполнения. Однако в управляющую память можно занести любое слово, не останавливая процесс. Это можно сделать при помощи операции ЖДАТЬ. Она заносит в управляющую память слово или набор слов, представленных в виде списка.

Кроме операции ЖДАТЬ с управляющей памятью связаны операции:

ЖДАТЬ? F - кладет на стек число экземпляров имени F , находящихся в управляющей памяти;

¬ЖДАТЬ F - убирает из управляющей памяти один экземпляр имени F и запускает на выполнение тот процесс, который был остановлен обращением к функции F . Если с F связано несколько остановленных процессов, то запускается на выполнение процесс, остановленный раньше других;

!ЖДАТЬ - кладет на стек список имен функций, находящихся в управляющей памяти;

¬!ЖДАТЬ - очищает управляющую память.

7) Префикс CONTROL .

CONTROL : F [* X1 X2 ... XN *] F1 F2 ... FN ;

Этот префикс вводит особый тип функций - управляющие функции. Каждая управляющая функция должна быть описана в одном из модулей. Отличие управляющей функции от других функций заключается в том, что после каждого ее изменения или переопределения управление передается модулю, в котором она определена: каждый модуль содержит кроме описаний функций последовательность действий, которая выполняется при передаче ему управления.

Вызов функций. Описание функции может содержать формальные параметры X1,X2,...,XM. Если функция описана как функция с па-

раметрами, то при обращении к ней можно использовать любую из двух форм вызова: либо <префикс> F , либо <префикс> F(G1,G2,... ...,GM), где Gi - фактические параметры. Их число должно совпадать с числом формальных параметров. При первой форме обращения фактические параметры берутся из стека.

Оператор вызова функции обязательно содержит имя функции и кроме него может содержать префикс и параметры. Этот оператор выполняет ряд действий, связанных с передачей фактических параметров, и в зависимости от префикса определяет режим выполнения тела функции. Простейший вызов F , не содержащий ни префикса, ни параметров, аналогичен оператору перехода с возвратом. Тело функции F не дублируется. Если функция F описана как параллельная, то выход из нее осуществляется после выполнения всех параллельных процессов. Приостановленный процесс считается выполненным, пока работает хотя бы один параллельный процесс - иначе произойдет прерывание с передачей управления на функцию НЕУСПЕХ. Если функция F описана как недетерминированная, то выход из нее осуществляется лишь в случае ее успешного выполнения. При неуспешном выполнении управление передается функции НЕУСПЕХ. Если функция НЕУСПЕХ не описана, то она либо осуществляет возврат, либо прерывает процесс выполнения. Однако если это прерывание произошло при вычислении условного выражения в условном операторе или цикле, то управление передается на выполнение следующей альтернативы.

Вызов COMP. F в начало тела функции F как бы добавляет оператор .COMP, переводящий любой режим в режим компиляции, и передает модифицированной функции F управление. При определении новых функций этот вызов позволяет заменить обращение к какой-либо функции на ее тело. Например, выполнение описаний: G A B C ; : F *EXEC COMP.G .COMP M ; определит функцию F так же, как и описание : F A B C M ; .

Вызов EXEC* F независимо от любого стандартного режима всегда инициирует выполнение функции F . Лишь нестандартная интерпретация может перевести вызов этого типа в режим компиляции. Этот вызов можно сравнить с операторами периода макрогенерации или препроцессирования: все такие операторы как бы следуют за словом EXEC*

Вызов ALLOC* F копирует тело функции F в активный стек (стек выполнения рекурсивных функций) и после передачи фактических параметров передает ему управление. Таким образом, этот вызов аналогичен вызову нереентрабельной функции.

Вызов NDET* F копирует всю оперативную память и инициирует выполнение функции F . При ее успешном выполнении этот вызов эквивалентен вызову без префикса. При неуспешном выполнении происходит восстановление памяти: память переходит в то состояние, которое было до обращения к функции F . Однако управляющая память не восстанавливается: неуспешное выполнение функции F можно использовать для изменения управляющей памяти. Вызов недетерминированной функции не копирует память, и на программиста возлагается забота о восстановлении значений глобальных переменных при возвратах. При вызове NDET* F ситуация противоположная. Память восстанавливается автоматически, но издержки памяти и времени могут быть значительными. Следовательно, вызовы функции с префиксом NDET* и недетерминированные функции являются взаимодополняющими средствами языка. Если процесс почти детерминированный, число ветвлений невелико, но каждая ветвь требует значительного объема вычислений, то все функции следует описывать как детерминированные, а для ветвлений использовать вызов с префиксом NDET*. При большом количестве возвратов следует использовать недетерминированные функции.

Аналогичную роль играет оператор ветвления (NDET F1 F2...FN). Вначале память копируется и управление передается функции F1 . При ее успешном выполнении работа оператора заканчивается, память не восстанавливается. При неуспешном выполнении функции F1 память восстанавливается и инициируется выполнение функции F2 и т.д. Если ни одна из функций Fi ($1 \le i \le N$) не завершает своего выполнения успешно, то выполнение оператора ветвления эквивалентно пустому оператору.

Вызов PAR* F запускает на параллельное с вызвавшей программой выполнение функцию F . В отличие от параллельных функций вызов PAR* F создает более независимый процесс F: все порождаемые в F описания функций и модулей локализованы в нем и не доступны параллельно работающим процессам. Аналогично, оператор (PAR F1 F2... ... FN) создает N независимых (в смысле локализации) параллельных

процессов. Вызов CONTROL* F вводит контроль за изменением уже определенных функций: в процессе выполнения функции F на экран дисплея или на печать будет выдаваться информация об изменении или переопределении любой уже определенной функции.

Макрофункции. Понятие макрофункции аналогично понятию макроса, которое, как известно, является столь же старым, как и понятие ассемблера. Вызов макроса отличается от вызова подпрограммы. Такое же различие между вызовами макрофункции и функции. Однако в функциональных языках механизм макросов играет менее важную роль, чем в компилируемых языках. Например, введение макрофункций в язык Форт вряд ли существенно расширило бы его возможности. Тем не менее, механизм макросов, являясь средством объединения принципов параметризации и активизации, дает возможность строить простые и лаконичные параметризованные описания функций без потери эффективности вычислений.

Процесс активизации разбивает программируемый алгоритм на сколь угодно мелкие составные части. При этом возникает множество однотипных функций. Прямая параметризация таких множеств с помощью обычных функций как правило приводит к их интерпретации во время выполнения и, тем самым, – к потери эффективности. Макрофункция выполняется один раз, именно в этом и состоит ее преимущество перед обычной функцией. Механизм макрофункций является наиболее простым и эффективным частным случаем преобразования пассивной строки в активную суперпозицию функций.

Все макрофункции в языке Алмаз делятся на две группы. Описания макрофункции первой группы содержат списки формальных параметров:

MACRO: F [* X1 X2 ... XN*] <строка символов > ;

В описаниях макрофункций второй группы список формальных параметров отсутствует. Отсутствие явно указанных формальных параметров не означает, что они не входят в тело макрофункции. Каждое вхождение параметра в тело макрофункции должно иметь вид %i., где i – целое неотрицательное число, являющееся номером формального параметра. Параметр с нулевым номером обозначает имя макрофункции. Формальными параметрами макрофункций первой группы являются произвольные слова, не содержащие пробелов. Их вхождения в тело

макрофункции также слева и справа ограничены символами: % и точка . Например:

MACRO: X [* S *] " : %S. : %S. 15 ; ""%S."";" ;

является описанием макрофункции X , выполнение которой порождает описания новых функций. При обращении "H" X будет порождена функция H , имеющая описание : H : H 15 ;"Ч";. При первом обращении к функции H она будет переопределена: : H 15;, а в стек будет заслан адрес идентификатора H . При последующих обращениях к функции H в стек будет засылаться число 15.

Макрофункции с явно описанными параметрами являются менее гибким средством обработки текстовой информации, чем макрофункции без параметров. Поэтому макрофункции с параметрами следует использовать как средство параметризации описаний функций и режимов выполнения, макрофункции без параметров – как средство обработки текстов. Макрофункции с параметрами более эффективны по быстродействию и более чувствительны к ошибкам, т.е. обладают большей надежностью.

Макрофункция с параметрами может быть перекомпилирована в макрофункцию без параметров:

MACRO: F #EXEC F(%1. , %2., ..., %X.) ;

Если необходим контроль за соответствием фактических и формальных параметров, то макрофункцию без параметров можно перекомпилировать в макрофункцию с параметрами:

MACRO: P [* 1 2 ... X *] #EXEC F(%1., %2., ...,%X.) ;

Тело макрофункции является строкой, поэтому к телам макрофункций применимы все операции над строками. Кроме этого, в языке определена еще одна операция MACRO, предназначенная для преобразования произвольной строки в тело макрофункции:

"S" MACRO N F

Если F – имя строки или макрофункции, то операция MACRO последовательно заменяет каждое вхождение строки S в F на параметр %N.. Вызов этой операции эквивалентен нормальному алгоритму с одной подстановкой: S → %N. , на вход которому подается строка или тело макрофункции F .

Модули. Понятие модуля аналогично понятию открытой подпрограммы. Его описание имеет вид

<префикс> MOD:N [*X1...XN*]<набор описаний> <суперпозиция функций>;

В набор описаний входят описания функций и модулей. Хотя тело модуля может содержать произвольную последовательность действий, а совокупность описаний может быть и пустой, тем не менее главное предназначение модуля - быть хранилищем множества описаний. Вызов модуля открывает доступ к хранимым в нем описаниям функций и других модулей, освобождение (FREE M) или закрытие модуля (CLOSE M) закрывает доступ. Закрытие модуля создает новый экземпляр описания модуля с тем же именем, если при обращении к нему его тело копировалось. Старое описание сохраняется. Функции, к которым был открыт доступ, могут меняться в процессе вычисления и при закрытии модуля его новый экземпляр будет содержать модифицированные описания функций. Вместо закрытия можно освободить модуль. В этом случае новый экземпляр модуля не создается. Освобождение модуля выбрасывает последнее описание и открывает доступ к предыдущему.

В отличие от функции имя модуля может быть пустым словом. Если описание модуля не содержит имени, то после его выполнения его адрес остается в стеке. Такие модули можно использовать аналогично блокам компилируемых языков, локализующим имена описанных в них объектов. Но модуль без имени является понятием более широким, чем блок, так как он позволяет какие-то объекты сделать локальными, а какие-то - глобальными. Те описания функций в теле модуля, которые были обработаны в режиме компиляции, остаются в сгенерированном описании модуля, и последовательность действий DUP .FREE EXEC сделает только их глобальными. Операция EXEC сделает все описания, содержащиеся в модуле, глобальными. Операция .FREE закроет доступ ко всем описаниям - и лишь в этом случае модуль без имени идентичен блоку.

Описание модуля, также как и описание функции, может содержать префикс.

1) Пустой префикс. Понятие модуля, описание которого не содержит префикса, аналогично понятию словаря языка Форт. Такие модули в основном предназначены для хранения совокупности описаний функций и других модулей, которые могут быть одновременно вызваны и одновременно освобождены. Единственное отличие от словаря - при вызове модуля выполняется последовательность действий (которая

может быть и пустой), записанная в нем в виде суперпозиции функций.

2) Префикс COMP. Этот префикс в заголовке описания модуля M указывает на то, что при обращении к модулю M его тело всегда будет компилироваться. Модули с таким префиксом, как правило, служат средством задания шаблонов памяти и действий. Другими словами: модули с префиксом COMP - это определяющие конструкции, частными случаями которых являются абстрактные типы данных, записи (структуры) с вариантами, конструкция CREATE - DOES > языка Форт, понятие класса в языке Симула-67, понятие объекта в языке Смолток и т.п.

3) Префикс EXEC . Наличие этого префикса в описании модуля означает, что модуль и все описанные в нем функции и модули всегда выполняются независимо от режима выполнения. Только управляемая интерпретация может остановить их выполнение. Модули с этим префиксом незаменимы тогда, когда выполняемая часть программы описывается гораздо легче, чем ее компилируемая часть. Нередко при активизации исходной информации ее компилируемую часть либо невозможно описать, либо описание получается слишком громоздким.

4) Префикс ALLOT . Наличие этого префикса в описании модуля означает, что старое описание (если оно было) сохраняется, но становится недоступным для использования. Освобождение модуля вновь открывает доступ к старому описанию. Таким образом, этот префикс по отношению к модулю действует так же, как и по отношению к функции. Вызов функции, описанной с этим префиксом копирует ее тело в активный стек, при вызове модуля его тело копируется в область доступных для использования функций и модулей, открывая доступ ко всем описанным в нем функциям и модулям. В этом случае при закрытии модуля создается экземпляр его нового описания. Старое описание сохраняется, но до освобождения нового экземпляра остается недоступным для использования. Новый экземпляр может отличаться от старого как набором функций и модулей (так как до создания нового экземпляра некоторые из них могут быть освобождены, могут быть порождены новые описания функций с префиксом ALLOT), так и тем, что описания функций и модулей могут изменяться в процессе вычисления. Основное назначение модулей с префиксом ALLOT - это создание достаточно эффективных недетерминированных процессов

5) Префикс NDET . Этот префикс вводит описание недетерминированного модуля. Способ выполнения такого модуля аналогичен способу выполнения недетерминированной функции. Вызов определяет точку возврата. При возврате вызывается следующий экземпляр описания модуля с тем же именем.

6) Префикс PAR . Этот префикс в заголовке модуля определяет его как параллельный. Вызов такого модуля, как и вызов функции, инициирует множество параллельных процессов. Все модули этого типа глобальны, т.е. все функции, содержащиеся в них, доступны во всех параллельных процессах. Оператор FREE M освобождает лишь один экземпляр модуля M - динамически последний. Освободить все экземпляры модуля можно при помощи цикла: ЦИКЛ FREE M КЦ. В общем случае параллельные процессы нельзя промоделировать на однопроцессорной машине - разве лишь мельчайшим квантованием времени. Это объясняется тем, что любой параллельный процесс может вмешаться в работу открытого (незащищенного) процесса в непредсказуемые моменты времени и существенно изменить его работу. Обычным приемом такого вмешательства является вызов глобальных параллельных модулей, которые могут прикрыть (сделать недоступными) все работающие функции, или освободить какие-то модули, чем можно немедленно остановить любой незащищенный процесс или, наконец, просто изменить некоторые глобальные функции.

7) Префикс CONTROL . Изменение или переопределение любой функции или модуля, содержащихся в теле модуля M , описанного с префиксом CONTROL , вызывает передачу управления модулю M .

Управляемая интерпретация. Выполнение программы осуществляется стандартным интерпретатором, который организует обращение к функции или модулю и возврат после их выполнения. Можно считать, что стандартный интерпретатор выполняет одну операцию EVAL - выполнить. Имя функции (ее адрес компиляции) поступает в стек, после чего выполняется операция EVAL. Поэтому если стандартному интерпретатору дать имя SI , то он может быть описан в виде

```
INTERPRET: SI EVAL ;
```

Язык Алмаз содержит средства динамического управления процессом интерпретации: стандартный интерпретатор может быть заменен другим интерпретатором, который должен быть описан в программе либо как интерпретатор с параметрами:

< префикс> INTERPRET: I [* F1 F2...FM *]<суперпозиция функций> ;

либо без параметров:

<префикс> INTERPRET: I <суперпозиция функций >;

Параметрами интерпретатора могут быть имена функций, модулей или интерпретаторов.

Если интерпретатор имеет параметры, то обращение к нему происходит в момент обращения к одному из его параметров: имя F1 остается на стеке, а действия интерпретатора, описанные в нем в виде суперпозиции функций, выполняются в режиме стандартного интерпретатора. Если имя не входит в список работающего в данный момент интерпретатора, то оно не выполняется, а компилируется. Если интерпретатор не имеет параметров, то обращение к нему происходит при каждом обращении к любой функции (или модулю). Стандартный интерпретатор является интерпретатором без параметров.

Вызов интерпретатора I1 закрывает доступ к работающему в данный момент интерпретатору I2 . Оператор FREE I1 вновь открывает доступ к интерпретатору I2 . В качестве примера рассмотрим интерпретатор F :

```
INTERPRET: F [* F *] .FREE ;
```

После вызова интерпретатора F все функции будут компилироваться. Второе обращение к F передаст управление ранее работавшему интерпретатору. Если это был стандартный интерпретатор, то будет возобновлено стандартное выполнение программы.

Описание интерпретатора может содержать префикс. Этот префикс переносится на каждое обращение без префикса к описанной функции или модулю. На числа, строки, списки и другие данные префикс не переносится. Не переносится он и на стандартные операции и на те обращения, которые уже имеют префикс.

Управляемая интерпретация является мощным средством программирования. Она позволяет менять режим выполнения программы в широком диапазоне - от полной интерпретации (в режиме отладки, трассировки, частичного выполнения и т.п.) до полной компиляции. Механизм управляемой интерпретации перекрывает возможности препроцессоров, макрогенераторов и языков, в которых реализованы

режимы компиляции-выполнения. В то же время реализация управляемой интерпретации не сложнее реализации обычных функций.

Грамматики. Грамматики служат средством перевода с одного языка на другой, средством интерпретации, активизации и внешней обработки произвольных текстов. При интерпретации исходный текст рассматривается как программа на языке Алмаз и выполняется. Активизация исходного текста совпадает с его переводом на язык Алмаз. Внешняя обработка выполняется с помощью операции отождествления, для определения которой необходимо задать грамматику. Перевод с языка L1 на язык L2 является специфическим частным случаем интерпретации текста на языке L1 при помощи конструкций языка L2. С этой точки зрения грамматика является посредником между языком Алмаз и другими языками. Описать грамматику языка L значит задать информацию, необходимую и достаточную для толкования текста на языке L.

Теоретической основой класса грамматик, используемых в языке Алмаз, является класс функциональных грамматик. Представление функциональной грамматики в языке Алмаз в виде набора специфических модулей определяет новую конструкцию языка. Она является некоторым обобщением понятия модуля, более точным названием которого могло бы служить выражение "грамматический модуль" - модуль, предназначенный для представления функциональной грамматики. Поэтому термин "грамматика" является сокращением выражения "грамматический модуль". Выполнение описания грамматики отображает ее в обычный модуль.

Описание грамматики имеет вид

GRAMMAR: G [* P1 P2...PM *]<описание функций> < правила> ;

Каждое правило имеет вид :P P1 P2 ... PK ; . Все символы грамматики представлены в виде слов и делятся на четыре класса:терминальные, нетерминальные, функциональные и прочие (или идентификаторы). Последнее название объясняется тем, что при описании реальных языков программирования "прочими" оказываются идентификаторы. Список терминальных символов грамматики представлен в виде функции TERM , список нетерминальных символов - в виде функций NTERM . Символ является функциональным, если он является именем функции, описанной в разделе "описания функций". Начальным

символом грамматики является первый символ функции NTERM . Все прочие символы имеют одинаковое описание, которое задается функцией OTHER : символ поступает в стек, после чего выполняется функция OTHER . Функции TERM , NTERM и OTHERдолжны быть описаны в разделе < описания функций >.

