ИПМ им.М.В.Келдыша РАН • Электронная библиотека

Препринты ИПМ • Препринт № 41 за 1971 г.

ISSN 2071-2898 (Print)
ISSN 2071-2901 (Online)

В.Ф. Турчин

Программирование на языке РЕФАЛ. 1. Неформальное введение в программирование на языке Рефал

Рекомендуемая форма библиографической ссылки: Турчин В.Ф. Программирование на языке РЕФАЛ. 1. Неформальное введение в программирование на языке Рефал // Препринты ИПМ им. М.В.Келдыша. 1971. № 41. 57 с.
https://library.keldysh.ru/preprint.asp?id=1971-41

ОРДЕНА ЛЕНИНА
ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
АКАДЕМИИ НАУК СССР

В.Ф. Турчин

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ РЕФАЛ

I. Неформальное введение в программирование на языке рефал

Препринт № 41 за 1971 г

Москва

ОРДЕНА ЛЕНИНА ИНСТИТУТ ПРИКЛАДНОЙ МАТЕМАТИКИ
АКАДЕМИИ НАУК СССР

В.Ф.ТУРЧИН

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ РЕФАЛ

I. НЕФОРМАЛЬНОЕ ВВЕДЕНИЕ В ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ РЕФАЛ

Москва, 1971г.

Настоящий выпуск является первым из серии препринтов, посвященных программированию на языке РЕФАЛ. При ссылках из одного препринта данной серии на другой указывается номер выпуска, отделенный чертой / от номера страницы, например: 2/34 34-ая стр. 2-го выпуска. В последний выпуск предполагается включить решения задач ко всей серии.

I. Универсальный метаязык программирования.

Вскоре после появления программно-управляемых вычислительных машин стало ясно, что значительная часть утомительной работы по составлению программы может быть возложена на саму же машину, и появились сначала ассемблеры, а затем проблемно-ориентированные языки (фортран, кобол, алгол и др.) и трансляторы для них. Возрастание числа проблемно-ориентированных языков программирования при наличии различных типов вычислительных машин поставило задачу автоматизации процесса создания трансляторов. Стали разрабатываться системы написания трансляторов, которые потребовали метаязыков для описания алгоритмических языков.

Когда разрабатывается трансляторная система, рассчитанная на определенный класс алгоритмических языков, соображения эффективности диктуют ограничение сферы применимости метаязыка данным классом алгоритмических языков. Метаязык получается более или менее специализированным, как и система, основанная на этом языке. Между тем, создание универсального метаязыка, удобного для описания любых алгоритмических языков, и системы программирования на этом языке, позволяющей легко производить трансляторы, обладало бы преимуществами, меняющими самый стиль использования проблемно-ориентированных алгоритмических языков, и создающими новые возможности автоматизации программирования.

В настоящее время пользователю вычислительной машиной предоставляется не более чем двухуровневая иерархия формализованных языков.

Верхний уровень образует проблемно-ориентированный язык. Этот уровень существует не всегда, ибо не для всех задач (особенно из числа решаемых профессиональными программистами) существуют подходящие проблемно-ориентированные языки, снабженные трансляторами. Нижний уровень образует группа машинно-ориентированных языков, из которых в подавляющем большинстве случаев используется язык ассемблера (автокод). Ассемблер порождает программу на собственно машинном языке, причем, как правило, через посредство еще одного языка - языка модулей (или редактора связей). Таким образом, нижний уровень состоит из трех подуровней, но мы объединяем их в одну группу, ибо переход с одного подуровня на другой - это, фактически, простая перекодировка (перевод "один к одному"). Хотя она производится автоматически (что само по себе очень важно и свидетельствует о пользе расщепления языка на подуровни), объем текста не уменьшается, не меняется и характер языка. Мы как бы имеем дело с более совершенной машиной, понимающей символическую адресацию и другие черты автокода. Более существенно, с системной точки зрения, появление в последнее время (инициатива принадлежит фирме IBM) системы макрокоманд, используемых в рамках языка ассемблера. Пользуясь определенным языком, программист может описать нужную ему макрокоманду, после чего использовать ее в тексте на автокоде, как обычную команду (оператор). Прежде чем приступить к переводу текста в машинный код, ассемблер превратит каждое обращение к макрокоманде в соответствующее макрорасширение - группу обычных команд автокода. Язык описания макрокоманд - это метаязык по отношению к языку ассемблера, он позволяет строить иерархию макрокоманд, описывая одни через другие. Его появление лишний раз подчеркивает необходимость в многоуровневой системе. Однако язык макрокоманд - весьма специализированный и ограниченный язык, предназна-