Выполнение описания грамматики заключается в построении модуля M , содержащего функции, необходимые для синтаксического анализа. Вначале для каждого правила: $A \to \Psi$;определяются множества FIRST(A) терминалов, каждый из которых удовлетворяет условию: если $xA\Psi$ - сентенциальная форма и из $xA\Psi$ выводима цепочка xay (x, y - терминальные цепочки), то $a \in$ FIRST(A) . Затем если $\Psi = a\xi$, то строится функция: Aa ξ ; если $\Psi = B\xi$, где B - нетерминал, то строится функция: Aa Be ξ ; , если Ψ пусто, то функция: Aa ЖДАТЬ a; . Если в результате построения функций появилось несколько описаний с одним именем Aa , то к таким описаниям добавляется префикс NDET . Модуль M содержит все функции, описанные в грамматике, и вновь построенные функции.

Грамматика после выполнения ее описания превращается в модуль, поэтому префиксы COMP, EXEC, ALLOT и CONTROLв заголовке ее описания имеют тот же смысл, что и для модуля. Префикс PAR в описании грамматики G означает, что после ее вызова разбор исходного предложения проводится сразу во всех грамматиках, имеющих имя G . Построенные после разбора суперпозиции выполняются параллельно. Префикс NDET в описании грамматики имеет существенно другой смысл, чем в описании модуля. Этот префикс означает, что исходная грамматика недетерминированная, т.е. по исходному предложению может быть построено несколько соответствующих ему суперпозиций. После разбора все эти суперпозиции выполняются параллельно.

Операция отождествления. Эта операция реализует один из частных видов систем словарных уравнений. Пусть $y_1, y_2, \ldots, y_n$ - произвольные множества слов в некотором алфавите Σ . Система уравнений имеет вид

$$\varphi_1 = \psi_1,$$
$$\varphi_2 = \psi_2,$$
$$\ldots\ldots$$
$$\varphi_m = \psi_m,$$

где φ_i, ψ_i $(1 \le i \le m)$ - слова в алфавите $\Sigma \cup \{x_j\} \cup \{z_p^k\}$ $(1 \le j \le J,\ 1 \le p \le n,\ 1 \le k \le K)$, x_j - параметры системы, z_p^k - неизвестные. При заданном наборе слов $(x_1^o, x_2^o, \ldots, x_J^o)$ требуется найти решение системы, т.е. множество слов z_p^k, таких, что $z_p^k \in y_p$. Если $m = J$, $\psi_i = x_i$, слова φ_i не содержат параметров x_j, множества y_p описываются КС-грамматикой, то такая система уравнений определяет операцию отождествления, аргументами которой являются переменные x_j, результатом - решение $(z_1^{k_1}, z_2^{k_2}, \ldots, z_n^{k_n})$, т.е. набор слов, превращающих уравнение в тождество. В общем случае эта операция является недетерминированной.

Реализация операции отождествления в общем виде приводит к громоздкой и неэффективной интерпретации с экспоненциальными временными оценками. Однако как средство описания алгоритмов, ориентированных на человека, эта операция в классе текстовых задач незаменима.

Управляющая система. Управляющая система - это интерпретатор, который может содержать описания модулей, интерпретаторов и грамматик, и, главное, расположенный в управляющей памяти. До сих пор управляющая память рассматривалась как черный ящик, в котором хранятся в виде упорядоченного списка имена функций. Такую память можно представить в виде одной функции

: A A1 A2 ... AN ;

где A1 - имена, как правило, неописанных функций. Все операции, работающие с управляющей памятью, легко описать как операции со списком A. Таким образом, управляющая память представляет собой стандартную управляющую систему с фиксированным набором операций.

Описание управляющей системы дает возможность влиять на выполнение до сих пор неуправляемых процессов. Прежде всего это относится к параллельным процессам. Управляющая система с параметрами реагирует только на обращение к функциям или модулям, являющимся ее параметрами. Освобождение системы может навсегда остановить приостановленные процессы: вызов новой управляющей системы закрывает доступ к управляющей памяти, а тем самым закрывает возможность запустить приостановленные процессы до освобождения вновь вызванной системы.

Управляющая система в общем случае полностью идентична программе, использующей все описанные средства языка, но находящейся как бы во втором слое памяти, недоступном для первого слоя, и выполняющаяся в привилегированном режиме. Описание управляющей системы может содержать описания других управляющих систем, которые образуют третий слой памяти, недоступный для второго. В принципе нет причин ограничивать количество таких слоев: система, описанная в системе i-го слоя, образует (i + 1)-й слой. Системы, описанные независимо одна от другой, образуют слои, недоступные друг для друга.

Управляющая система - это операционная система языка высокого уровня, причем операционная система, меняющаяся динамически в зависимости от требований решаемой задачи.

Итак, мы привели здесь теоретические предпосылки, определяющие направление поиска основ языка представления знаний, и предложили вариант абстрактного языка этого класса.

Заметим, что в основания математики положена теория множеств. Парадоксы теории множеств делают эти основания хрупкими. Их можно сделать более прочными, если заменить теорию множеств теорией языков. С точки зрения теории языков, одни семантические парадоксы (типа: "Я всегда лгу") представляют собой разновидность "дурной" рекурсии и легко разрешимы, другие (когда по множеству строится не принадлежащий этому множеству элемент) по существу не являются парадоксами: по любому языку можно построить семантически правильное предложение, смысл которого невыразим на данном языке.

К сожалению, мы очень коротко обсудили эти вопросы. Безусловно они заслуживают более подробного изложения.

Глава 2. СИСТЕМА ОБРАБОТКИ СЛОЖНООРГАНИЗОВАННОЙ ИНФОРМАЦИИ

В первой главе содержится общая постановка проблемы, решение которой позволило бы ответить на вопрос, каким должен быть язык представления знаний. Этот язык должен поддерживать функциональные методы программирования. Следовательно, он должен иметь развитую структуру управления, в частности, содержать средства описания недетерминированных и параллельных процессов. Кроме того, он должен поддерживать принцип активизации, т.е. должен содержать средства преобразования пассивных данных в активные. Наконец, он должен обеспечивать возможность перехода от математической формулировки задачи к решающему ее алгоритму. Таким образом, если содержание предыдущей главы сводилось к ответу на основной вопрос, "что должен делать язык", то содержание данной главы сводится к описанию того, "что язык может". Точнее что может система обработки информации, цель построения которой заключалось в реализации возможностей языка представления знаний. Но главное не в том, что она может, а в том, как ей удается делать то, что она может.

Основная цель данной главы - на примере конкретной системы обработки информации раскрыть сущность функционального подхода к программированию. Поэтому описание системы и ее возможностей имеет второстепенное значение по сравнению с описанием методов ее реализации. Точное описание методов реализации невозможно без явного предъявления текстов программ: именно в текстах программ заключается сущность подхода. Сказанное в какой-то степени должно служить оправданием (на первый взгляд кажущейся) перегруженности описания системы текстами программ.

Система реализована в виде набора функций на языке Форт-83. Поэтому для полного понимания ее описания необходимо знать язык Форт. Книга С.Н.Баранова и Н.Р.Ноздрунова [3] является неплохим учебником по этому языку. Инициатива реализации системы принадлежит А.А.Титову. Без его участия язык представления знаний остался бы абстрактным языком программирования.

§ I. Обработка списков

Список - это последовательность элементов, взятая в круглые скобки. Поскольку в языке Форт круглые скобки являются ограничителями комментариев, то для изображения круглых скобок используются слова (Ø и Ø). Элементами списка могут быть числа, идентификаторы, списки, ссылки, массивы чисел, массивы ссылок и функции.

Представление списков. Список после компиляции представляет собой суперпозицию функций, содержащую обращения к векторным переменным NUM , ID , LS , REF, ARRNUM, AREF и FUNC . Каждая такая переменная связывается с соответствующим элементом списка. Список преобразуется в активную форму в процессе компиляции. Левая скобка заменяется на последовательность ("w) 1 CALL , правая - на EXIT LS . Это преобразование осуществляется функциями (Ø и Ø).

```
: ("w) R@  2+ R> DUP @ + >R ;
: LENGTH! HERE OVER - SWAP ! ;
:((Ø)  COMPILE ("w) HERE 0 , CALL , ;
: (Ø)) COMPILE EXIT LENGTH! COMPILE LS ;
: (Ø  ((Ø)  I2 ; IMMEDIATE
: Ø) I2 < > IF." ? Ø) - НЕПАРНЫЕ СКОБКИ OR QUIT THEN
    (Ø)) ;  IMMEDIATE
```

Число представляется в виде LITA< число > NUM , идентификатор - (") < строка со счетчиком > ID , ссылка - LITAREF< адрес > REF , массив чисел - ("w)ANUM 1 < последовательность чисел > ARRNUM , массив ссылок - ("w)AREF 1< последовательность адресов > ARREF , функция - ("w)FUNC 1 CALL < тело функции >FUNC. 1 - длина последовательности в байтах, CALL - метка входа в адресный интерпретатор.

Функции LITA и LITAREF имеют идентичные описания:

```
:LITA R> DUP 2+ >R ;
```

Функции ("w), ("w)ANUM , ("w)AREF и ("w)FUNC также имеют одинаковые описания. Дублирование функций необходимо для того, чтобы по их кодам компиляции можно было идентифицировать тип элемента.

Пример. Список (Ø ABC 4 (Ø EF[RET] Ø)N 2 15 18 N] [R SWAP DROP R] [F R@ 2+ DROP F] (Ø [ARRNUM] 4 [AREF] 2Ø) Ø) в процессе компиляции преобразуется в последовательность: ("w) 116 CALL (") 3 ABC ID LITA 4 NUM ("w) 20 CALL (") 2 EF Ø ID LITAREF 0 REF EXIT LS ("w)ANUM 8 2 15 18 ARRNUM("w)AREF 6 SWAP DROP ARRE

```
("w)FUNC 10 CALL R@ 2+ DROP FUNC ("w) 30 CALL ("w)ANUM 10
0 0 0 0 ARRNUM ("w)AREF 6 0 0 ARREF EXIT LS EXIT LS .
```

Ввод списка и преобразование его в суперпозицию функций осуществляет функция INLIST. Скобки [N, N], [R, R], [F и F] являются функциями немедленного действия, описание которых аналогично описанию функций (Ø и Ø). Адрес метки CALL, которую компилирует функция (Ø является адресом компиляции списка. Передача управления по этому адресу инициирует выполнение списка. Эти адреса могут быть значениями списковых переменных или находиться в адресном стеке. Необходимость в адресном стеке возникла в связи со сборкой мусора: адрес в арифметическом стеке невозможно отличить от числа.

Списковые переменные описываются при помощи конструкции

```
LISTVAR
  LISTVAR N F1 F2 ... FN
```

Все они оформляются в виде активизированного списка и с каждым описанием переменной связывается векторная переменная MR . Такое представление переменных дает возможность производить любые действия с их описаниями.

Для работы с адресным стеком построен набор операций, аналогичных операциям с арифметическим стеком:

```
>A  - заслать элемент в адресный стек;
A>  - положить на стек верхний элемент адресного стека;
A@  - скопировать в стек верхний элемент адресного стека;
N AREVER  - переставить в обратном порядке N элементов адресного стека;
:EXEC A>  EXECUTE ;
```

Операции ADUP, ADROP, AOVER, ASWAP, AROT, APICK, AROLL полностью аналогичны операциям DUP, DROP и т.д.

Операции ?NUM, ?ID, ?LS, ?REF, ?ARRNUM, ?ARREF, ?RUNC предназначены для распознавания типа элемента списка. Операция MODE вырабатывает значение i (1 ≤ i ≤ 8) , соответствующее типу элемента. Если i = 1 , то элемент является числом, если i = 2 , то идентификатором и т.д. Если i = 8 , то тип элемента не распознан, что соответствует чаще всего ошибочной ситуации.

Функция ARREFP :

```
:ARREFP DUP DUP 2- @ + 2- SWAP
    ?DO I REF 2 +LOOP ;
```


позволяет обрабатывать массив ссылок, как бы превращая его в последовательность элементов типа "ссылка". Аналогичное описание (вместо переменной REF - переменная NUM) имеет функция ARRNUMP

Функции (:, :), (*,*) предназначены для присваивания значений векторным переменным NUM, ID и т.д. В результате выполнения суперпозиции

```
(: F1 F2 F3 F4 F5 F6 F7 :)
```

переменная NUM примет значение F1 , переменная ID - значение F2 и т.д. После выполнения последовательности (* F *) все переменные будут иметь одно и то же значение F.

Операции над списками. Функция LEN подсчитывает число элементов списка. Она использует один фактический параметр:

```
: LENP DROP 1+ ;
: LEN (* LENP *) 0 EXEC ;
```

Функция LEN - первая функция, запрограммированная в функциональном стиле, поэтому рассмотрим подробнее процесс ее выполнения. Адрес списка, который является аргументом этой функции, находится на вершине адресного стека. Функция LEN вначале присваивает всем векторным переменным NUM, ID и т.д. адрес компиляции функции LENP . Далее она кладет на стек ноль и передает управление списку. При выполнении списка функции ("), ("w) и т.д. оставляют на стеке адрес соответствующего элемента и передают управление на векторные переменные. Каждая из этих переменных выполняет функцию LENP , т.е. выбрасывает из стека адрес и прибавляет к верхнему элементу стека единицу. (Вначале на стеке находится число 0.) После выхода из списка на стеке останется число элементов этого списка.

Функция ADD2 находит сумму всех чисел списка:

```
: ADD2P @ + ;
: ADD2  (: ADD2P DROP EXECUTE REFP ARRNUMP ARREFP DROP :)
        0 EXEC ;
```

Вначале функция ADD2 присваивает значения векторным переменным-параметрам списка. Параметр NUM получает значение ADD2P , параметр ID - значение DROP и т.д. Функция REFP - достаточно универсальный фактический параметр для переменной REF . Эта функция разыменовывает ссылку и в зависимости от типа элемента работает

либо как NUM , либо как ID , либо как LS , либо как REF (в данном случае снова обращается сама к себе), либо как ARRNUM , либо как ARREF , либо как FUNC . При этом циклические ссылки обрабатываются один раз. Если на вход функции ADD2 подать список, рассмотренный выше в примере, то после ее выполнения на стеке останется число 39.

Функция ELEM вычисляет адрес i-го элемента списка:

```
: DLEMP >A 2DUP = IF 2DROP Ø RDROP
                  ELSE 1+ ADROP THEN ;
: ELEM ( * ELEMP *) 1 EXEC
       IF . ." -? ELEM - ВЫХОД ЗА ДИАПАЗОН" CR QUIT THEN;
```

При обращении к функции ELEM индекс i должен находиться на стеке, адрес списка – на адресном стеке. Функция ELEMP кладет адрес элемента на адресный стек и сравнивает текущий индекс с индексом 1 . При их совпадении на стеке остается ноль , в адресном стеке – искомый адрес, операция RDROP осуществит выход из списка. При несовпадении к текущему индексу добавится единица, а адрес, положенный в адресный стек, будет выброшен операцией ADROP .

Функция RELEM находит i-й с правого конца элемент списка:

```
: RELEM ADUP LEN SWAP - 1+ ELEM ;
```

Функция COPY копирует на вершину словаря не только списки, но и элементы других типов:

```
:  ICOPYP (ПАРАМЕТР ДЛЯ КОПИРОВАНИЯ ИДЕНТИФИКАТОРА)
       DUP 2- HERE ROT @ 5 + = CELLS
       DUP ALLOT CMOVE ?STACK ;
:  NRCOPYP (ПАРАМЕТР ДЛЯ КОПИРОВАНИЯ ЧИСЛА ИЛИ ССЫЛКИ)
       2- HERE 6 ALLOT 6 CMOVE ?STACK ;
:  ALCOPYP (ПАРАМЕТР ДЛЯ КОПИРОВАНИЯ СПИСКА, МАССИВА ИЛИ
            ФУНКЦИИ)
       4 - DUP 2+@ 4 + HERE SWAP DUP ALLOT CMOVE ?STACK ;
:  ICOPY (КОПИРОВАНИЕ ИДЕНТИФИКАТОРА)
       A> HERE 2+ >A ICOPYP ;
:  NRCOPY (КОПИРОВАНИЕ ЧИСЛА ИЛИ ССЫЛКИ)
       A> HERE 2+ >A NRCOPYP ;
:  ALCOPY (КОПИРОВАНИЕ СПИСКА, МАССИВА ИЛИ ФУНКЦИИ)
       A> HERE 4 + >A ALCOPYP ;
```



```
: COPY (КОПИРОВАНИЕ ЭЛЕМЕНТА СПИСКА)
      ?ID IF ICOPY
          ELSE ?NUM ?REF OR
            IF NRCOPY
            ELSE ?LS ?ARRNUM ?ARREF ?FUNC OR OR OR
              IF ALCOPY
              ELSE ." ?COPY - НЕПРАВИЛЬНЫЙ ЭЛЕМЕНТ "
                                    CR QUIT
              THEN
            THEN
      THEN ;
```

Функция SEQ строит по исходному списку новый список, который не содержит внутренних скобок и ссылок. В построенном списке могут остаться лишь висячие ссылки:

```
: SEQ (: NRCOPYP ICOPYP EXECUTE REFP ALCOPYP ARREFP ALCOPYP ;)
    HERE 4 + >A ASWAP ((Ø) EXEC (Ø)) ;
```

Функция CAR копирует первый элемент списка на вершину словаря:

```
: CARP >A COPY RDROP ;
: CAR (* CARP *) EXEC ;
```

Функция CONS вставляет значение первого аргумента в качестве первого элемента в список, который является значением второго аргумента:

```
: CONSP >A COPY ADROP ;
: CONS (* CONSP *) HERE 4 + >A 3 AREVER
                   ((Ø) COPY ADROP EXEC (Ø)) ;
```

Функция CDR по исходному списку строит новый список, выбрасывая первый элемент исходного списка:

```
: CDRP (* CONSP *) DROP ;
: CDR (* CDRP *) HERE 4 + >A ASWAP
                 ((Ø) EXEC (Ø)) ;
```

Функция ADMAX оставляет на стеке адрес максимального числа списка:

```
: ADMAXP DUP >R @ OVER > IF 2DROP R> DUP @
                         ELSE RDROP THEN ;
: ADMAX (: ADMAXP DROP EXECUTE REFP ARRNUMP ARREFP DROP :)
        0 -32768 EXEC DROP >A ;
```

Функция NULL определяет, пуст ли список. Эта функция не использует функциональное представление списка:

```
: NULL  A> 2- @  6 =
```

Функция LISTDEPTH вычисляет максимальную глубину вложенности подсписков в список:

```
: LISTDEPTHP >R 1+ 2DUP < IF PRESS DUP THEN
            R> EXECUTE 1- ;
: LISTDEPTH (: DROP DROP LISTDEPTHP REFP DROP ARREFP DROP :)
            0 0 EXEC DROP ;
```

Функция LLIST формирует один список из N элементов, адреса которых находятся в адресном стеке. Число N должно находиться на стеке:

```
:LLIST DUP AREVER ((@) >R 0 DO COPY ADROP
                          LOOP R@ (@)) R> 2+ >A ;
```

Функция NCTNLIST конкатенирует N списков. Число N должно быть на стеке:

```
:NCTNLIST (* CONSP *) DUP AREVER
          ((@) >R 0 DO EXEC
                   LOOP R@  (@)) R> 2+ >A ;
```

Функция CTNLIST строит конкатенацию двух списков:

```
: CTNLIST (* CONSP *)
          ((@) ASWAP EXEC EXEC DUP 2+ >A (@)) ;
```

Функция REVERSE переставляет в обратном порядке элементы списка. Порядок расположения элементов в подсписках остается прежним:

```
: LREV 0 EXEC HERE 4 +
       ((@) ROT 0 DO COPY ADROP LOOP (@)) >A ;
: REVERSEP >A 1+ ;
: REVERSE (* REVERSEP *) LREV ;
```

Функция REVL реверсирует порядок элементов списка и всех подсписков списка:

```
: REVLP >A LREV 1+ ;
: REVL (* REVERSEP *) LS< REVLP LREV ;
```

Функция EQUAL проверяет на равенство элементы списка.

```
: IEQUAL DUP C@  1+      = CELLS EQU ;
: NEQUAL @ SWAP @ <> ;
: LEQUAL DUP 2- @  EQU ;
```



```
: EQU OVER + SWAP ?DO I @  OVER < > IF DROP FALSE LEAVE
                                    ELSE 2+ THEN
                  2 +LOOP ;
: EQUAL A> ?ID IF A> IEQUAL
               ELSE ?NUM ?REF OR
                  IF A> NEQUAL
                  ELSE A> LEQUAL
                  THEN
               THEN ;
```

Функция MEMBER определяет, является ли первый аргумент членом списка, который является вторым аргументом:

```
: IEQUP DUP A@  IEQUAL IF OUTL ELSE DROP THEN ;
: LEQUP DUP A@  LEQUAL IF OUTL
                       ELSE DUP 2- @ A@  2- @ >
                          IF EXECUTE  ELSE DROP THEN
                       THEN ;
: ANEQUP DUP A@ LEQUAL IF OUTL ELSE DROP THEN ;
: AREQUP DUP >R ANEQUP R> ARREFP ;
: MEMBERP [F NUM< NEQUP ARRNUM< ARRNUMP F]
          [F ID< IEQUP F]  [F LS< LEQUP F]
          [F REF< NEQUP F]
          [F ARRNUM< ANEQUP F] [F ARREF< AREQUP F]
          [F FUNC< ANEQUP F] [F ."?MEMBER- НЕ ЭЛЕМЕНТ
               СПИСКА" QUPT F] ;
: MEMBER (* DROP *) LS< EXECUTE REF< REFP ARREF< ARREFP
        MODE FELEMEX MEMBERP 0 >R EXECUTE
        R@  IF ADROP >A TRUE
            ELSE RDROP FALSE
            THEN ;
```

§2. Операции над множествами

Множество представляется в виде массива ссылок.

Функция ORDER упорядочивает массив чисел по возрастанию:

```
: ORDERP DUP >R @  2DUP >
         IF R@  2- ! R@  ! ADROP FALSE >A
         ELSE 2DROP THEN R> @ ;
: ORDER REF1< ORDERP
```

```
        BEGIN   -32768 A@  TRUE >A ARREFP1 DROP A>
        UNTIL ADROP  ;
```

Здесь функция ARREFP1 аналогична описанной выше функции ARREFP. Переменная REF в ней заменена на переменную REF1 . Функция REF1< присваивает векторной переменной REF1 значение ORDERP.

Функция UNION по двум множествам строит их объединение. Функции SETP1, SETP2 и SETP3 - вспомогательные функции, которые являются фактическими параметрами переменных:

```
:SETP1  @  OVER  =  IF OUTL THEN  ;
:SETP2  @  A@  0 >R ARREFP R@  DUP
        IFNOT FDROP THEN NUM
        IF LROP ELSE , THEN  ;
:SETP3 DUP IFNOT 2 A@  2-  +1 THEN  ;
:UNION HERE 2+ A> COPY -2 ALLOT
        REF< SETP1 REF1< SETP2 NUM< 0< >
        A1REFP1 LENGTH! COMPILE ARREF  ;
```

Используемая здесь функция OUTL обеспечивает выход из вложенных списков и циклов:

```
:OUTL BEGIN R> R@  WHILE DROP REPEAT
      ADROP >R  ;
```

Функция UNION вначале копирует на вершину словаря первое множество, затем присваивает переменным REF, REF1 и NUM соответственно значения SETP1, SETP2 и 0< >. После этого функция ARREFP1 добавляет к скопированному первому множеству те элементы из второго множества, которые не содержатся в первом.

Функции INTERSECT и DIFF по двум множествам строят соответственно их пересечение и разность. Функция CREASET является общей частью для этих функций:

```
: CREASET COMPILE ("w)AREF HERE 2 , HERE >A
          3 RSVER REF< SETP1 REF1< SETP2 A> ARREFP1
          ADROP LENGTH! COMPILE ARREF  ;
: INTERSECT  NUM<  0= , CREASET  ;
: DIFF NUM< 0< > CREASET  ;
```

Функция ARR -> SET преобразует массив ссылок в множество. Массив может содержать одинаковые ссылки. Множество не содержит одинаковых ссылок.


```
:ARR -> SET HERE 4 + >A NUM< SETP3 CREASET ;
```

Функция CTNSET конкатенирует два множества. Результатом в общем случае является не множество, а массив:

```
:CTNSETP @  ,  ;
:CTNSET A> COPY -2 ALLOT REF1< CTNSETP ARREFP1
      A@  2- LENGTH! COMPILE ARREF  ;
```

§3. Операции реляционной алгебры

Отношения (или реляционные таблицы) в данной системе представлены в форме специфицированного списка с именем:

RELATION F [ATR A1,...,AN ATR] (Ø <1-й кортеж Ø) ...
...(Ø <k-й кортеж > Ø) ENDREL

где RELATION , [ATR, ATR] и ENDREL - функции (три последние - немедленного действия), преобразующие внешнюю форму отношения во внутреннее представление, F - имя отношения, A1 ,A2 ,..., AN - имена атрибутов, которые должны быть описаны как переменные типа QUAN. В отличие от обычной реляционной таблицы кортеж может содержать элементы любого из семи типов. Единственное ограничение на кортеж - все кортежи должны содержать одинаковое число элементов.

Функция RELATION строит заголовок отношения и проверяет наличие левой скобки списка атрибутов - функции [ATR. Функции [ATR и ATR] строят массив ссылок аналогично функциям [R и R] :

```
:ATR] 16 <> IF ." ?ATR  - непарные скобки" CR QUIT THEN
     LENGTH! COMPILE ARREF  [COMPILE]  INLIST ; IMMEDIATE
:[ATR COMPILE ("w)ARREF HERE 0 , 16
     BEGIN ' DUP ['] ATR] =  IF EXECUTE EXIT
                             ELSE  , THEN
     AGAIN  ;  IMMEDIATE
: RELATION ?EXEC  [COMPILE] : ' DUP  ['] [ATR =
   IF EXECUTE
   ELSE , " ?RELATION - "LATEST ID. "НЕ ОТНОШЕНИЕ"
        CR QUIT
   THEN  ;
```

Функция ENDREL производит преобразование введенных кортежей: все векторные переменные - NUM , IL , LS , REF , ARRNUM, ARREF

и FUNCона заменяет на адреса компиляции функций, ТО A1 , ТО A2 ,... ..., ТО AN. Векторные переменные остаются лишь при массиве атрибутов (переменная ARREF) и при каждом списке, который представляет кортеж (переменная LS). Функция ATR! заменяет векторную переменную на адрес компиляции функции ТО Ai :

```
: ATR! DROP 4 + R@  2-  ! ;
: LSATR AT LS SWAP (* ATR! *) A@  ARREF@
        A@  2- @  2/ RETER EXECUTE TO LS ;
```

Функция LSATR является параметром для векторной переменной LS. Эта функция вначале сохраняет на стеке значение параметра LS - адрес компиляции функции LSATR, затем присваивает всем векторным переменным значение ATR! . После этого считывает в стек из адресного стека адреса компиляции переменных A1 , A2 ,..., AN , переставляет в обратном порядке (N+ I) элементов стека, так что на стеке оказывается адрес списка (кортежа), и передает ему управление. Восстановив значение переменной LS , функция LSATR заканчивает свою работу:

```
: ENDREL   COMPILE   ;  LS< LSATR ARREF< >A
          LATEST NAME> EXECUTE ADROP RDROP L; ;
```

Вначале функция ENDREL завершает определение отношения F . Затем она начинает его модифицировать: присваивает переменным LS и ARREF соответственно значения LSATR и >A и передает вновь построенному отношению управление. В результате выполнения отношения F все векторные переменные в каждом кортеже будут заменены на коды компиляции функций ТО Ai.

Функция UNIONREL вычисляет объединение двух отношений:

```
: RELFND EXEC ADROP >A COMPILE EXIT :
: UNIONP DUP >A AOVER MEMBER
        IF DROP ELSE CONSP THEN ADROP;
: UNIONREL (* CONSP *) HERE CALL , AOVER EXEC
       ARREF< DROP LS< UNIONP RELEND :
```

Функция INTERREL вычисляет пересечение двух отношений:

```
: INTERP >A AOVER MEMBER IF COPY
                        THEN ADROP ;
: INTERREL HERE CALL ,
          ARREF< CONSP LS< INTERP RELEND ;
```

Функция DIFREL вычисляет разность двух отношений.


```
:DIFREL HERE CALL , ASWAP
       ARREF< CONSP LS< UNIONP RELEND ;
```

Функция SELECT выбирает из отношения столбцы с именами атрибутов $Ai_1, Ai_2, \ldots, Ai_k$. Обращение к этой функции имеет вид

```
[R Ai1 Ai2 ... Aik R]   SELECT
```

Адрес компиляции отношения, из которого идет выборка, должен находиться на адресном стеке:

```
:SELECT  -2 ALLOT COMPILE (SELECT) ; IMMEDIATE
:(SELECT) >A HERE CALL , COPY SWAP  -2 ALLOT
    COMPILE ARREF ARREF< DROP
    [F @  EXECUTE CONSP F] [ -2 ALLOT ] TO REF
    [F EXECUTE ((R) A@  ARREFP (R)) F] [ -2 ALLOT ] TO LS
     RELEND ;
```

Функция DEC* вычисляет декартово произведение двух отношений:

```
:DEC*P HERE 2+ 2 PICK CONSP -4 ALLOT SWAP 2+ HERE
       OVER 4 - @  2- DUP ALLOT CMOVE ?STACK LENGTH! ;
:DEC* ARREF< RDROP ASWAP AOVER EXEC ADUP EXEC
      HERE CALL , >A >A CONSET ADROP
     [F AT LS SWAP LS< DEC*P A@  EXECUTE DROP TO LS F]
      [ -2 ALLOT ] TO LS ARREF< DROP RELEND ;
```

Функция SELECT2 выбирает из отношения все кортежи с заданным значением атрибута Ai. Обращение к этой функции имеет вид

```
SELECT2  Ai  < значение >  L;
```

Значением может быть элемент любого из семи типов. Адрес компиляции отношения, из которого идет выборка, должен находиться на адресном стеке:

```
: SELECT2  [COMPILE] ['] [COMPILE] INLIST
           -2 ALLOT COMPILE (SELECT2) ; IMMEDIATE
: SELECT2P DUP  >A EXECUTE OVER EXECUTE >A DUP >A
           EQUAL IF COPY THEN ADROP ;
: (SELECT2) HERE CALL , >A ASWAP
            LS< SELECTP2 ARREF< DROP EXEC
            2DROP COMPILE EXIT ;
```

Функция RELEQUAL сравнивает два кортежа на равенство значений тех атрибутов, которые заданы в виде множества. Адреса кортежей должны находиться на стеке, адрес множества - на адресном стеке.

```
        : RELEQUAIP @   EXECUTE CONSP  ;
: RELEQUAL  REF1< RELEQUAIP  EXECUTE
            HERE 4 + ((@ A@  ARREFP1  (@))
            SWAP  EXECUTE  HERE  4  +
            ((@) A> ARREFP1 (@)) >A DUP >A @QUAL
            SWAP 4 - HERE - ALLOT ;
```

Функция JOIN соединяет (конкатенирует) те кортежи двух отношений, у которых совпадают значения общих для обоих отношений атрибутов. Адреса отношений должны находиться на адресном стеке:

```
: JOIUP ADUP 2DUP RELEQUAL
        IF EXECUTE HERE 2+ OVER CONSP
           -4 ALLOT AOVER A> ARREFP1 (@))
        ELSE DROP THEN ;
: JOIN HERE CALL , REF1< RELEQUAIP ARREF< RDROP
       AOVER EXEC ADUP EXEC ARREF< DROP
       2DUP >A >A DIFF >A >A INTERSET
       [F AT LS SWAP LS< JOINP 2 APICK EXEC DROP TO LS F]
        I -2 ALLOT I TO LS 3 ROLL EXEC
       ADROP ADROP ADROP >A COMPILE EXIT ;
```

§4. Моделирование автоматов

Конечные автоматы. Конечный автомат задается функцией переходов вида $f(a,Q_i)=Q_j$, где a – символ входного алфавита, Q_i и Q_j – состояния автомата. Множество состояний конечно и однозначно идентифицируется набором целых чисел. Поэтому каждое состояние на языке Форт может быть описано как целочисленная константа: i CONSTANT Q_i. Для каждого символа a построим все пары состояний (Q_{i_s},Q_{j_s}), такие, что $f(a,Q_{i_s})=Q_{j_s}$ $(1\le s\le m)$ и определим функцию a следующим образом:

```
: a Qi1Qj1 * ... * QimQjm ⊥ ;
```

где функции * и ⊥ имеют вид

```
: * >R OVER = IF DROP R> THEN RDROP ;
: ⊥   * ABORT " ОШИБКА " ;
```

После этого входная цепочка $a_{i_1}a_{i_2}\dots a_{j_k}$, каждый символ которой отделен пробелом, представляет собой суперпозицию функций

на языке Форт, выполнение которой идентично работе конечного автомата.

Магазинные автоматы. Если конечный автомат представляет собой частичную функцию $f(a_{i_1}a_{i_2}\dots a_{j_k},Q)$, второй аргумент которой имеет элементарную структуру и поэтому его активизация не имеет смысла, то магазинный автомат является частичной функцией $f(a_{i_1}a_{i_2}\dots a_{i_k},z_{j_1}z_{j_2}\dots z_{j_s},Q)$ два первых аргумента которой могут быть активизированы. В результате порождаются два параллельных процесса.

Рассмотрим вначале случай, когда число состояний автомата равно единице. Детерминированный магазинный автомат с одним состоянием адекватен LL(1)-грамматике. Поскольку синтаксический анализ в этом классе грамматик сам по себе имеет важное практическое значение, рассмотрим проблему построения конструкторов синтаксических анализаторов в классе LL(1)-грамматик. Конструктором здесь является отображение Ψ, активизирующее класс LL(1)-грамматик. Это отображение каждой грамматике ставит в соответствие набор описаний функций. По каждому правилу грамматики строится одно описание функции:

– если правило имеет вид $A :: a\Psi\{f_i\}$, где a – терминал, f_i – семантическая функция, связанная с данным правилом, то по нему строится функция:

```
: Aa  Ψ̂  fi ;
```

где $\hat{\Psi}$ – слово Ψ, все символы которого разделены пробелами;

– по правилу $A :: H\Psi\{f_i\}$, где H – нетерминал и $a\in FIRST(H)$, строится функция:

```
: Aa Ha  Ψ̂  fi ;
```

– по правилу $A :: \{f_i\}$, если a следует за A хотя бы в одной выводимой цепочке, строится функция:

```
: Aa EMPTY fi ;        : EMPTY 1 1EMPTY ! ;
```

Если ограничиться только синтаксическим анализом, то каждая функция f_i может быть определена при помощи конструкции COMP :

```
: COMP CREATE DOES> 2- , ;
```

Символы грамматики как терминальные $a_1,a_2,\dots,a_n$, так и нетерми-

нальные $A_1, A_2, \ldots, A_M$ описываются при помощи конструкции SYMBOLS :

```
: SYMBOL CREATE DOES> 2- >NAME PAUSE EXECUTE ;
: SYMBOLS Ø DO SYMBOL LOOP ;
  K SYMBOLS A1 A2 ... AM a1 a2 ... aN   (K= M + N)
  VARIABLE 1EMPTY
```

Пример. Грамматика G_0:

$S :: (S\ S)\ \{f_1\} \mid a\ \{f_2\}$,

отображается в набор описаний:

```
4 SYMBOLS S ) a (
: S( S S ) f1 ;
: Sa f2 ;
```

Если грамматика не содержит правил с пустыми правыми частями, то синтаксический анализатор представляет собой оператор запуска двух параллельных процессов:

```
START( S F ),
```

где S - начальный символ грамматики, F - функция, имеющая описание:

```
: F BEGIN BL WORD KTN FIND Ø= Ø ?ERROR PAUSE AGAIN ;
```

В общем случае синтаксический анализатор запускает на параллельное выполнение функции G и F:

```
: G Ø 1EMPTY ! S ;
: F BEGIN 1EMPTY @  IF Ø 1EMPTY ! PAD>
                    ELSE BL WORD >PAD
                    THEN KTN FIND Ø=Ø ?ERROR PAUSE AGAIN ;
```

Здесь KTN, >PAD и PAD> - функции катенации строк, копирования строки с вершины словаря в область PAD и обратно. Анализируемая строка должна находиться во входном потоке.

По описанию магазинного автомата с многими состояниями набор необходимых для его моделирования функций строится следующим образом:

если команда перехода имеет вид $(z,a,Q_i) \to (\Psi,Q_j)$, где z - символ рабочего алфавита, то по нему строится функция:

```
: zaQi Qj Ψ̂ ;
```

если команда перехода имеет вид $(z,Q_i) \to (\Psi,Q_j)$, то строится функция:

```
: zQi EMPTY Qj Ψ̂ ;
```


Второй процесс уже при помощи катенации трех символов образует имя функции zaQ_i, поэтому его описание несколько усложняется:

```
: F BEGIN 1EMPTY @ IF Ø 1EMPTY ! ELSE BL WORD THEN
  2DUP KTN 2 PICK SWAP KTN FIND IF 2DROP ELSE KTN
  FIND Ø= Ø ?ERROR THEN PAUSE AGAIN ;
```

Магазинные автоматы с N магазинами индуцируют N + I параллельный процесс. Структура порождаемых функций остается такой же как и при N = I. Все функции делятся на N классов. Каждый процесс обращается к функциям лишь своего класса. Анализируемая строка порождает (N + I)-й процесс. В общем случае имя функции имеет вид: $Z1Z2\ldots ZNaQ_i$. Проблема моделирования других классов автоматов легко сводится к моделированию N-магазинных автоматов.

§5. Недетерминированные функции

Обращение к недетерминированной функции порождает недетерминированный процесс. Главной проблемой в организации недетерминированных процессов является проблема восстановления памяти при возвратах. Эта проблема решается достаточно просто, если на недетерминированные функции наложить ряд ограничений. Для функций это вполне естественные ограничения. Во-первых, число аргументов функции должно быть фиксированным и одинаковым для всех ее экземпляров. Во-вторых, функция не должна стирать информацию в словаре, она может лишь записывать новую, начиная с вершины словаря (которую имеет право стирать). В-третьих, при неуспешном выполнении она должна восстанавливать стек данных. При функциональном методе программирования, когда стек данных используется лишь для хранения аргументов функций, а вся новая информация, не стирая старой, записывается на вершину словаря, эти ограничения можно считать приемлимыми. Возможности описания недетерминированных процессов при этих ограничениях по крайней мере не ниже возможностей языка Пролог.