ченный лишь для превращения макрообращений в макрорасширения. Поэтому хотя он формально и является метаязыком по отношению к автокоду, то-есть, описывает процесс порождения объектов автокода, в практическом использовании он примерно аналогичен аппарату процедур в алголе-60. (Вспомним, что формальная семантика алгола также включает замену обращения к процедуре на её должным образом модифицированное тело). Эти средства способны выразить иерархию понятий в рамках данного языка, но не могут вывести за его пределы. Иерархия макрокоманд или процедур - это нечто гораздо меньшее, чем иерархия языков. Чтобы без стеснений строить языковые иерархии, программист должен иметь универсальный метаязык, с помощью которого он мог бы:

I) описывать сколь угодно сложные преобразования одного текста в другой, без каких бы то ни было ограничений на характер преобразования;

2) вводить в процесс трансляции любую внешнюю информацию, являющуюся, быть может, отражением проблемной ориентации языка или подъязыка;

3) обращать язык сам на себя, описывая преобразование только что введенных или потенциально возможных правил преобразования.

Создание языковой иерархии в программировании имеет два аспекта: формальное выражение иерархии понятий, используемых в программе, и передача машине нетворческой технической работы, выполняемой при переходе с высшего уровня на низший - автоматизация программирования. Все организованные системы устроены иерархически - не обязательно в том смысле, что существует направленный в одну сторону поток управляющей информации, но обязательно в смысле существования иерархии подсистем и их функций. Выразить эту иерархию явным и формальным образом - первая задача языка, важная уже сама по себе. Человек не может сосре-

доточить внимание более чем на нескольких (в пределах первого десятка) объектах одновременно. Отсюда возникают определенные ограничения на число однородных объектов, входящих в составной объект, предлагаемый человеку для изучения, конструирования или модификации - одним словом в качестве объекта интеллектуальной деятельности. Если число однородных объектов укладывается в первый десяток, такая ситуация является идеальной. Это большая удача, если важная формула выражается с помощью нескольких символов. Если однородных объектов несколько десятков, мы мысленно разбиваем их на несколько групп по несколько объектов, и такую ситуацию следует считать нормальной. Когда число объектов выражается сотнями, ситуацию можно назвать плохой, но терпимой. Наконец, когда приходится иметь дело с тысячами однородных объектов (например, машинных команд), ситуация становится нетерпимой. Преодоление технических трудностей, связанных, в конечном счете, с невозможностью охватить умственным взором всю систему в целом, становится главным делом, а исходные идеи отступают на задний план. Математика превращается в программирование.

Если большая система содержит (или должна содержать по замыслу) тождественные подсистемы, то с помощью введения обозначений, иначе говоря, используя понятие подстановки, можно отразить иерархию строения системы и в то же время сократить объем информации, предъявляемой человеку, заставив машину подставлять вместо сокращений их развернутые значения. При этом подставляемое выражение само может быть результатом подстановки (замена формальных параметров на фактические). Здесь мы видим, что с первой функцией языковой иерархии сочетается её вторая функция. Подстановка - это простейший метаязыковый элемент, столь простой, что он может быть введен непосредственно в язык (как

мы уже отмечали выше). Это первый шаг на пути устранения избыточной информации, на пути отделения идей от их реализации и формального выражения иерархии понятий. Однако простой подстановки недостаточно, чтобы значительно продвинуться на этом пути. Например, трансляция с алгола - гораздо более сложный процесс, чем замена сокращений. Поэтому, если на каком-то уровне языка мы не видим в тексте никаких явных повторений, это вовсе не означает,что никакой иерархии понятий в основе этого текста не лежит. Просто эта иерархия требует для своего выражения более сильных средств, чем простая подстановка (разумеется,в широком смысле слова подстановкой можно назвать любое преобразование). Возьмем тот же алгол. Если в текст на этом языке входит пять формул (арифметических выражений), полученных как последовательные производные некоторого исходного выражения, то никакая иерархия процедур не поможет нам сократить объем текста и выявить его структуру. В то же время, введя понятия производной и имея программу аналитического дифференцирования (эта программа и представляет из себя формальное определение понятия производной), мы можем записать программу чрезвычайно компактно и заставить машину выполнить всю черную работу. Формализация ранее не формализованных понятий и автоматизация программирования - два аспекта одного процесса. Автоматизация невозможна без формализаций, формализация приводит к автоматизации, ибо отделяет идеи от их исполнения.

Взглянем с этой точки зрения на любой большой текст на алгоритмическом языке - не важно, будет ли это текст на "высокоуровневом" проблемно-ориентированном языке, или на языке машины. Можно ли допустить, что каждая группа символов, каждая мельчайшая деталь программы - продукт особого, неповторимого хода мысли её создателя? Очевидно,

нельзя. Если объект состоит из тысяч однородных объектов, можно с уверенностью сказать, что при его создании использовались какие-то регулярные методы, какая-то иерархия понятий, только эта иерархия не получила формального выражения в иерархии языков. Конечно, большая программа распадается на подпрограммы и блоки, которые создаются и описываются по отдельности, но это лишь частичная формализация, не доведенная до того уровня, когда становится возможной существенная автоматизация программирования (сравните программу на алголе с программой на машинном языке, дополненной полуформальным описанием подпрограмм и блоков).

Почему же формализация не доводится до конца? Из-за трудностей, связанных с трансляцией, с переходом от языка к языку. Чтобы получить выигрыш (в смысле затраты труда программиста) от создания нового языка, надо заставить машину совершать достаточно сложную работу при переводе с этого языка, а для этого надо написать достаточно сложный транслятор. Если много пользователей решают много однотипных задач, для них создается проблемно-ориентированный язык и транслятор к нему. В этом случае затраты на создание транслятора явно окупаются. Но есть много нестандартных задач, которые обладают каждая своей иерархией понятий и требуют каждая своего транслятора или даже серии трансляторов (соответственно числу уровней иерархии). Чтобы сделать приемлемым на практике метод программирования, включающий создание иерархии специализированных языков и трансляторов для каждой крупной задачи, необходим универсальный метаязык, причем метаязык, удовлетворяющий двум требованиям. Во-первых, он должен быть удобным для человека. Текст на метаязыке должен представляться не в виде сложной и запутанной программы, которая каким-то неведомым образом осуществляет трансляцию, а скорее в виде семантического описания

языка, с которого производится трансляция, в виде набора предложений, выражающих понятия этого языка через более простые понятия. Во-вторых, метаязык должен допускать эффективную реализацию на вычислительной машине, иначе его систематическое использование повлечет большие потери машинного времени.

Такой метаязык сделал бы столь же естественным использование узко специализированных языков программирования и построение иерархии языков, сколь естественным является, например, в алголе использование процедур, вводимых программистом *ad hoc* (специально для данного случая) и построение иерархии процедур. Однако в отличие от аппарата процедур, который, как мы видели, не позволяет преодолеть некоторых ограничений, система программирования, основанная на универсальном языке, свободна от ограничений на вводимые понятия. Действительно, так как наш метаязык направлен на алгоритмические языки, причем вследствие универсальности должен быть пригоден для описания любых языков, и в то же время сам он является также алгоритмическим языком, он в качестве объектов работы может иметь свои собственные тексты. Поэтому рамки системы раздвигаются до бесконечности: она способна вместить любые фигуры самоописания и самопреобразования.