Возможны различные варианты реализации недетерминированных процессов на языке Форт. По-разному можно вводить описания недетерминированных функций, разной может быть и форма обращения к ним, наконец, по-разному можно хранить информацию о точках возврата. В предлагаемом ниже варианте информация о точках возврата

представлена в виде суперпозиции функций, готовой к выполнению. Неуспешное выполнение какой-либо функции передает ей управление, и она сама восстанавливает память и организует возврат. Эта суперпозиция строится операторами обращений к недетерминированным функциям, которые сами являются действующими суперпозициями функций.

Описание недетерминированных функций имеет вид

N ND: F K ARG: X1 X2...XK ☼ <тело функции> ... ☼ <тело функции> ND ; где N - максимальное число экземпляров функции, K - число аргументов, X1 , X2 ,..., XK - формальные параметры. Некоторые экземпляры функции могут быть описаны отдельно в виде

☼ F <тело функции> ND;

или порождаться динамически и присваиваться F функцией NTO .

Функции ND:, ARG: , ☼ , ND и NTO имеют описания:

```
: ND: CREATE HERE SWAP DUP C, Ø C, 1+ 2* ALLOT DOES> DUP
      1+ C@  2 PICK <IF 1 NFAIL ! EXIT THEN
      DUP DUP C@  2* 2+ + @ EXECUTE SWAP 2* + @ EXECUTE ;
: ARG: HERE 2- >R >R R@  Ø DO CREATE HERE 2- Ø , LOOP R>
      HERE R> ! CALL , Ø DO , COMPILE ! LOOP ] ;
: ☼  STATE @  IF COMPILE EXIT ELSE ' 2+ THEN HERE OVER
      ND! CALL , ] ;  IMMEDIATE
: ND! DUP C@  OVER 1+ C@  1+ DUP >R <
       IF 2- >NAME ID. ". ? - ПЕРЕПОЛНЕНИЕ" CR QUIT
       ELSE DUP 1+ R@  SWAP C! R> 2* + ! THEN ;
: ND;  DROP  [COMPILE]  ;  ; IMMEDIATE
: NTO R> DUP 2+ >R @  2+ HERE SWAP ND! ;
: SH! 2 SH -! SH @ ! ;
```

С функцией F связывается вектор состоящий из N + 2 двухбайтовых слов. Первое слово содержит числа N и N1 , где N1 - фактическое число экземпляров функции F . Это число может меняться в процессе вычислений. Следующие N слов содержат адреса компиляции экземпляров функции F . Если N1 меньше N , то последние (N - N1) слова содержат нуль, (N + 2)-е слово содержит адрес компиляции функции, которая присваивает формальным параметрам значения фактических параметров. Ее создает функция ARG:. Заполнение полей вектора осуществляет функция ☼ . Функция ND; выбрасывает адрес компиляции функции F и завершает ее определение.

Обращение к недетерминированной функции G имеет вид

SUPER <(E1 E2 ... EK)> G ;

где E1, E2,...,EK - выражения, вычисляющие значения фактических параметров, а функция SUPER, <(,)> имеют описания:

```
: SUPER  ['] SUPER1 SH! ;
: <(  [F ;  IMMEDIATE
: )>  F] -2 ALLOT  [COMPILE]  ['] COMPILE NDCOPY ;  IMMEDIATE
```

Функции <(и)> компилируют запись вида

("W)FUNC 1 CALL E1 E2 ... EK EXIT LIT F NDCOPY

При выполнении этой записи на стек будут положены адрес начала вычисления фактических параметров и код компиляции функции F , после чего управление будет передано функции NDCOPY . Функция SUPER может отсутствовать в обращении функции F . Ее наличие означает, что при успешном выполнении функции F необходимо построить суперпозицию из успешно выполненных экземпляров недетерминированных функций. Адрес компиляции этой функции остается на стеке.

Функция NDCOPY в области, адрес начала которой содержит переменная SH , формирует справа налево запись вида

CALL ("W)FUNC 1 CALL E1 E2 ... EK EXIT LS ("W) 11 "стек возвратов" ARREF LITA Ø NUM LITAREF F REF SUPER1 SUC

Адрес компиляции функции SUPER1 в область SH заносит функция SUPER:

```
: NDCOPY ['] SUC SH @ DUP @ CALL = IFNOT 2+ THEN ! ['] REF
        SH! SH! ['] LITAREF SH! ['] 
        NUM SH! Ø SH! ['] LITA SH! ['] ARREF SH! RP@
        R@ @ - DUP BEGIN ?DUP WHILE R> SH! 1- REPEAT 2+
         SH! ['] ("W) SH! ['] LS SH! DUP 4 - SWAP 2- @ 2+
         SH @ OVER - SWAP CMOVE CALL SH! PUSK ;
: NREF @  SWAP ;
: SUC NFALL @ IF CALL R> DUP @ ['] SUPER1 = IFNOT 2- THEN
     DUP SH ! ! PUSK ELSE FDROP THEN ;
: SUPER1 NFALL @ IFNOT (* DROP *) NUM< @ REF< NREF HERE Ø
     SH @ EXECUTE CALL , BEGIN ?DUP WHILE, REPEAT COMPILE
     EXIT THEN ;
: NDR@ R> RP! SWAP 2- DUP @ BEGIN 2- ?DUP WHILE SWAP 2+
     SWAP OVER @ >R REPEAT DROP >R ;
: NUM@  DUP @ 1+ OVER ! @ ;
```

```
: REF@ @ EXECUTE ;
: PUSK (: NUM@ DROP EXECUTE REF@ DROP NDR@ DROP :) Ø
    NFAIL ! SH @ EXECUTE ;
```

В начальный момент SH = SNØ- 2 . Значением переменной SNØ является адрес самого правого конца области хранения информации для возвратов. По этому адресу хранится код компиляции функции FALL :

```
: FAIL ABORT "ПОЛНАЯ НЕУДАЧА"
```

Вся информация, хранимая в области SK , доступна для программиста и он имеет более широкие, чем в языке Пролог, возможности для управления механизмом возвратов. Допустим, что необходимо вернуться к точке вызова функции A , после которой были вызваны недетерминированные функции A1,A2,..., AN , все имена которых отличны от A . Этот возврат может осуществить функция RETURN1 :

```
:C @ OVER = IF DROP R> 4 + >R SUC THEN ;
:RETURN1 (* DROP *)REF< C 1 NFAIL! R@ @ R> 2+ >R SH@EXECUTE;
```

Если функция RETURN1 осуществляет возврат к ближайшей точке вызова функции A , то функция N RETURN A - к N-й точке ее вызова (среди функций A1,A2,...,AM(N - 1) имен совпадает с A):

```
: D 1+ DUP >R OVER 2OVER = IF 2DROP RDROP R> 4 + >R SUC
                          ELSE R> THEN ;
:RETURN (* DROP *)REF< D 1 NFAIL ;R@ @ R> 2+ >R Ø SH@EXECUTE
```

Параметр NUM позволяет управлять порядком вызова экземпляров функции при возврате. Допустим, нужно вызвать i-й экземпляр последней вызванной функции. Для этого достаточно в качестве вызова фактического параметра для NUM взять функцию ! :

```
:ELEMI (: ! DROP EXECUTE REF@ DROP NDR@ DROP :) DUP
    SH @ EXECUTE ;
```

Обращение i ELEMI осуществит возврат к последней точке вызова недетерминированной функции. Перебор экземпляров этой функции начнется с ее i-го экземпляра.

§6. Сборка мусора

"Сборка мусора" - термин, который до сих пор не нашел себе достойной замены, несмотря на неоднократные попытки заменить его более подходящим термином. По смыслу более подходят термины "уборка мусора", "выбрасывание мусора", "утилизация памяти" и

т.п. Но традиция имеет силу закона, поэтому "сборка мусора" - законный термин, который используется всегда, когда речь идет о том, чтобы освободить память от мусора, т.е. от тех значений, к которым нет доступа.

Система обработки информации, описанию которой посвящена данная глава, не может обойтись без сборщика мусора, также как и любая аналогичная ей система, в которой значения - списки, отношения, тексты и т.п. - занимают, как правило, большой объем памяти, а сами значения и ссылки на них порождаются и уничтожаются динамически в процессе вычислений. Сборщик мусора в данной системе запрограммирован в функциональном стиле, что позволяет и даже обязывает подробно рассмотреть как структуру, так и процесс его выполнения. Сборщик мусора - обычная функция, доступная для использования наравне с другими функциями системы. Он оформлен в виде функции SHFL :

```
: SHFL COMPILE EXIT MARK LMOVE ;
```

Вначале функция SHFL компилирует на вершину словаря функцию EXIT. Затем она отмечает доступные значения, после чего отмеченные значения перемещаются на новое место. Первую часть работы выполняет функция MARK, вторую - функция LMOVE. Для того чтобы понять процесс выполнения этих функций, необходимо рассмотреть внутреннее представление множества переменных и множества значений в памяти ЭВМ.

Описание переменных находится внутри программы, представляющей собой обычный текст на языке Форт, который начинается с функции LBEGIN и кончается функцией LEND:

```
:LBEGIN HERE TO LVØ CALL , (BR ;
:LEND BR) COMPILE EXIT INSTA ;
```

После компиляции программа - это единая функция, адрес компиляции которой содержится в переменной LVØ. Эта функция имеет следующую структуру:

```
CALL(BR ... BR)< описания> (BR...BR) < описания> ...
    ...(BR...BR) EXIT ,
```

где < описания> - последовательность списковых переменных, которые определяются конструкцией LISTVAR :

```
: LISTVAR >R BR) ((M) R> 0
          DO ((@) -2 ALLOT QUAN
```

```
            LENGTH! COMPILE MR
        LOOP (R)  (BR  ;
```

Функции (BR и BR) компилируют условные переходы, позволяющие обойти текст (произвольный текст на языке Форт), находящийся между ними:

```
:(BR COMPILE BRANCH HERE 0 , 14  ;
:BR)  14  <>  IF  ." ?BR ) -НЕПАРНЫЕ СКОБКИ" CR QUIT
            THEN LENGTH!  ;
```

Функция INISTA , к которой обращается функция LEND , инициализирует адресный стек, под который отводится 200 байтов, и область значений, адрес начала которой присваивается векторной переменной LVAL:

```
: INISTA HERE TO LV 2ØØ ALLOT HERE 2-
        DUP DUP ! TO AØ HERE TO LVAL CALL , ;
```

Таким образом, область значений также оформляется в виде функции, адрес компиляции которой содержится в переменной LVAL . Тело этой функции не содержит лишь вхождения функции EXIT - ее компилирует в самом начале своей работы сборщик мусора. В принципе вся область значений доступна для использования. Такая возможность может быть полезна, особенно в процессе составления и отладки программы.

Функция MARK присваивает значения семи стандартным векторным переменным и переменной MR и передает управление на функцию, которая является внутренним представлением программы:

```
: MARK (: DROP DROP EXECUTE MARKP DROP ARREFP DROP :)
      [F NAME>  >BODY REF F]  [ -2 ALLOT ] TO MR
      LVØ ASTACKP  ;
```

Функция ASTACKP просматривает адресный стек и отмечает все доступные из него значения:

```
: ASTACKP APØ  AØ
        ?DO I REF 2 +LOOP  ;
```

Основную работу выполняет функция MARKP , адрес компиляции которой присваивается переменной REF . Эта функция отмечает доступные значения, заменяя векторные переменные NUM , ID ,... FUNC (с каждым значением связано одно вхождение переменной) на переменные SNUM , SID ,..., SFUNC .


```
: MARKP1  [F ['] SNUM A> 2+ ! F]
          [F ['] SID A> DUP @ 1+ =CELLS + ! F]
          [F ['] SLS A@ >R DUP @ + 2DUP @ =
             IF 2DROP ADROP ELSE ! RDROP A> 2+ >R THEN F]
          [F ['] SREF A@ 2+ 2DUP @ =
             IF 2DROP ADROP
             ELSE ! A> RDROP AT REF 2+ >R THEN F]
          [F ['] SARRNUM A> 2- DUP @ + ! F]
          [F ['] SARREF A@ 2- DUP @ + 2DUP @ =
             IF 2DROP ADROP ELSE ! A> RDROP ['] ARREFP
               2+ >A THEN F]
          [F ['] SFUNC A> 2- DUP @ + ! F]  ;
: MARKP @ ?DUP
        IF >A MODE FELEMEX MARKP1
        THEN
```

Функция MARKP1 представляет по-существу массив функций. Функция FELEMEX выбирает в зависимости от типа отмечаемого значения одну из функций этого массива и выполняет ее.

```
: FELEMEX [COMPILE] ['] 
        IF >A ELEM EXEC F] [ -2 ALLOT ]
         STATE @  IF ,
                  ELSE EXECUTE
                  THEN ;  IMMEDIATE
```

Функция LMOVE инициирует передвижение всех отмеченных значений на новое место:

```
: LMOVE AT LVAL 2+ HERE - ALLOT
       ['] MOVENUM TO SNUM ['] MOVEID TO SID
       ['] MOVELS TO SLS ['] MOVEREF TO SREF
       ['] MOVEARN TO SARRNUM ['] MOVEAREF TO SARREF
       ['] MOVEFUNC TO SFUNC
       (* DROP *) LS< EXECUTE LVAL  ;
```

Вначале эта функция устанавливает адрес вершины словаря на начало области значений. Затем каждой переменной SNUM , SID ,... ..., SFUNC присваивается код компиляции функций MOVENUM , MOVEID ,..., MOVEFUNC , каждая из которых передвигает на новое место либо число, либо идентификатор, либо ....функцию. Затем

семи стандартным векторным переменным, кроме LS , присваивается значение DROP и управление передается области значений. После этого каждое отмеченное значение передвигается на новое место самостоятельно:

```
: MOVEP DUP DUP A> - HERE =
        IF DROP R@ HERE - ALLOT
        ELSE SETREF >A COPY A> THEN ;
: MOVENUM ['] NUM R@ 2- ! 2 >A MOVEP DROP ;
: MOVEID ['] ID R@ 2- ! 2 >A MOVEP DROP ;
: MOVELS ['] LS R@ 2- ! 4 >A MOVEP 2+ >R ;
: MOVEREF ['] REF R@ 2- ! 2 >A MOVEP DROP ;
: MOVEARN ['] ARRNUM R@ 2- ! 4 >A MOVEP DROP ;
: MOVEAREF ['] AREF R@ 2- ! 4 >A MOVEP DROP ;
: MOVEFUNC ['] FUNC R@ 2- ! 4 >A MOVEP DROP ;
```

Функция MOVEP является общей частью для всех семи функций. Вначале каждая из этих семи функций заменяет переменную вида SP на F , после чего обращается к функции MOVEP . Эта функция проверяет, нужно ли передвигать данный элемент: если текущий адрес вершины словаря отличается от адреса данного элемента (в зависимости от его типа) на два или на четыре, то элемент остается на месте. Меняется лишь адрес вершины словаря. В противном случае происходит его перемещение на новое место. Так как при этом меняется его адрес, то необходимо установить новые ссылки. Эту работу выполняет функция SETREF :

```
: SETREF REF< SETADDR AT SREF >R ['] SETADDR TO SREF
        LV@ ASTACKP REF< DROP R> TO SREF ;
```

Вначале эта функция присваивает переменным REF и SREF новое значение – код компиляции функции SETADDR. Прежде чем закончить работу функция SETREF восстанавливает значения переменных REF и SREF .

Функция SETADDR устанавливает новые адреса:

```
: SETADDR DUP >R @ ?DUP
        IF >A DUP A@ =
           IF 2 ?LS ? ARRNUM OR ?ARREF OR
              IF 2+ THEN HERE + R> ! EXIT
           ELSE DUP >A ?LS ?ARRNUM OR ?ARREF OR
              IF A> 4@ 2DUP U< >R SWAP DUP 2- @
                 > OVER SWAP U< A> AND
                 IF OVER - HERE + 4 + R> ! EXIT
                 THEN DROP
              ELSE ADROP
              THEN
        THEN RDROP
          ?LS IF A> 2+ >R
              ELSE ?REF
                 IF A> [ LATEST NAME> >BODY ] >R
                 ELSE ?ARREF
                    IF A> ['] ARREFF 2+ >R
                    ELSE ADROP
                    THEN
                 THEN
              THEN
        ELSE RDROP
        THEN ;
```


§7. Альтернативные варианты реализации системы

Система реализована на языке Форт. В основу реализации положен принцип активизации исходной информации. Два эти факта могут служить основанием альтернативного выбора: система может быть реализована на другом языке другими методами.

Первая альтернатива: другой язык с сохранением принципа активизации. Как было сказано во введении, вряд ли возможно представить список, отношение, произвольный текст или граф в виде функции на таких языках, как ПЛ/1, Алгол-68, Паскаль, Ада. То же самое можно сказать про интерпретируемые языки Лисп, Планер, Пролог. Группа языков объектно-ориентированного программирования – Симула-67, Модула-2, Смолток – в большей степени, чем другие языки, обеспечивают программную поддержку принципа активизации, но тем не менее существенно уступают в этом языку Форт. При сохранении принципа активизации альтернативы языку Форт (по крайней мере среди широкораспространенных языков программирования) нет.

Вторая альтернатива: язык Форт без ограничения на форму представления информации. Это значит, что обрабатываемые объекты представлены в обычной пассивной форме. Адекватной заменой класса

активизированных объектов может служить только операция EVAL (выполнить), которая "выполняет" объекты данного класса: просматривая объект и анализируя его составные части, она в зависимости от структуры объекта выполняет те или иные действия. Представим, например, элементы списка в пассивной форме. Первые два байта перед каждым элементом выделим для информации о типе значения (например, три старших бита) и о длине значения. Тогда выполнение активного списка будет идентично выполнению функции EVAL (адрес списка должен находиться на стеке):

```
: EVAL  BEGIN  2+
          DUP DUP >R @  3400 AND 64 / DUP 1  =
            IF DROP NUM
             ELSE DUP 2  =
               IF DROP ID
               ELSE DUP 3  =
                IF DROP LS
               ELSE DUP 4  =
                IF DROP REF
               ELSE DUP 5  =
                IF DROP ARRNUM
               ELSE DUP 6  =
                IF DROP ARREF
               ELSE DUP 7  =
                IF DROP FUNC
               ELSE EXIT
               THEN
             THEN
            THEN
           THEN
          THEN
         THEN
        THEN
     R> DUP @  17777 AND +
   AGAIN
```

Такая реализация снижает возможности системы. Во-первых, список или любой другой элемент списка могут находиться внутри тела любой функции, во-вторых, обработка должна быть управляемой,

Оба эти условия невыполнимы, если список представлен в пассивной форме. Выполнение этих условий немедленно приводит к активной форме представления списка.

Третья альтернатива: другой язык без сохранения принципа активизации. Для адекватной замены принципа активизации необходима реализация приведенной выше операции EVAL . Ни на каком компилируемом или объектно-ориентированном языке эта операция без перехода в режим интерпретации не реализуема. На языках Лисп или Плэнер ее можно описать, но она будет крайне неэффективна по быстродействию.

Из сказанного следует, что альтернативы для языка Форт и метода реализации системы нет, а поскольку знание - это информация в активной форме, то в основу языка представления знаний должен быть положен язык в чем-то аналогичный языку Форт. Более того, поиски внутреннего представления активизированных объектов и опыт реализации системы привели к твердому убеждению, что внутренним представлением этих объектов должен быть косвенный шитый код, и именно он должен служить средством реализации языка представления знаний.

Косвенный шитый код является необходимым, но далеко не достаточным средством представления знаний. Недостаточна для этого и та программная поддержка, которую обеспечивает язык Форт. Здесь прежде всего следует сказать об отсутствии в этом языке средств описания параллельных процессов. Многозадачный режим, который легко реализуем в рамках этого языка, не пригоден для реализации множества мелких, но тесно взаимодействующих информационных процессов, в частности, для реализации механизма их синхронизации. Это существенно ограничивает область применения принципа активизации, так как активизация аргументов n-арной функции порождает n параллельных процессов.

Язык Форт недостаточен также для адекватного представления 2-го, 3-го, 4-го и 7-го типов функций (и в особенности модулей) языка Алмаз. Для их реализации необходима программная поддержка режима компиляции-выполнения и особого режима прерываний. Некоторые частные (но важные) случаи реализуемы в рамках языка Форт, некоторые лишь ценой модификаций глубинных механизмов языка, но в целом реализация этих режимов без аппаратной поддерж-

ки, по-видимому, невозможно. То же самое следует сказать и о режиме управляемой интерпретации.

Управляющая память, а тем более управляющая система, могут быть реализованы лишь в виде специализированных управляющих блоков центрального процессора ЭВМ.

И, наконец, последнее - составной частью языка представления знаний должен быть язык, близкий к естественному, который, с одной стороны, был бы посредником между человеком и ЭВМ, с другой - средством представления тех знаний об окружающем мире, которые могут быть адекватно представлены только на естественном языке.

В этой главе мы дали описание системы обработки слабоорганизованной информации. Эта система может рассматриваться в контексте предыдущей и следующей глав как практическая реализация некоторых абстрактных идей. Кроме этого ее можно использовать в качестве шаблона для создания аналогичных систем или просто как систему программирования для решения конкретных задач или, наконец, как средство обучения методам программирования на языке Форт.

Описанная система ориентирована на поддержку различных методов программирования. Она содержит средства обработки списков (объектов более сложных, чем списки языка Лисп), параллельные и недетерминированные процессы, что позволяет использовать логические методы программирования, средства обработки текстовой информации, в частности, конструкторы лексических и синтаксических анализаторов и генераторов кода, средства построения баз данных, средства, поддерживающие объектно-ориентированные методы программирования. Все это говорит о том, что возможности системы позволяют обрабатывать информационные потоки большой сложности.

Глава 3. ОСНОВЫ СЕМАНТИЧЕСКОГО ЯЗЫКА

В предыдущих главах основное внимание было уделено внутреннему представлению знаний. От правильного выбора представления зависит очень многое, даже сама возможность хранения и обработки знаний в ЭВМ. Но внутреннее представление возникает из внешнего в процессе компиляции. Одно представление является почти зеркальным отображением другого. Однако между ними есть принципиальное различие: внешнее представление должно обеспечивать взаимопонимание между человеком и машиной. Естественный язык (ЕЯ) обладает в этом отношении многими привлекательными чертами. Но лишь для человека. Для машины он слишком сложен. Нужен язык-посредник. Нет другого способа построить такой язык, кроме одного - формализовать естественный язык и тем самым упростить для машины процесс его понимания. Результатом такой формализации является семантический язык (СЯ). Этот язык, близкий к естественному языку, может служить посредником между человеком и ЭВМ. Описанию некоторых наиболее важных понятий, связанных с формализацией русского языка, посвящена данная глава. Ее содержание частично пересекается с содержанием работы [11], но в основном является его продолжением. Поэтому для полного понимания этой главы необходимо предварительное знакомство с работами [9, 11].

§ 1. Синтаксическая классификация предложений русского языка

Синтаксис отражает большинство языковых фактов и дает первую непосредственную информацию о языке. Форма слова, связи слов в предложении лежат на поверхности и легко наблюдаемы. В каждом простом предложении русского языка есть слово, как правило единственное, на котором держится все предложение. Если его убрать, то предложение развалится. Иногда превращается в другое предложение. Таким словом (центром) предложения часто оказывается глагол. Но далеко не только глагол. Предложение "Ему некогда работать" держится на слове "некогда", предложение "Ему время идти" - на слове "время", "Здесь не пройти" - на слове "не", "Нет возможности" - на слове "нет", "Ему весело шагать" - на слове "весело" и т.д. Центром может быть и пустое слово (глагол "быть" в настоящем времени, как правило, опускается). Например, "Он в Москве".

Каждое предложение является суперпозицией функций. Центр предложения - это самая внешняя функция суперпозиции. Если суперпозиция имеет вид $f_1(f_2(x_1,\ldots,x_n))$, то выбрасывание центрального слова f_1 преобразует исходное предложение в другое - $f_2(x_1,\ldots,x_n)$.

Описание синтаксиса языка прежде всего должно содержать описание обращений к каждой функции. В русском языке форма обращения к функции определяется совокупностью грамматических типов ее аргументов. Информация о типах аргументов является основной информацией, необходимой для синтаксического анализа. Любая часть речи в любой грамматической форме, а также целые предложения, могут служить аргументами для определенного множества слов.

Синтаксис неразрывно связан с семантикой, и поэтому иногда целесообразно в его описание включать часть семантической информации. Например, слово "радость" может иметь следующее синтаксическое описание:

радость = (кого | чьих , что= предл= у) | (кому | для кого x , делать y) caus ($S_0 y(x)$, радость $(x,*)$)

Такое описание позволяет два различных способа использования слова "радость" свести к одному из них: "Для нее радость помогать людям" = "То, что она помогает людям, доставляет ей радость".

Некоторые функции могут иметь два (или более) предложения в качестве аргументов. Такими функциями являются составные союзы.

Опыт построения синтаксических анализаторов русского языка выявил необходимость классификации простых предложений русского языка с целью повышения эффективности анализатора. Наиболее сложная часть анализа - нахождение центрального слова предложения. Любая часть речи может быть центром предложения. Именно она и определяет тип предложения. Построенная ниже классификация содержит семь основных синтаксических типов предложений, определяемых глаголами, существительными, прилагательными, наречиями, предлогами, союзами и символом "-". Причастия относятся к группе прилагательного, деепричастия образуют вложенные суперпозиции с общим первым аргументом, следовательно, нового типа предложений не образуют. Каждому типу соответствует своя группа предложений, которая делится на подгруппы, которая, в свою очередь, может делиться тоже.

Первая группа - глагольная. Она состоит из трех подгрупп. Каждая подгруппа определяется формой глагола. Глагол может быть в активной, пассивной форме или в форме инфинитива. Каждая группа представлена набором типичных для этой группы предложений.

I.1. а) Ему следует подождать. Есть кому работать.
б) Его знобит. Ее лихорадит. Лодку перевернуло.
в) Светает. Вечереет. Морозит.
I.2. а) Считают, что он умен. У них говорят, что ...
б) Молнией зажгло сарай.
I.3. а) Лес шумит. Собака лает.
б) Он читает книгу. Иван повесил картину.
в) Петр сидит смотрит телевизор. Она лежит читает.
2.I. а) Ему хочется петь.
б) Ему работается.
2.2. Дом строится рабочими.
2.3. а) Говорилось, что он массон.
б) Курить запрещается. Предлагаются путевки.
в) Воды убывает.
3.I. Быть ему космонавтом.
3.2. Идти ему.
3.3. Он бежать!

Центром предложения второй группы является существительное. Общий смысл предложений этого типа - $func_0(x)$ - "имеет место быть то, что названо данным существительным".

I.I. а) Дождь. Ветер. Час дня. Пора ехать.
б) Дом с мезанином. Часы с боем. Человек с ружьем.
в) Цветов! Народу!
I.2. а) Ему благодарность от командования.
б) Для нее радость помогать людям.
I.3. Платье в горошек. Усы колесом.
2.I. У него машина. Он с книгой.
2.2. С ним обморок. У нее истерика.
2.3. У него желание ехать.
3.I. а) Он в Москве.
б) Он в бригадирах.
в) Он в шинели. Пальто в грязи. Она в слезах.

3.2. Он в обмороке, в дружбе, в полете, под арестом,...

3.3. Его счастье в труде. В публике смех. В семье горе.

Прилагательные образуют предложения вида: A(S) , где S - существительное, A - прилагательное. Причастие рассматривается как отглагольное прилагательное.

I.I. а) Цветок красный. Платье голубое с желтым в крапинку отливом.

б) Женщина в белом.

I.2. а) Ягоды зелены.

б) Он благодарен.

в) Он слаб здоровьем.

I.3. а) Дом построен.

б) Им довольны. В школе удовлетворены.

2.I. Иван склонен преувеличивать. Он способен быть космонавтом.

2.2. Его рады видеть.

Четвертую группу предложений образуют наречия. В роли наречия может находиться существительное с предлогом.

I.I. Он запанибрата с Иваном Васильевичем. Он сродни ей. Твое одиночество веку под стать.

I.2. Ему весело кататься.

I.3. а) Он быстро бежит.

б) Возможно, что он придет.

2.I. Он говорит с уважением (без уважения).

2.2. В разъяснение тезиса он привел ряд интересных фактов.

2.3. Он был весь белый от снега. Он пропустил занятия по болезни. За отсутствием простой листок на гербовой.

3.I. а) Видно следы. Набрано грибов.

б) У ребят налольенс рыбы. У них договорено о поездке.

в) В бригаде решено пахать.

3.2. Морозно. Ветрено. Дождливо.

Пятую группу предложений образуют предлоги. Но лишь некоторые из них. Многие предлоги служат средством присоединения аргументов и не образуют целых предложений. Однако сложные предлоги, имеющие два аргумента, образуют целые предложения. Семантически они не отличаются от предложений, порождаемых существительными в роли наречия.

I) Сложные предлоги, содержащие в своем составе первообразный предлог и существительное, такие, как: вследствие, с помощью, в условиях, по причине, в отношении и т.п. имеют единую схему описания:

сложный предлог = (кого-чего x , предл y)предл

(Вследствие засухи наступил голод. Иван повесил картину с помощью лестницы.)

Небольшое число предлогов этого типа (в противовес, в противоположность, в контраст, не в пример) имеют синтаксическую структуру вида

сложный предлог = (кому-чему x ,предл y)предл.

2) Сложные предлоги, содержащие существительное и два первообразных предлога (<предлог> <существительное >< предлог >) первым аргументом имеют существительное в той грамматической форме, которая требуется для второго первообразного предлога. Вторым аргументом является целое предложение.

В отличие от = (кого-чего x ,предл y)предл;

в связи с = (чем x , предл y)предл;

в направлении к = (кому-чему x ,предл y)предл.

3) Отглагольные предлоги воспроизводят связи тех глаголов, от которых они образованы.

Несмотря на = (кого-чего x ,предл y)предл;

судя по = (кому-чему x,предл y)предл.

Шестую группу предложений образуют союзы. Все предложения этой группы являются сложными в смысле грамматики русского языка. Синтаксический анализ таких предложений в принципе мало чем отличается от анализа обычных конструкций языков программирования (условных операторов, циклов). Но если в языках программирования таких конструкций единицы, то здесь их десятки, и каждый союз требует отдельного описания, иногда очень громоздкого (например, союз "чем"). Наибольшая сложность синтаксического анализа предложений этого типа связана с тем, что одинаковые части различных аргументов в предложении не дублируются и синтаксический анализатор обязан их пополнять.

Седьмая группа содержит предложения, центром которых является символ "-" . Предложение этого типа состоит из двух частей и имеет вид: f - g . Эта группа предложений естественно делится на две

большие группы. К первой группе относятся предложения, в которых символ "-" заменяет некоторую конкретную функцию, однозначно восстачавливаемую по синтаксическому типу своих аргументов f и g. Эти предложения могут быть преобразованы в предложения первых шести групп. Ко второй подгруппе относятся предложения, в которых одна из частей (f или g) является аргументом другой части, но как бы оторвана от основной части предложения. Происходит резкое разделение предложения на рему и тему, противопоставление главного (с информационной точки зрения) второстепенному. При этом аргумент может быть преобразован в синтаксическую форму, отличную от той, которая необходима для построения синтаксически правильных предложений.

I.I. а) f^- = (кто-что x, делать y) предл. Символ "-" заменяет слова " в том", "в том. чтобы", "заключается в том, чтобы". Например: Главное лечение - лежать. Здесь работать - большая честь.

б) f^- = (кому-чему x, делать y) предл. Заменены слова: "надо", "необходимо", "должен". Например: Ему - ехать.

в) f^- = (какой x, делать y) предл. Заменены слова: "для того, чтобы". Например: Молод - гулять.

I.2. а) f^- = (кто-что x, кем-чем y) предл. Смысл символа "-": x имеет вид y, x сделался y, x подсбен y. Например: Волосы - бобриком. Усы - колесом. Слезы - ручьем.

б) f^- = (кому x, кто-что y) предл. Символ "-" заменяет слова: "необходимо дать (отдать)". Например: Пятилетке - ударный труд.

в) f^- = (кто-что x, с кем y) предл. Эквивалент: у x есть y. Например: Друзья - с ним. С ним - обморок.

I.3. Обе части выражения $f - g$ имеют одинаковый грамматический тип. Символ "-" заменяет слова: "значит", "есть", "является". Например: Иван - плотник. Курить - здоровью вредить. Красное - это черное. Идут дожди - будет урожай.

Обе части выражения $f - g$ имеют различный грамматический тип.

2.I. Аргумент сохраняет свою синтаксическую форму. Примеры:

а) Убить двух зайцев - вот чего он хочет. Они не пройдут - теперь он это знает.

б) Мужчина - высокого роста. Ему - благодарность. Памятник - Маяковскому.

в) Цветок - красный. Письмо - написано.

г) Он лжет - это непростительно. Лекции начались - это хорошо. Сидеть молча - плохо. Деньги - кстати. Гулять - поздно.

2.2. а) Глагол в личной форме преобразуется в отглагольное существительное: $f \to s_0 f$. Например:

Непростительно, что он лжет → Его ложь - это непростительно.

б) Инфинитив преобразуется в отглагольное существительное. Например:

Только успевай работать → Работы - только успевай.

в) $f(x_1, x_2, \ldots, x_n) \to s_1 f(x_1, x_2, \ldots, x_n)$. Например:

Понятно, что человек в одиночестве скучает. → Человек, скучающий в одиночестве - это понятно.

§ 2. Базисные функции

Выполнение суперпозиции функций, представленных словами русского языка, порождает суперпозицию базисных функций семантического языка (СЯ). Базисная функция это такая функция, которая определена на любых объектах и дальнейшему толкованию в рамках СЯ не подлежит. Основной набор этих функций был описан в работе [II]. Здесь приведены лишь некоторые из них. Они обозначаются латинскими буквами, потому что соответствующие слова русского языка из-за неоднозначности (вне контекста) могут внести путаницу. Более конкретные функции названы близкими по смыслу русскими словами.

I. $func_0(x)$ - имеет место быть x;
$func_0$ (дождь) - идет дождь; $func_0$ (ветер) - дует ветер;

2. incep(x) - x начинается, fin(x) - x заканчивается;
cont(x) - x продолжается;

3. aus(x,f) - x делает так, чтобы f;
caus(x, incep спит (y)) - x усыпляет y (x делает так, чтобы y начал спать); caus(x,fin спит (y)) - x будит y;

4. copul(x,y) - x является (есть) y;

5. result(f) - результат f;
result(ложится (x)) - x лежит;

6. $oper_1(x,f)$ – x совершает f ;
$oper_1(x,$полет$)$ – x совершает полет; $oper_1$ (x , уважение) – x испытывает уважение; $oper(x,S_of)$ – x совершает деяние f;
7. sing(x) – элемент множества x ;
8. mult(x) – множество элементов x;
9. loc(x,y) – x находится в y ;
10. hab(x,y) – x имеет y;
11. oploc(x,y) – x воспринимается y;
12. lab(x,y) – x подвергается действию y ;

lab (дерево,молния) – молния ударила в дерево.

Другие базисные функции, смысл которых будет пояснятся по ходу изложения: var(x,f), fact(f), prepar(x,f), degrad (x), mloc(x,y), uzor(x,y), cor(f),deb(f), ver(f), aspect(f), poss(f), repet(f), norm(x), magn(x) . К базисным операциям относятся также операции сравнения чисел: $<$, $\leq$, $=$, $\geq$, $>$.

Особую роль в СЯ играет функция caus. С самой общей точки зрения любое событие каузируется кем-то или чем-то. Поэтому любое действие в СЯ можно было бы описывать в виде caus(g,f) , где g – либо суперпозиция функций, либо, когда не известен субъект каузирования, символ * . Но СЯ, в частности русский язык, способен различать ситуации каузирования-некаузирования. Поэтому предложение "У него есть деньги" на СЯ следует переводить как hab (он,деньги), а предложение 'ему есть деньги' (например, на почте) как caus(*, hab (он, деньги)).

Каждая базисная функция выражает некоторый абстрактный смысл вне временных и пространственных рамок. Для того чтобы использоваться в языке, функция и/или ее аргументы должны быть конкретизированы. Функции конкретизируются заданием времени (прошедшее, настоящее, будущее) и признака завершенности-незавершенности. Для этой цели используются обозначения: perf, imperf, fut . Например, выражение caus(x, incep $func_0(y)$) имеет смысл: x создает(вообще, всегда) y или x есть то, что создает(вообще,всегда) y . Следующие выражения конкретны: imperf caus(x,incep $func_0(y)$) – x создавал y , perf caus(x,incep $func_0(y)$) – x создал y , fut perf caus(x,incep $func_0(y)$) – x создаст y , fut caus(x,incep $func_0(y)$) – x будет создавать y . Для конкретизации настоящего времени необходимо хотя бы у одного аргумента указать артикль – определенный или неопределенный: def x, indef x . Инфинитив функции $f(x_1,\dots,x_n)$ записывается в виде $f(*,x_2,\dots,x_n)$. Многократность-единичность действия выражается при помощи функций mult(f), sing(f) . Например, sing (стрелять) – стрельнуть, mult (стрелять) – многократно стрелять. Эти функции используются и для других целей.

Запись S_if является названием i-го аргумента функции f ; записи $S_if(x_1,\dots,x_n)$ соответствует причастный оборот или выражение: x_i , который $f(x_1,\dots,x_n)$ или тот(та, то, те), который $f(x_1,\dots,x_n)$ (если i-й аргумент отсутствует). Например, S_1 продавать=продавец, S_2 продает(Иван,книга) – книга, которую продает Иван (книга, продаваемая Иваном), S_2продает(Иван, *) – то, что продает Иван.

Выражение $f_1(f_2(x,\dots,x_n),y_1,\dots,y_m)$ всегда может быть переведено на русский язык деепричастным оборотом. Суперпозиция съела(бегала(собака,внутри(двор)),мясо) соответствует русскому предложению 'Собака, бегая во дворе, съела мясо'; суперпозиция съела(S_1 бегала(собака,внутри(двор)),мясо) – предложению 'Собака, которая бегала во дворе, съела мясо'.

Каждая функция f кроме обозначения может иметь имя, которое в СЯ совпадает с записью S_of . В русском языке именам соответствуют отглагольные существительные. Например, S_o уважать=уважение. Полного соответствия между СЯ и русским языком нет. В русском языке есть имена S_of , для которых нет соответствующих им глаголов f. Например: буря, гроза, склероз. Имена функций позволяют различать вызов по значению и вызов по наименованию. Это различие ведет к разным способам выполнения предложения и тем самым к различным его смыслам.