Основывая систему программирования на универсальном языке программирования, мы вынуждены либо сильно ограничивать возможности программиста, либо создавать сложнейший язык, трудный не только для реализации, но и для изучения, и для использования. Фиксируя же лишь метаязык, мы получаем гораздо более гибкую систему, способную к бесконечному развитию. В рамках такой системы можно надеяться разработать такие методы программирования, когда человек будет создавать только идейный стержень, только зародыш программы, а развитие этого зародыша в зрелый, нормально функционирующий организм, будет происходить авто-

матически, хотя и не без контроля со стороны своего творца. Разумеется, создание метаязыка не избавляет нас от разработки содержательных понятий и языков программирования, но оно дает для этого необходимый инструмент.

2. Преобразование символьной информации.

Мы говорили о метаязыке программирования как об алгоритмическом языке, ориентированном на описание трансляции, то-есть, переработки одного текста на алгоритмическом языке в другой. Является ли такой подход достаточно общим? Не существуют ли какие-то другие типы метаязыков, пригодных для формального описания семантики алгоритмических языков?

Покажем, что метаязык для формального описания семантики алгоритмических языков сам является алгоритмическим языком.

Пусть $T_L(A)$ - текст на алгоритмическом языке L являющийся описанием некоего алгоритма A. И пусть $T_A(D)$ - текст, представляющий собой начальные данные D для алгоритма A, записанные так, как этого требует язык L Совокупность этих двух текстов, которую мы будем изображать в виде $T_L(A) + T_A(D)$, представляет собой входную информацию для некоторого кибернетического устройства (машины) C_L, "понимающей" язык L - такой смысл имеет утверждение, что L - формализованный алгоритмический язык. Получив эту информацию, машина C_L приступает к работе и производит некоторую выходную информацию - текст F который есть результат применения алгоритма A к начальным данным D Символически мы изобразим это так:

$$C_L\left(T_L(A) + T_A(D)\right) = F$$

Обозначим теперь через $T_M(L)$ описание языка L на формализованном метаязыке M и будем считать, что это описание - полное, то-есть, оно включает как синтаксис, так и семантику. Но тот факт, что оно включает семантику, означает, что если описание языка $T_M(L)$ сопроводить описанием алгоритма $T_L(A)$, то это будет достаточной информацией для выполнения алгоритма A с любыми начальными данными. А значит должна существовать такая машина C_M, которая, получив в качестве входной информации текст $T_M(L) + T_L(A) + T_A(D)$, произведет на выходе информацию F

$$C_M\left(T_M(L) + T_L(A) + T_A(D)\right) = F$$

Машина C_M определяет семантику языка M подобно тому, как машина C_L определяет семантику языка L. Так как на входе и выходе машина C_M имеет текстовые объекты, язык M - алгоритмический. Отличие языка M от языка L состоит только в том, что машина C_M требует не двучленной, а трехчленной входной информации: описание языка, описание алгоритма и описание начальных данных. Первый член - текст на самом языке M - это, по существу, программа для машины C_M, а именно, программа, указывающая, как перерабатывать текст $T_L(A) + T_A(D)$. Следовательно, алгоритмический язык M ориентирован на преобразование текстовой информации. Особенностью машины C_M является,то, что вместе с текстом $T_L(A)$ перерабатываются сразу начальные данные $T_A(D)$ так что в качестве результата выдается результат работы алгоритма A Такой режим использования текста $T_L(A)$ называется режимом интерпретации. Когда текст $T_L(A)$ транслируется без начальных данных в текст $T_{L'}(A)$ на другом языке L', говорят о режиме компиляции.