Приведем примеры использования базисных функций и их приблизительные переводы на русский язык.

1) S_o* – имя произвольного действия: деяние, дела(людей, человека); явление (природы);
sing S_o* – дело, поступок(человека);
oper(x,S_o*) – x совершает деяние (дела,дело,поступок), x действует;
$S_o$oper(*,S_o*) – деятельность;
oper(x,S_o*(y)) – x совершает деяние, связанное с y ;

oper(x, лов(рыба)) – x ведет лов рыбы = x ловит рыбу;
oper(x,S_0* (математика)) – x занимается математикой;
oper(x, S_0* (игрушки)) – x занимается игрушками;
S_1 oper(*,S_0* (лошади)) – лошадник;
S_1 oper(*,S_0* (голуби)) – голубятник;
S_1 oper(*,S_0 *(планер)) – планерист;
S_0 oper(*,S_0* (планер)) – планеризм;
S_2 oper(хирург, S_1S_0*(*)) – типичное название того, с чем связаны деяния хирурга = хирургия;
S_1 oper(* , термическая(обработка)) – термист;
S_1 oper(* , марафонский бег)) – марафонец;
S_2 oper (террорист, *) – террор;
S_2 oper (шпион, *) – шпионаж.

Описание слова 'шпионаж' (а значит и его смысл) по отношению к слову 'шпион' отличается от описания слова 'хирургия' по отношению к слову 'хирург'. В одном ряду со словом 'шпионаж' находятся слова 'террор , бег, копирование' и т.п. Слово 'хирургия' образует второй ряд, к которому относятся слова 'планеризм, голубеводство, педагогика, магия' и т.п.

2) Функция copul(x,y) своим первым аргументом имеет имя объекта, вторым – имя объекта или прилагательное.

copul(x , скаредный) – x скаредный;
S_1 copul(* , скаредный) – скаред;
S_1 copul(*, чужой) – чужак;
S_1 copul(*, крепкий) – крепыш;
copul(x, подсобный(рабочий)) – x – подсобный рабочий;
S_1 copul(* , подсобный(рабочий)) – подсобник;
S_1 copul(* , хронический(больной)) – хроник;
S_1 copul(* , строгий(выговор)) – строгач;
S_0 copul(* , скаредный) – скаредность;
S_0 copul(* , равный) – равенство;
S_0 copul(* , лирика) – лиризм;
S_0 copul(* , добрый) – доброта;
incep copul(x,y) – x становится y;
perf incep copul (x,y) – x стал y;
incep copul (x, белый) – x белеет (становится белым);
incep copul (x, рабочий) – x становится рабочим;

oper(x, S_0 copul(*,y)) – x совершает действие, связанное со свойством y ;
oper(x, S_0 copul(* , белый)) – x белеет (вдали);
oper(x, S_0 copul(* , бесвкусный)) – x проявляет бесвкусицу;
oper(x, S_0 copul(* , злой)) – x злится;
oper(x, S_0 copul(*, жених)) – x женихается.

3) temp(x,y) – x происходит во время y ;
S_0 temp – время;
S_2 temp(x,*) – момент или интервал времени;
S_0 temp(*, ночь) – ночевка;
S_0 temp(*, зима) – зимовка;
oper(x,S_0 temp(*,y)) –x проводит время y ;
oper(x, S_0 temp(*, зима)) – x совершает зимовку = x проводит зиму = x зимует;
oper (делает(x ,игрушки), S_0 temp(*,лето)) – x проводит лето, делая игрушки.

4) Функция mult(x) (или mult x) определена как на объектах, так и на действиях и называет либо множество однотипных объектов, либо многократность действия.

mult (человек) – человечество;
mult (горошина) – горох;
mult (террор) – терроризм;
mult (купец) – купечество;
mult (хлестнуть) – хлестать;
perf варит (x,mult (каша)) – x наварил много каш;
S_1 copul (mult(*),громадный) – типичное название множества объектов, обладающих признаком 'громадный' = громадье;
S_1 copul(mult(*), периодический) – периодика (газеты,журналы);
S_1 copul(mult(*), растение) – растительность;
S_1 copul(mult(*), голый) – голытьба;

Функция sing(x) является обратной к функции mult(x) :
sing(mult(f)) = f .
sing (горох) – горошина;
sing (студенчество) – студент;

5) Функция usox(x,y) определена на объектах и действиях и имеет смысл: y используется для x .

cause(x, usox(y,z)) – x использует z для y ;

S_1 caus(*,usor(*,револьверный(станок))) - типичное название того, кто использует резольверный станок= револьверщик;

S_1caus(*,usor(*, термическая(установка))) - термист;

S_1oper(*, термическая(обработка)) - термист

(тем самым слово 'термист' имеет два смысла).

6) S_2 loc(f,*) - типичное название места, где происходит f.

S_2loc(copul (течение,быстрое), *) - быстрина;

S_2loc(copul(*, окрестный), *) - окрестность.

7) Функция magn(x) определена на именах объектов и действий и достаточно близко соответствует русскому слову 'очень'; функция antimagn - антоним функции magn .

S_0copul(*, желтый) - желтизна;

antimagn(S_0 copul(*,желтый)) - желтинка;

magn(дом) - домина, домище;

antimagn(шум) - шумок;

antimagn(стог) - стожишко;

antimagn(петь) - напеть;

perf magn(критиковать) - раскритиковать.

8) caus(x,func$_0$(S_1 copul (* , подобен(базар)))) - x делает так, что имеет место нечто, подобное базару = x базарит;

caus(x, incep copul(y,калека)) - x калечит y;

9) Первым аргументом функции bon(f,x) может быть как объект, так и действие, вторым - объект типа 'человек'. Смысл этой функции: f является хорошим для x .

bon(result(f),x) - хороший для x результат в f ;

antibon(result(f),x) - плохой для x результат в f ;

perf caus(лечит(x,y).antibon(result (лечит(x,y)),y)) - x , леча y , сделал так, что результат лечения для y - плохой = = x долечил (!) y ;

10) Функция fact определена на тех объектах, которые, существуя, могут не проявляться, не исполняться, не соблюдаться: чувство, надежда, мечта, тишина, граница, обычай, традиция, симфония, свойство и т.п.

oper(x,уважение) - x испытывает уважение= x уважает;

oper(x,S_0fact (уважение)) - x проявляет уважение;

oper(x,S_0copul(*, хитрый)) - x совершает хитрость= x хитрит;

oper(x,S_0 fact(S_0 copul(* ,хитрый))) - x проявляет хитрость;

caus(x, fin fact(y)) - x нарушает y (границу, тишину, обычай,..)

oper(x,S_0 fact (симфония)) - x исполняет симфонию.

11) Функция result(f) определена на множестве действий и определяет типичный результат действия f . Для нее всегда выполняется тождество:

result(caus(x, incep f))= f .

12) У функции hab(x,y) первым аргументом является имя объекта, вторым - одна из характеристик объекта или объект. Функция hab(x,y) выражает смысл: x имеет y .

S_1 hab (дом,мезонин) - дом с мезонином;

hab (Иван,деньги) - у Ивана есть деньги.

§ 3. Объекты

Объект - это абстрактное понятие, в естественных языках которому соответствует понятие существительного, называющего некоторый физический объект (в отличие от действия). В семантическом языке объект A представляет собой неупорядоченную совокупность M_A характеристических функций:

$$M_A = (f_1^A, f_2^A, \ldots, f_{m_A}^A)$$

Интуитивно каждый объект A является представителем множества тех конкретных объектов, на каждом из которых определены все функции f_1^A . Представление объекта в виде множества M_A позволяет определить частичную упорядоченность на множестве всех объектов: $A_1 \leq A_2$, тогда и только тогда, когда $M_{A_1} \subseteq M_{A_2}$. Если $A_1 \leq A_2$, то A_1 соответствует родовому понятию по отношению к A_2 , т.е. A_2 более конкретен чем A_1 . При этом функции множества M_{A_1} в наборе M_{A_2} могут иметь суженную область значений. Если $A_1 < A_2$ и не существует A_3 , такого, что $A_1 < A_3 < A_2$, то A_1 представляет видовое понятие по отношению к A_2 . Один из таких объектов в семантическом языке обозначается через gener(A_2) .

При описании объекта A задается имя каждой функции f_i^A , область ее значений и (единственное) имя объекта A . Это описание аналогично описанию вида в языках программирования и служит в СЯ той же цели - для задания информации об используемых в речи объектах. Имя объекта есть идентификатор (последовательность си-

мволов), однозначно указывающий на данный объект. В этом проявляется существенное различие между СЯ и ЕЯ. Если A_1 – имя объекта, представляющего видовое понятие по отношению к объекту A_2, то объект A_2 описывается в виде

$$A_2 = (A_1 ; f^{A_1}_{i_1} = B_1, \ldots, f^{A_1}_{i_k} = B_k ; i^{A_2}_1, \ldots, f^{A_2}_{m_{A_2}})$$

$$(1 \le m_{A_2} ; \quad 0 \le k),$$

где B_1 – описание суженной области функции $f^{A_1}_1$.

В качестве примера приведем описания нескольких объектов СЯ, приближенно соответствующих понятиям: "физический объект", "живой", "животное", "человек" и др. Именами характеризующих функций являются идентификаторы, составленные из русских слов, но никакого отношения к русскому языку они, конечно, не имеют. Если областью значений некоторой функции является множество целых чисел i ($-10 \le i \le 10$), то имя функции помечается символом i, если множество любых чисел – символом a. Эти символы определяют относительные и абсолютные значения. Например, температура –5 определяет понятие "холодный", температура a =27 соответствует выражению: температура равна 27^0. Функция, содержащая альтернативы $f_1, f_2, \ldots, f_n$, обозначается через $(f_1 \mid f_2 \mid \ldots \mid f_n)$.

Физобъект =
(физсостояние i, температура (i | a),
цвет=((алый|...|белый|...|черный|...),интенсивность i),
размер=(линвеличина=(высота(i|a),ширина(i | a),длина(i|a)),
объем(i | a),вес(i | a),
положение=(верт | гориз | висит),
состав=(состояние=(газ | жидкость | густой |твердый i),
вещество=(металл=(железо,сталь,...) | вода | земля | гранит...
...)),
внешвид=(форма=(правильная=(шар,куб,пирамида,...),неправильная)),
свойства=(гибкость i ,хрупкость i ,твердость i ,...),
поверхность=(физобъект;скользкость i ,гладкость i ,углубление i),
покрытие=(физобъект),подобен i =(физобъект),
части=(верх=физобъект,низ=физобъект,перед=физобъект,середина=физобъект,центр=физобъект,зад=физобъект,бок=физобъект),
возраст(i | a),красота i ,реальность=(действ | воображ))

живой =
(физобъект;пища=(физобъект),здоровье(i|болезнь),активность i ,местообитание=(место))

животное=
(живой;пол=(м | ж),психосостояние=(бешеный | норм),восприятие =(зрение i,слух i,обоняние i,осязание i,тактилчувствительность i),память i,умспособности i,
домашний=(да | нет),
язык=(блеяние | ... | речь |...|шипение...),
физсостояние=(бодрость i,удовлетворение i,голод i),
характер=(агрессивность i,настроение i),
половлечение i,
способпередвижения=(ползание | плавание | полет | ходьба ...),
отношение i =(животное),
родственники=(mult(животное)),
храбрость i,гордость i,грубость i,доброта i,страх i,интерес i,сила i,общительность i,свобода i ,сон i,худоба i),

человек =
(животное;верх=(голова=(верх=(макушка,волосы),середина=(череп=(середина=мозг)),перед=(лицо=(верх=лоб,середина=(нос, глаза,щеки),низ=(рот=(губы,зубы),подбородок,борода=(да,нет)))
зад=затылок,бок=(уши,бакенбарды=(да | нет))...)),
середина=(туловище=(верх=шея,...),...),
бок=(руки=(левая,правая),плечи ,...),...,
язык=речь,способпередвижения=ходьба,умспособности=10;
фио=(фамилия=идент,имя=идент.отчество=идент),
национальность=(абхазец|...|грузин|...|русский|...|якут),
местожительство=(страна=идент,город=идент,адрес=идент),
профессия=(агроном|...|слесарь|...|ямщик),местоработы,
специальность=идент,
образование=(высшее i,специальность=идент),
характер=(темперамент i,аккуратность i,благородство i,воля i, склонность i= mult (действие),искренность i,упрямство i, хладнокровие i,черствость i,честность i,рассеянность i, гордыня i,жадность i,эгоизм i,самолюбие i,хвастовство i),

чувство=(симпатия i,реакция i),
оптимизм i,юмор i,работоспособность i,счастье i, ум i,легкомыслие i ,трезвость i,
вероисповедование=(католик | ... | неверующий),
характеристикадействий=(намеренность i,усилие i,риск i,доброжелательность i,ожидание i,трудность i,виновность i, одобрение i,добросовестность i,целесообразность i,тщательность i,тайность i,причина=(действие),предварительность i)

отец=
(человек;пол=м,дети= mult (человек),отношение i=(дети))

взрослый человек=
(человек;возраст= >16 лет)

мужчина=
(взрослый человек;пол=м,...)

эгоист=
(человек;пол=м,эгоизм= >5)

эгоистка=
(человек;пол=ж,эгоизм= >5)

альтруист=
(человек;пол=м,эгоизм= < -5)

Для того чтобы был понятен смысл идентификаторов, используемых в описаниях объектов, приведем ряды слов русского языка, характеризующие некоторые из этих идентификаторов в порядке убывания индекса i .

отношение i =(...ласка,забота,доброта,милосердие,деликатность,любезность,вежливость,учтивость,равнодушие,нелюбезность,грубость,жестокость,...);

настроение i =(...радость,веселость,мажорное настроение,равнодушие,минорное настроение,грусть,печаль,горесть,тоска,мрачное настроение,...);

удовлетворение i =(эйфория,...,блаженство,наслаждение,удовольствие,удовлетворение,равнодушие,кручина,скорбь,горе,страдание,мучение,...);

искренность i =(...искренность,простодушие,...,скрытность,лукавство,хитрость,притворство,фальшивость,лживость,лицемерие, ...);

реакция i =(...,ярость,бешенство,гнев,негодование,возмущение, злость,сердитость,досада,волнение,...,восхищение,восторг, упоение,экзальтация,...);

гордыня i =(...,гордыня,спесь,чванство,надменность,высокомерие, зазнайство,самодовольство,себялюбие,...,скромность,...);

гордость i =(...,гордость,достоинство,самолюбие,...,стеснительность,застенчивость,конфузливость,униженность,...);

предварительность i =(...,заранее,предварительно,...,перед,..., вслед,...,после,...).

Аналогично каждой из характеристик описанных выше объектов сопоставлен соответствующий ряд русских слов. При построении этих рядов были использованы словари антонимов [12] и синонимов [13] .

Примеры перевода с семантического языка на русский язык:

opmloc (x, зрение(y)) - y видит x ;
opmloc(x , (слух(y)) - y слышит x ;
caus(x , видит(x,y)) - x смотрит y ;
caus(x,opmloc (y, память(x))) - x запоминает y ;
настроение(x) = 4 - x веселый;
настроение(x) = -3 - x печальный;
удовлетворение (x) = 3 - x доволен;
бодрость (x) = < -2 - x усталый;
caus(x, opmloc (fact (хвастовство(x) = >5),y)) - x делает так, что y воспринимает проявление хвастовства x -а = x хвастается перед y ;
caus(x, opmloc(fact (гордыня(x) = 4),y)) - x высокомерен с y.

§ 4. Операции над объектами

Так как каждый объект представляет собой множество характеристических функций, то нетрудно определить набор бинарных операций, которые по двум объектам строят новый объект. Здесь мы рассмотрим одну простую операцию, которая имеет прямое отношение к русскому языку и определяет признаковые прилагательные.

Если функция f входит в набор функций объекта в , то будем называть ее признаком объекта в . Запись f(в) = y определяет бинарную операцию A(z,в) на объектах z и в , где z - объ-

ект, содержащий одну функцию f , т.е. z=(f = y). Эта операция по имени объекта B строит имя нового, более конкретного объекта, отличающегося от B лишь тем, что функций f на нем принимает значение y , которое является либо идентификатором, либо числом, либо диапазоном чисел (например, >2). Такая запись вполне удовлетворительно выражает смысл операции A , но для того чтобы приблизить СЯ и по форме к ЕЯ (с целью более тонкого анализа процесса словообразования) построим имя Afy функции, которая получается из операции A(z,B) как результат смешанного вычисления при заданных f и y : Afy(B)≡ f(B) = y . Имя Afy - полный аналог прилагательного в ЕЯ, поэтому запись Afy будем называть прилагательным семантического языка. Например, прилагательное 'A настроение 4' можно перевести на русский язык словом 'веселый': A настроение 4(человек)=веселый человек. Это прилагательное в СЯ определено только на объектах, являющихся конкретизациями объекта 'животное'.

Из чисто формальных соображений имеет смысл ввести всюду определенную (на объектах, действиях и т.п.) тождественную функцию: A(x) =x . Функции S_0A , Adv (см.ниже), образованные от функции A , будем считать также всюду определенными, тождественными функциями.

Прилагательные принципиально отличаются от существительных гораздо большей функциональностью (грубо говоря, разбросанностью значений). Если описанные выше объекты, даже такие сложные как 'человек', поддаются компонентному анализу [5] - разложению на характеризующие компоненты, то прилагательные этим свойством, как правило, не обладают. Дело в том, что одно и то же имя признака - прилагательное - может называть различные признаки. Например, при описании объектов 'животное' и 'отец' был выбран один и тот же идентификатор 'отношение 1'. Тем самым в СЯ появляется возможность создания прилагательного 'A отношение 1' с двумя значениями: отношение к кому-то или отношение к своим детям. Для ЕЯ эта ситуация типична. Однако это не приводит к каким-либо неудобствам, так как предложение снимает эту неоднозначность: прилагательное является всегда однозначной функцией существительного. Семантика существительного в ЕЯ (его описание в ЕЯ) гарантирует однозначность прилагательного. Однако вне контекста прилагательное многозначно (см. примеры в работе [9]).

Пусть признак f входит в объекты A_1 и A_2. Возможны два типа многозначности прилагательного: 1) A_1 и A_2 несравнимы; 2) A_1< A_2 (тем самым в объект A_2 признак f входит дважды). В первом случае вычисление функций Afy(A_1), Afy(A_2) , т.е. выбор по A_1 соответствующего признака не представляет труда. Во втором случае вычисление функции Afy(A_2) может быть осложнено проблемой выбора: неизвестно, на какой признак указывает имя Afy . Но в ЕЯ действует неписанное правило: прилагательное указывает на тот признак, который принадлежит более конкретному понятию, и не принадлежит более общему, хотя признак более общего понятия полностью из поля зрения не исчезает, а остается на втором плане. Например, словосочетание 'добрый отец' указывает в первую очередь на отношение отца к детям, хотя остается и смысл 'отец - добрый человек' .

Если прилагательное x указывает на какой-либо признак f и объект A этого признака не содержит, то прилагательное x как функция существительного легко может быть доопределено на объект A без какой-либо потери однозначности. Поэтому с легкостью возникают такие понятия как 'легкая радость', 'тяжелая тоска', 'черная душа', 'серые мысли', 'свежая идея' и т.п.

Имя Afy образуется как результат смешанного вычисления бинарной операции A(z,B) , где z= (f = y) - простейший объект, содержащий всего одну функцию f с конкретным значением y . Но тем не менее этот объект может иметь свое собственное имя, и чтобы использоваться в языке, он обязан иметь имя. В СЯ это имя совпадает с обозначением S_0Afy . В русском языке такие имена образуются, как правило, при помощи суффиксов: -ость, -ство, -и, реже (устаревшее словообразование) -изн, -от: S_0A цветкрасный - краснота; S_0A реакция5 - бешенство; S_0A цветбелый - белизна; S_0A гордыня4 - высокомерие; S_0A реакция5 - ярость.

В СЯ имена Afy и S_0Afy связаны тождеством:

S_0Afy(x) = S_0copul(x,Afy(x)) .

Теперь можно расширить области определения функций Afy ,распространив их на множество всех имен S_0Afy .

A гордыня4(S_0A реакция6) - высокомерная ярость;

A гордость-2(S_0A искренность-4) - застенчивое притворство,

Тот факт, что имя S_0Afy указывает на признак fy объекта x будем записывать в виде $S_0Afy(x)$. Определим прилагательное Ax, образованное от имени объекта x, как функцию на именах S_0Afy:

$$Ax(S_0Afy) = S_0Afy(x),$$

Аналогичные равенства в русском языке:
отцовская доброта=доброта отца; эгоистическое самолюбие=самолюбие эгоиста; человеческая гордость=гордость человека.

На множестве имен Afy определим функции $Advgz$ (где f,g – признаки объектов, y,z – их значения) как решение уравнения:

(I) $func_0(Agz(S_0Afy(x))) = copul(x,Advgz(Afy)(x))$.

функциям $Advgz$ в русском языке соответствуют наречия, определенные на множестве прилагательных. Например, $func_0$ (A гордость -Z(упрямство4(Иван))) = copul (Иван, Adv гордость-Z(упрямство4)(Иван)). Левой части равенства на русском языке соответствует фраза: имеет место быть застенчивое упрямство Ивана; правой – предложение: Иван застенчиво упрям.

Из уравнения (I) непосредственно следует, что

$$S_0Advgz(Afy) = Agz(S_0Afy).$$

S_0(высокомерно яростный)=высокомерная ярость.

Если в уравнение (I) вместо функции Agz подставить тождественную функцию A, получим тождество:

$$func_0(S_0Afy(x)) = copul(x,Afy(x)).$$

Следует подчеркнуть особую важность механизма словообразования. Без него не может обойтись ни один естественный язык. Чтобы быть адекватным, семантический язык обязан содержать средства словообразования, хотя и более скромные, чем естественный язык. Словообразование в СЯ служит средством уточнения процесса выполнения суперпозиции функций.

Выражения $Afy(x)$, $S_0Afy(x)$ не являются законченными предложениями семантического языка. Первое из них вычисляет лишь ссылку на все объекты x, имеющие признак f со значением y. Второе – порождает ссылку на признак f со значением y объекта x. Эти выражения могут быть составными частями базисных функций, например, таких, как $func_0(x)$, $copul(x,y)$ и т.д., которые образуют законченные предложения СЯ. При выполнении предложения $func_0(Afy(x))$ порождается объект x с признаком f, равным y, при выполнении – $func_0(S_0Afy(x))$ (или $copul(x,Afy(x))$) объекту x присваивается значение y признака f. Таким образом, первому предложению в языках программирования соответствует оператор описания, второму – оператор присваивания.

Предложениям вида: $copul(x,Afy)$ на русском языке, как правило, соответствуют предложения с кратким прилагательным: copul (ягоды, A зеленый1) = ягоды зелены. Поэтому запись Afy назовем кратким прилагательным семантического языка.

Сравнительная степень прилагательного в СЯ выражается последовательностью предложений: $func_0(S_0Afi(x)).func_0(S_0Afj(y)).i \oplus j$, где символ $\oplus$ является одним из символов. $<,\leq,\geq,>$. Например, $func_0(S_0A$ зеленый$i(x)).func_0(S_0A$ зеленый$j(y)).i>j = x$ более зелен, чем y (x зеленее y). Однако вместо трех предложений можно использовать более короткую запись: $func_0(S_0Afi \oplus (x,y))$.

Запись S_0Afy является именем объекта с одним признаком f, который имеет область значений y. Эта область либо содержит единственное значение, либо совпадает с областью значений индекса i. Во втором случае на объекте S_0Afi можно определить функцию $magn$:

$$magn(S_0Afi) = S_0Afi + sign(i).$$

Функция $magn$ по имени S_0Afi строит либо имя S_0Afi+1, если $i>0$, либо имя S_0Afi-1, если $i<0$. В русском языке этой функции соответствуют слова: огромный, большой, полный, сильный и т.п. Например,

S_0A реакция-7 – восторг;
$magn(S_0A$ реакция-7) – большой восторг;
S_0A реакция6 – ярость;
$magn(magn(S_0A$ реакция6)) – огромная ярость;
S_0A удовлетворение6 – блаженство;
$magn(S_0A$ удовлетворение6) – полное блаженство.

Функция $magn$ легко доопределяется на множество прилагательных вида Afi:

$$magn(Afi) = Afi + sign(i).$$

На русский язык выражение $magn(Afi)$ переводится наречиями: очень, слишком, полностью:

A реакция7 – яростный;
$magn(A$ реакция7) = A реакция8 – очень яростный;

magn(A реакция-7) - очень восторженный.

В дальнейшем область определения функции magn будет существенно расширена. Через magn(i)(x) будем обозначать функцию magn(magn ... ... (magn(x))...) .

Аналогичным образом определяется функция anti:

anti(S_0Afi) = S_0Af − i; anti(Afi) = Af − i .

В СЯ выполняются синонимические тождества:

S_0 magnAfi = magnS_0Afi

S_0antiAfi = antiS_0Afi .

Пример. S_0 (очень красный)=большая краснота.

§5. Действия и операции над ними

Действие - понятие семантического языка, аналогичное понятию глагола естественного языка. Формальное определение этого понятия имеет вид

$$f(x_1,x_2,\ldots,x_n) = (\sigma\ ;\ f_1,f_2,\ldots,f_n)\ ,$$

где f - символ (идентификатор) действия, $x_1,x_2,\ldots,x_m$ - аргументы, являющиеся именами объектов, σ - последовательность суперпозиций базисных функций, описывающая сущность действия; f_i - признаки этого действия.

Опишем несколько действий СЯ. Вместо абстрактных идентификаторов действий будем использовать фразы русского языка, семантически близкие к описываемым действиям.

x совершает f = (oper$_1$(x,f);

интенсивно i , скорость i ,продолжительно i ,

совместность i =(мимоходом,...,совместно,...,наравных,главнымобразом),

завершенность= (impf,perf,perffut,fut),

многократность= (mult,sing),

постоянство=(...,постоянно,...,регулярно,...,часто,...,редко,... ...,единожды),

субъективнаяоценка=(вульгарно,...,грубо,...,нейтрально,...,нежно,...,ласково) | торжественно),

направленностьдействия=(вокруг,над,...,под,...,сквозь,через, мимо),

полнотаобъекта=(весь,полностью,...,частично,...),

дополнительно,повторность=(снова | нет),

расположениеобъектов=(...,вперемешку,вплотную,рядом,вместе,... ...,врозь)).

Все характеристики действий можно разделить на две группы: одни из них характеризуют собственно действие (интенсивность, продолжительность и т.п.), другие - характеризуют объект, на который направлено действие: направленность действия описывает его пространственное распространение относительно объекта, полнота объекта указывает, на какую часть объекта направлено действие: на весь объект или на его (большую - меньшую) часть. Если действие направлено на множество объектов, то характеристика 'расположение объектов' определяет, как элементы множества расположены между собой; субъективная оценка определяет стилистическую окраску действия: кушать - есть - жрать; характеристика 'дополнительно' означает, что данное действие является продолжением аналогичного действия и направлено на некоторые дополнительные объекты.

Примеры

perf inoer скорость5(волноваться)=взволноваться;

perf интенсивно5(крикчуть)=закрикнуть;

perf торжественно(благодарить)=возблагодарить;

perf тщательно(белить)=выбелить; (тщательность - характеристика субъекта действия (Adv$_1$fy)):

perf дополнительно(купить)=докупить;

perf мимоходом(прибежать)=забежать;

perf полностью(плескать)=заплескать;

perf предварительно(планировать)=запланировать;

perf вокруг(бежит (x, дом)) - x обежал вокруг дома;

perf снова(воспитывать)=перевоспитывать;

perf частично(мыть)=замыть.

Если ПР - предлог: в, над, под и т.д., то

x находится ПР y = (loc(x,y):

ПР=(внутри | на | над | под | сбоку | ... | расстояние i));

x находится в комнате= (loc(x, комната); ПР=внутри);

x на кухне= (loc(x, кухня); ПР=внутри);

x на крыше= (loc(x, крыша); ПР=на);

x рядом с домом= (loc(x, дом); расстояние=1);

x у дома= (loc(x,дом); расстояние=0);

x был в доме= (perf loc(x, дом); ПР=внутри);
x перемещается из y в z , используя w , по v со скоростью i = (fin loc(x,y). incep loc (x ,z)) , транспорт=((физобъект w|part (x))) , по=(суша | вода | воздух | космос),скорость (i | a)) . Если x - человек, то:
x едет=(перемещается,транспорт=(физобъект),по=суша);
x едет на телеге=(перемещается;транспорт=телега,по=суша);
x идет=(перемещается; транспорт=ноги,по=суша);
x плывет=(перемещается; по=вода);
x плывет на корабле=(перемещается;транспорт=корабль,по=вода);
x летит=(перемещается; по=(воздух | космос));
корабль плывет=(перемещается; по=вода);
x плывет под водой=(caus((loc(x, вода), ПР=внутри),fin loc (x,y).incep loc(x,z))) .

Для того чтобы сократить длину описаний, в тех случаях,когда одно действие отличается от другого лишь суперпозицией σ , будем использовать записи вида
переместился= perf перемещается;
перемещался= imperf перемещается;
переместится= fut переместился;
будет перемещаться= fut перемещается.

Множество действий, также как и множество объектов, легко поддается иерархическому упорядочению. От более общих действий, сужая сбласть значений характеризующих функций, а в случае необходимости, добавляя новые, что для действий не типично, можно переходить к более конкретным действиям.

Операции над действиями. Как было сказано выше, от символов действий можно образовывать имена S_0f, S_1f, $S_1f(x_1,x_2,\ldots,x_n)$. Без какого-либо изменения могут быть определены (признаковые) функции Afy если f - признак некоторого действия. Например, A скорость i>5 (перемещается(x)) - x быстро перемещается. Такие функции на русский язык переводятся наречиями. Если аргументом функций Afy являются имена действия S_0f_1 , то в русском языке им соответствуют прилагательные: A скорость >5(перемещение) - быстрое перемещение. В семантическом языке жесткая скобочная запись позволяет не различать имена признаковых функций, определенных на действиях и на их именах. Однако по аналогии с

именами Adv_i ($i>1$) , вводимыми ниже, будем обозначать их словом Adv_0fy.

Пусть f - признак первого аргумента действия $f_1(x_1,x_2,\ldots,x_n)$. Суперпозиция $g(x_1,S_0Afy(x_1))$ = caus(x_1,fact(S_0Afy (x_1))) (x_1делает так, что проявляется свойство $f(x_1)$, т.е. x_1 проявляет f) характеризует объект x_1 . Определим функцию Adv_1fy равенством:

$$Adv_1fy(f_1(x_1,x_2,\ldots,x_n)) = f_1(g(x_1,S_0Afy),x_2,\ldots,x_n).$$

Имена Adv_1fy семантического языка соответствуют наречиям русского языка (или существительным с предлогом 'с'), которые определяют состояние субъекта действия. Аналогичным образом можно определить функции Adv_1fy , характеризующие i-й аргумент функции f_1 .

Примеры наречий семантического языка и их эквивалентов в русском языке:

Adv_1 настроение6 - радостно;
Adv_1 настроение5 - весело;
Adv_1 настроение-3 - печально;
Adv_1 искренность3 - простодушно;
Adv_1 искренность-7 - лживо;
Adv_1 гордыня4 - высокомерно;
Adv_1 гордыня-5 - скромно.

Adv_1 настроение-3(смотрит(Иван,Петр)) - Иван печально смотрит на Петра;
Adv_1 отношение1(говорит(Иван,Петр)) - Иван учтиво(с учтивостью) говорит с Петром;
Adv_4 стоимость-5(купил(Иван,книга, *,*)) - Иван дешево купил книгу;
Adv_2 тщательность5(строится(дом,рабочие)) - дом тщательно (с тщательностью) строится рабочими.

В СЯ выполняется синонимическое тождество:

$$S_0Adv_1fy(g) = Afy(S_0g) .$$

Примеры аналогичных равенств русского языка:
S_0(сильно любит)=сильная любовь;
S_0(разгульно живет)=разгульная жизнь;
S_0(быстро едет)=быстрая езда;
S_0(усердно работает)=усердная работа.

Каждое действие СЯ по-своему определяет доступ к признакам своих аргументов. Некоторые действия не разрешают доступ даже к признакам первого аргумента. Глагол 'видеть' русского языка -

один из примеров такого действия: * Иван высокомерно видит Петра. В СЯ какие-либо ограничения на доступ к признакам вряд ли уместны, но при переводе с ЕЯ на СЯ их, безусловно, необходимо учитывать. Поэтому при описании действий, соответствующих русскому языку, одной из важнейших характеристик действия является характеристика его аргументов.

Пусть x - имя i-го аргумента функции f. Определим прилагательное $A_i x$ как функцию:

$$A_i x(S_0 f) = S_0 f(*,\ldots,x,\ldots,*) .$$

птичий перелет= A_1 птица(перелет)=перелет птиц; марафонский забег=забег марафонцев; лесные заготовки=A_2 лес(заготовки)=заготовки леса; карандашная зарисовка=A_3 карандаш(зарисовка)=зарисовка карандашом.

Эти равенства следует читать справа-налево, так как правая часть определяет левую.

<u>События</u>. Это понятие соответствует понятию предложения ЕЯ. Событие - это действие во времени и пространстве. Формально оно представляет собой запись $temp(loc(\sigma,l),t)$, где σ - конкретизированная суперпозиция функций, или имя объекта, действия или признака, l - место, t - время. Каждая функция в суперпозиции σ есть имя действия или базисная функция. Иерархии понятий времени и места строятся так же,как и иерархия физических объектов. Любой физический объект может быть указателем места. Но понятие места в языке принципиально отличается от понятия физического объекта, поэтому в СЯ оно выделено как особая характеристика событий.

Аргументы l и t могут принимать значение $*$. В этом случае выполняются тождества:

$$temp(loc(\sigma,l),*) = loc(\sigma,l);$$

$$loc(\sigma,*) = \sigma .$$

Если σ является именем, то

$$loc(\sigma,*) = func_0(\sigma) .$$

§ 5. Система синонимического перифразирования

В работе [II] была построена система перифразирования. Здесь приводится более полная и точная система, в которой учитываются тождества, содержащие прилагательные и наречия. Число наречий и соответственно прилагательных в каждом равенстве может быть более одного или равно нулю.

При переводе с русского языка на СЯ следует различать тип наречия: <u>как</u>, <u>как долго</u>, <u>когда</u>, <u>где</u>, <u>куда</u> и т.п. Наречия типа <u>как</u> определены "на глаголах"; наречия типа <u>как долго</u> - на некоторых глаголах несовершенного вида; остальные наречия являются аргументами глаголов или базисных функций, поэтому при перифразировании не могут быть преобразованы в соответствующие им прилагательные.

Некоторые наречия могут играть двойную роль в предложении. Например, наречие 'своевременно' (<u>как</u> | <u>когда</u>) может быть определено "на глаголах" и может быть аргументом базисной функции temp. Поэтому 'Он своевременно помог ей'=своевременно(помог(он,она))= =он оказал ей своевременную помощь (см. п.3) на с.II2). 'Он своевременно помог ей'= temp (помог(он,она),своевременно)= temp (он оказал помощь,своевременно)='Он оказал ей помощь своевременно'. Наречие 'существенно' имеет тип <u>как</u>, поэтому 'Он существенно помог ей'='Он оказал ей существенную помощь'. Однако 'Он существенно оказал ей помощь' - неправильная фраза и ее невозможно получить как перифразировку предложения. 'Он существенно помог ей'.

Аналогично, 'давно'=(<u>когда</u> | <u>как долго</u>): 'Он давно помог ей'= = temp (помог(он,она),давно)='Он давно оказал ей помощь'. Представление в виде: давно(помог(он,она)) на СЯ невозможно, поэтому невозможна перифразировка: 'Он оказал ей давнюю помощь'. Однако наречия типа <u>как долго</u> определены на некоторых глаголах несовершенного вида: 'Он давно ненавидит пауков'='Он испытывает давнюю ненависть к паукам'.

Приведенные ниже тождества I) - I5) всегда выполняются в семантическом языке. В русском языке допустимо далеко не каждая из этих перифразировок. Во-первых, не каждый глагол f позволяет образовывать имена $S_i f$ $(0 \le i \le n)$. Во-вторых, базисные функции oper, copul и т.д. при некоторых значениях своих аргументов не имеют прямого перевода на русский язык.

I) $Adv_i fy(g(x_1,\ldots,x_n)) = T\ func_0(Afy(S_i 3(x_1,\ldots,x_n)))$.

где T = (perf, imperf, perf fut, fut)- характеристика функции g .

'Народ торжественно встретил его'=торжественно(встретил(народ,

он))= perf $func_0$ (торжественная встреча(народ,он))='Торжественная встреча его с народом состоялась';
'Иван пришел своевременно'=своевременно(пришел (Иван))='Имел место своевременный приход Ивана'.

2) $Adv_0fy(g(x_1,\ldots,x_n)) = T\ func_1(Afy(S_0g(x_1,\ldots,x_n)),x_i)$.

'Он сильно тоскует'='Сильная тоска гложет его';
'Иван жестоко отомстил ему'='Жестокое отмщение Ивана настигло его';
'Он единолично владеет угодьями'= $func_2$ (единоличное владение (он),угодия)='Его единоличное владение распространяется на угодия'.

3) $Adv_0fy(g(x_1,\ldots,x_n)) = T\ oper_1(x_i,Afy(S_0g(x_1,\ldots,x_n)))$.

'Петр блестяще победил врага'='Петр одержал блестящую победу над врагом';
'Иван глубоко уважает Марию'='Иван испытывает глубокое уважение к Марии';
'Он жил весело и разгульно'='Он вел веселую и разгульную жизнь';
'Он долго ловит рыбу'='Он ведет долгий лов рыбы';
'Иван твердо решил писать'='Иван принял твердое решение писать';
'Партизаны упорно сопротивляются противнику'= $oper_2$ (противник, упорное сопротивление(партизаны))='Противник встречает упорное сопротивление партизан'.

4) $Adv_0fy(g(x_1,\ldots,x_n)) = T\ caus(x_i,lab(x_2,Afy(S_0g(x_1,\ldots,x_n))))$.

'Он нежно заботится о ней'='Он окружает ее нежной заботой';
'Иван жестоко наказал его'='Иван подверг его жестокому наказанию'.

5) $Adv_0fy(g(x_1,\ldots,x_n)) = T\ copul(S_0g(x_1,\ldots,x_n),Afv)$;
$Adv_ify(g(x_1,\ldots,x_n)) = T\ copul(S_ig(x_1,\ldots,x_n),Afy(S_ig))$
$(1 \le i \le n)$.

'Он сильно ненавидит Ивана'='Его ненависть к Ивану сильна';
'Иван весело купил тетрадь'='Иван,который купил тетрадь,был веселым покупателем';
'Он пришел своевременно'='Его приход был своевременным';
'Петр блестяще победил врага'='Победа Петра над врагом была блестящей';

'Он жил весело и разгульно'='Его жизнь была веселой и разгульной'.

6) $Adv_ify(g(x_1,\ldots,x_n)) = T\ oper_i(conv_{ij}(g(x_1,\ldots,x_n)),S_0copul(*,Afy))$.

Наречие здесь определено на признаках i-го аргумента. Функция $oper_1$ на русский язык в этом случае переводится словами: проявляет, испытывает. Если признак относится к характеристикам: чувство, настроение или удовлетворение, то используется слово испытывает, иначе - проявляет. Временные характеристики функции g переносится на функцию $oper_i$.
'Он нежно заботится о ней'='Заботясь о ней, он проявляет нежность';
'Иван жестоко наказал его'='Наказывая его,Иван проявил жестокость';
'Он работает с усердием'='Работая, он проявляет усердие';
'Он ловил рыбу с радостью'='Ловя рыбу, он испытывал радость' = 'Ловя рыбу, он радовался'.

7) $Adv_ify(g(x_1,\ldots,x_n)) = temp(T\ copul(x_i,Afv),imperf\ g(x_1,\ldots,x_n))$.

'Он нежно заботится о ней' ='Он - нежный,когда заботится о ней';
'Иван жестоко наказал его'='Иван был жестоким, когда наказывал его';
'Он ловил рыбу с радостью'='Он был радостным, когда ловил рыбу';
'Он работал с усердием'='Он был усердным, когда работал';
'Иван спокойно купил книгу'='Иван был спокойным, когда покупал книгу'.

8) $S_0f = S_0*(S_1f)$;
$S_0f(x) = S_0*(S_1\ copul(x,S_1f))$.

'Продажа заканчивается'='Работа продавцов заканчивается';
'Лов рыбы продолжается'='Работа рыбаков продолжается';
'Агрессия усиливается'='Действия агрессора усиливаются'.

9) perf $caus(x.f(x_1,\ldots,x_n)) \supset$ perf $f(x_1,\ldots,x_n)$;
perf incep $f(x_1,\ldots,x_n) = f(x_1,\ldots,x_n)$
perf caus(Иван, incep висит(картина))=Иван повесил картину $\supset$ perf incep висит(картина)=висит(картина)=картина висит.

10) $\neg(f(x_1,\ldots,x_n)) = \neg f(x_1,\ldots,x_n) \vee f(\neg x_1,\ldots,x_n) \vee$

$$\ldots \vee f(x_1, \ldots, \neg x_n) .$$

Более точно: правая часть равенства должна содержать все подмножества (отрицаний) множества аргументов.

'Неверно, что Иван купил ботинки'='Либо Иван не купил (а продал) ботинки, либо не Иван купил ботинки, либо Иван купил не ботинки';

'Он не должен туда ходить'= $\neg$(должен(он,ходить(туда)))='Либо он может туда не ходить, либо не он должен туда ходить, либо он должен туда не ходить': ($\neg$ должен)(x , f)=может(x , $\neg f$).

II) $\mathrm{magn}\ \mathrm{Adv} f y(g) = \mathrm{func}_0(\mathrm{magn} A f y(S_0 g))$

Аналогичные тождества с функцией magn имеют место для всех остальных тождеств.

'Петр очень уверенно победил'='Имела место очень уверенная победа Петра'='Петр одержал очень уверенную победу'='Победа Петра была очень уверенной';

'Побеждая, Петр проявил большую уверенность'='Петр был очень уверен, когда побеждал'='Действия Петра-победителя были очень уверенными'.

Первое множество перифразировок в этом примере не синонимично второму: наречие 'уверенно' в первом случае рассматривается как $\mathrm{Adv}_0 f y$, во втором - как $\mathrm{Adv}_1 f y$, а каждому из них соответствуют различные способы вычисления значений.

I2) $f(g(x_1, \ldots, x_n), y_1, \ldots, y_m) = g(x_1, \ldots, x_n).f(x_1, y_1, \ldots, y_m)$.

'Иван, ловя рыбу, радовался'='Иван ловил рыбу и радовался'.

I3) $f(x_1, \ldots, x_n) = \mathrm{conv}_{i_1, i_2, \ldots, i_n}(f)(x_{i_1}, \ldots, x_{i_n})$

Здесь $\mathrm{conv}_{i_1, i_2, \ldots, i_n}(f)$ - конверсия функции f .

'Иван купил тетрадь у Петра'='Петр продал тетрадь Ивану'.

I4) $\mathrm{func}_0(\mathrm{Agz}(S_0 A f y(x))) = \mathrm{copul}(x, \mathrm{Advgz}(A f y))$.

Имеет место высокомерная ярость Петра'='Петр - высокомерно яростен'.

Следующие тождества были приведены в § 4 . Их левые и правые части не являются целыми предложениями.

I5) $S_0 \mathrm{Advgz}(A f y) = \mathrm{Agz}(S_0 A f y)$;

$S_0 \mathrm{Adv}_1 f y(g) = A f y(S_0 g)$;

$S_0 \mathrm{magn} A f i = \mathrm{magn} S_0 A f i$;

$S_0 \mathrm{anti}(A f i) = \mathrm{anti} S_0 A f i$;

$A x(S_0 A f y) = S_0 A f y(x)$;

$A x(S_0 f) = S_0 f(x)$.

§7. Семантика словообразования русского языка

В семантическом языке любое понятие изображается некоторой суперпозицией функций (множеством функций). Будем говорить, что понятия X и Y находятся в отношении словообразовательной (непосредственной) мотивации (X - мотивирующее, Y - мотивированное), если суперпозиция X является составной частью суперпозиции Y , и не существует суперпозиции Z , содержащей X и содержащейся в Y .

Например, пусть $X = A f y(x)$ - прилагательное СЯ, тогда $Y = \mathrm{incep}\ \mathrm{copul}(x, A f y(x))$ мотивировано понятием X . Этой мотивации в русском языке соответствует суффикс -е, образующий глаголы от прилагательных: влажный - влажнеть, седой - седеть, красный - краснеть. В СЯ русский суффикс -е можно представить как функцию, отображающую запись x в запись $\mathrm{incep}\ \mathrm{copul}(y,x)$: $x \to \mathrm{incep}\ \mathrm{copul}(y,x)$. Эту функцию назовем словообразовательной семантикой суффикса -е.

Под семантикой словообразования русского языка будем понимать сопоставление каждому словообразовательному аффиксу соответствующей ему функции, определенной на множестве суперпозиций семантического языка.

Кроме основной задачи, которая заключается в описании семантики словообразования, не менее интересна задача: сопоставить определение интуитивного понятия мотивации, которое дается, например, русской грамматикой [10], с формальным определением этого понятия.

Формальная мотивация - строго семантическое понятие. В СЯ понятие X мотивирует понятие Y , если X содержательно беднее, чем Y . Отметим сразу, что сравнение двух определений мотиваций на материале русского языка показывает, что словообразовательный механизм русского языка строго системен.

Следующие примеры демонстрируют полное совпадение обоих определений (при толковании: устарелый=старый и ненужный): старый → стареть → устареть → устарелый → устарелость; старый → incep copul(y, старый) → perf incep copul (y, старый). copul(y, ненужный) → AS_1(perf incep copul (* , старый). copul(*, ненужный)) → S_0(copul(y,AS_1(perf incep copul(* , старый). (* , ненужный)))). Слово 'неравенство' мотивируется словом 'неравный': S_0copul(*, неравный) и словом 'равенство': S_0 copul (*, равный). Еще один пример: глупо= Adv ум-5(x), глупый= A ум-5(x), глупить=oper(x,A ум-5 mult (поступок)) = Adv ум-5(поступать) - совершать глупые поступки, глупо поступать. Таким образом, слово 'глупить' мотивируется как словом 'глупый', так и словом 'глупо'.

Множество примеров полного совпадения интуитивной и формальной мотивации неисчерпаемо, хотя можно привести и примеры расхождений.

Описание словообразования существенно облегчает процесс перевода слов русского языка на семантический язык.

Приведенную ниже словообразовательную семантику нельзя считать полностью адекватной семантике русского языка. При более точном описании следовало бы учитывать семантические признаки мотивирующих слов. Например, суффикс -ун, обладая основной семантической характеристикой $f \to S_1 f$, иногда добавляет к мотивированному им существительному признак, 'склонный к f'. Этим признаком различаются слова: лжец - лгун, певец - певун.

Объем книги не позволяет привести описание всех префиксов и суффиксов русского языка [10]. Но приведенные ниже семантические схемы достаточны для понимания метода описания семантики словообразования.

Словообразование существительных. А. Мотивация глаголами. Суффиксы, образующие существительные от глаголов, служат средством выражения семантики: $f \to S_0 f$, $f \to S_1 f$ (гораздо реже: $f \to S_2 f$). В некоторых случаях реализуется схема $f \to S_i g(f)$, где g есть либо uzor, либо loc, либо result.

1) $f \to S_0 f$ (лов, раздувание, стрижка, убийство, стрельба);

2) $f \to S_1 f$ (мот, житель, клеветник, мойщик, игрок, урожай);

3) $f \to S_2 f$ (рассказ, подарок, похлебка, послание, пашня, баловень, вышивание, ученик(Петрова));

4) $f \to S_2 uzor(f,*)$ (поднос, проявитель, подойник, седло, носилки);

5) $f \to S_2 loc(f,*)$ (загон, вытрезвитель, зимовник, изолятор, бойня);

6) $f \to S_0 result\ f$ (вырез, обрубок, крыша, записка, катыш, варево).

Б. Мотивация прилагательными. Некоторые прилагательные мотивируют существительные, являющиеся названиями растений, животных и т.п. Семантика прилагательного играла определенную роль в момент создания таких существительных, но в настоящее время семантическая связь между ними утеряна. Ясно, что такая мотивация остается за рамками формализации.

1) $f \to S_1 copul(*,f)$ (скаред, озорник, тупица, хитрюга, богач, злюка);

2) $f(x) \to S_1 copul(*,f(x))$ (хроник, сушина, копирка, медовуха);

3) $f(x) \to S_1 caus(*,uzor(y,f(x)))$ (термист, левша, револьвертщик);

4) $f(x) \to S_1 oper(*,f(x))$ (ударник, марафонец, грубиян);

5) $f \to S_1 copul(mult(*),f)$ (опричнина, громадье, воинство, голытьба);

6) $f \to S_2 loc(copul(*,f),*)$ (горельник, быстрина, верховье);

7) $f \to S_0 copul(*,f)$ (удаль, смелость, радушие, доброта, синева);

8) $f \to antimagn\ S_0 copul(*,f)$ (желтина, ехидца);

9) $f \to sing\ S_0 copul(*,f)$ (лукавинка).

В. Мотивация существительными. Семантика словообразования и здесь подчиняется общей схеме: $f \to S_i g(x_1,\dots,f,\dots,x_n)$; но разброс значений гораздо более широкий, чем в предыдущих двух случаях. Ниже приведены лишь некоторые наиболее типичные схемы словообразования.

1) $x \to S_1 oper(*,S_0 *(x))$ (лошадник, голубятник, фокусник, табунщик);

2) $x \to S_1 oper(*,x)$ (завистник, сплетник, горюха, диверсант, бунтарь);

3) $x \to S_2 loc(x,*)$ (телятник, гнойник, глазница, хлебница, псарня);

4) $x \to S_1 caus(*, uzor(*,x))$ (весовщик, плугарь, очкарь, трубач);

5) $x \to S_1 hab(*,x)$ (помощник, горбач, горбун, плантатор, диабетик);

6) $x \to S_1 hab(*, magn(x))$ (ушан, губан, пузан);

7) $x \to S_1 caus(*, incep\ func_0(x))$ (фортификатор, лектор);

8) $x \to S_0 oper(*,x)$ (терроризм, жеребьевка, дождевание);

9) $x \to S_0 copul(*,x)$ (бандитизм, лизина, героизм, метраж, кокетство);

I0) $x \to S_2 oper(x,*)$ (шпионаж);

II) $x \to mult(x)$ (водь, бабье, солдатня, колоннада, братья, березняк);

I2) $x \to sing(x)$ (горошина, дождинка, паутинка, крупица, железяка);

I3) $x \to antimagn(x)$ (шумок, лобик, ленца, сельцо, ручонка, бочонок).

Словообразование глаголов. А. Мотивация именами. Первой (основной) семантической схемой словообразования глаголов является схема: $x \to caus(y, g_x(z,x))$, где функция g_x, выбираемая по слову x есть одна из функций: $incepfunc_0$, $incepcopul$, lab, $uzor$, $incenloc$, $incephab$, $copul$ (x, подобен (y)). Проблема словообразования для этой схемы может быть сформулирована следующим образом: дана функция $g(z,x)$; требуется найти функцию $f(y,z)$, удовлетворяющую уравнению: $result\ f(y,z) = g(z,x)$. Минимальным (в смысле длины записи и количества информации) решением этого уравнения является функция $caus(y, g(z,x))$. Суффиксами, реализующими эту семантическую схему, в основном являются суффиксы: -и, -нича, -ова, -ирова, -изирова.

I) $x \to caus(y, incep\ hab(z,x))$ (ранить, женить, ссудить, титуловать);

2) $x \to caus(y, incep\ copul(z,x))$ (сиротить, калечить, бодрить);

3) $x \to caus(y, incep\ func_0(x))$ (дымить, коптить, мусорить, жертвовать);

4) $x \to caus(y, lab(z,x))$ (пилить, лопатить, асфальтировать, судить, глянцевать, арканить, спорить);

5) $x \to caus(y, incep\ loc(z,x))$ (складировать);

6) $x \to caus(y, uzor(z,x))$ (математизировать, перчить, костылять);

7) $x \to caus(y, copul\ (y,\ подобен(x)))$ (парусить, важничать, молодить);

8) $x \to caus(y, incep\ func_0(S_1 copul(*, подобен(x))))$ (базарить);

Второй семантической схемой словообразования глаголов является схема: $x \to f(y,x)$, где функция $f(y,x)$ - решение уравнения (x - существительное): $S_1 f(y,x) = x$. Простейшим решением этого уравнения является функция $oper(y, S_0 copul(*,x))$. Эта схема реализуется, в основном, суффиксом -нича, реже суффиксами -и, -ствова.

9) $x \to oper(y, S_0 copul(*,x))$ (гореветь, обедать, нукать).

Третьей схемой словообразования глаголов является схема, порождаемая уравнением: $S_1 f(y,x) = x$, где x - прилагательное. Решением этого уравнения является функция $oper(y, S_0 fact(S_0\ copul(*,x)))$.

I0) $x \to oper(y, S_0 fact(S_0 copul(*,x)))$ (хитрить, хромать).

Б. Мотивация глаголами. Суффиксальное образование глаголов, мотивированных глаголами, характеризует общая семантическая схема: $f \to g(f)$, где функция g есть либо $perf\ sing$ (реализуется суффиксом -ну), либо perf (интенсивно (sing f)) (реализуется суффиксом -ану), либо imperf или mult (суффиксы: -ва, -а, -и), либо caus(x,f) (суффиксы: -и, -а).

I) $f \to perf\ sing\ f$ (махнуть, хлебнуть, плюнуть, глянуть);

2) $f \to perf$ (интенсивно (sing f)) (стегануть, сказануть, тряхануть);

3) $perf\ f \to imperf\ f$ (вырубать, вооружать, забывать);

4) $imperf \to mult\ imperf\ f$ (хаживать, знавать, нашивать);

5) $f \to caus(x,f)$ (гасить, кипятить, лепить, морозить);

Аналогичным образом описывается семантика словообразования при помощи префиксов. Приведем лишь несколько типичных схем префиксально-суффиксального словообразования глаголов.

Суффикс -и:

I) $x \to perf\ caus(y, incep\ copul(z,x))$ (выпрямить, снизить);

2) x -> perf caus(y,incep copul(z,magn(x))) (истончить, исполошить);

3) x -> perf caus(y,incep hab(z,x)) (озаглавить, остеклить);

4) x -> perf caus(y,fin hab(x,y)) (обескрылить, расчехлить);

5) x -> perf caus(y,incep copul(z,antimagn(x))) (подновить, подкислить);

6) x -> perf caus(y,fin copul(z,x)) (раскулачить, рассекретить, рассредоточить).

Суффикс -е:

1) x -> perf incep copul(y, покрыт(x)) (замшеть, обомшеть);

2) x -> perf incep copul(y,x) (задеревеневть, ополоуметь);

3) x -> perf incep hab(y,x) (оморщинеть);

4) x -> perf fin hab(y,x) (обезрыбеть, обезденежеть).

Суффикс -ну:

1) x -> antimagn(x) (всхлипнуть, взгрустнуть, прихворнуть).

ЛИТЕРАТУРА

1. АЛАГИЧ С., АРБИБ М. Программирование корректных структурированных программ. М., 1984.
2. АЛЕКСАНДРОВА З.Б. Словарь синонимов русского языка. М., 1969.
3. БАРАНОВ С.Н., НОЗДРУНОВ Н.Р. Язык Форт и его реализация. Л., 1988.
4. БУРБАКИ Н. Теория множеств. М., 1965.
5. ГРИС Д. Наука программирование. М., 1984.
6. ДАЛ У., ДЕЙКСТРА Э., ХООР К. Структурное программирование. М., 1975.
7. ДЕЙКСТРА Э. Дисциплина программирования. М., 1978.
8. КАХРО М.И., КАЛЬЯ А.Р., ТЫГУГ Э.Х. Инструментальная система программирования ЕС ЭВМ (ПРИЗ). М., 1981.
9. МЕЛЬЧУК И.А. Опыт теории лингвистических моделей "Смысл - текст": Семантика, синтаксис. М., 1974.
10. РУССКАЯ грамматика. Т.I,2. М., 1982.
11. ТУЗОВ В.А. Математическая модель языка. Л., 1984.
12. СЛОВАРЬ антонимов. М., 1985.
13. СЛОВАРЬ синонимов. Л., 1975.

СОДЕРЖАНИЕ

Тузов Виталий Алексеевич

ЯЗЫКИ ПРЕДСТАВЛЕНИЯ ЗНАНИЙ. Учебное пособие.
Редактор М.И.Лаптева
Техн.редактор Л.Н.Иванова
Мл.редактор Л.А.Кальвин

Св. темпл. 493.

Подписано в печать с оригинала-макета 03.I2.90. Ф-т 60x90/I6. Бум.тип. № 3. Печать офсетная. Усл.печ.л. 7,5. Усл.кр.-отт.7,5. Уч.-изд.л. 6,9I. Тираж 500 экз. Заказ № 469 . Цена 20 коп.

Редакционно-издательский отдел Ленинградского университета. 199034, Ленинград, Университетская наб., 7/9.
Печатно-множительная лаборатория Ленинградского университета. 199034, Ленинград, наб. Макарова, 6.