Итак, любой универсальный способ формального описания семантики алгоритмических языков - то ли путем создания интерпретирующей машины C_M, то ли путем описания процесса трансляции - требует создания алгоритмического языка, удобного для манипуляций над произвольными (следствие универсальности) текстовыми объектами, то-есть, над последовательностями символов из некоторого конечного алфавита. Разумеется, любой алгоритмический язык рассчитан на действия над последовательностями символов, но в обычных проблемно-ориентированных языках всегда существует определенная специфика объектов (числа, бухгалтерские записи и т.п.) и действий над ними (например, арифметические операции), которая и определяет специфику языка. В нашем же случае спецификой является именно отсутствие всякой специфики. Эта ориентация языка получила название "преобразование символьной (или текстовой) информации", или просто "символьные преобразования" - "*symbol manipulation*" в литературе на английском языке. Впрочем, утверждение об отсутствии всякой специфики не совсем точно. Определенная специфика в объекте и характере преобразований все-таки есть. Она состоит в том, что эти объекты - не случайные нагромождения символов, а слова и фразы какого-то языка. Никаких ограничений на языки, с которыми мы собираемся работать, мы формально не накладываем, тем не менее языки остаются языками, и на деле это накладывает отпечаток на объекты, с которыми будет иметь дело язык M, а также на типичные действия над ними. Мы видим, здесь, что хотя искомый нами язык является обычным алгоритмическим языком, и можно назвать его проблемным языком, ориентированным на симовольные преобразования, он сохраняет в то же время свою метаязыковую природу, ибо он рассчитан на действия с языковыми объектами.

Кроме приложений к проблеме трансляции, алгоритмические языки, ориентированные на символьные преобразования, имеют и другие, не менее важные, применения. В первую очередь это машинное выполнение громоздких аналитических выкладок в теоретической физике и прикладной математике. Хотя в настоящее время значение этих работ по сравнению со значением машинного производства арифметических действий невелико, можно предсказать, что оно будет неуклонно возрастать и приведет в конце концов к крупнейшим сдвигам в математизированных науках. Другие сферы использования алгоритмических языков, ориентированных на символьные преобразования - это проектирование "умных" информационных систем, осуществляющих нетривиальную логическую обработку информации, перевод с естественных языков, машинное доказательство теорем, моделирование целенаправленного поведения и т.п. Надо отметить, что все эти области принципиально ничем не отличаются от области автоматизации программирования, и часто между ними невозможно даже провести черту. Во всех случаях мы учим машину совершать сложные преобразования над объектами, определенными в весьма развитых языках (алгоритмические языки, язык алгебры, язык исчисления предикатов).

Язык рефал (алгоритмический язык рекурсивных функций) задуман и функционирует как универсальный метаязык для описания преобразований языковых объектов. Однако поскольку заранее совершенно не очевидно, что рекурсивные функции имеют какое-либо отношение к задаче создания универсального метаязыка, мы будем на первых порах называть наш метаязык языком М. Наша цель - показать, как из природы поставленной задачи вытекают основные черты создаваемого языка. В частности, у нас естественным образом появится понятие рекурсивной функции, что лишний раз подтверждает его глубину и важ-

ность. Описав неформально важнейшие понятия рефала в процессе их становления, мы перейдем (в Главе 2) к формальному описанию языка.

3. Понятие конкретизации

Поставим вопрос: каким должен быть язык, ориентированный на указанные выше проблемы, чтобы он был удобен для пользователя?

Машинно-независимые алгоритмические языки, широко используемые в современном программировании, удобны для записи задач из определенной области вследствие того, что они строятся на основе формализации ряда понятий, важных и характерных для данной специальной области. Нам же нужен язык, ориентированный не на какие-либо конкретные понятия, а предназначенный для описания любых языков и понятий (метаязык), поэтому такой язык будет удобен для человека лишь в том случае, если он схватит какие-то чрезвычайно общие и в то же время важные черты человеческого мышления или точнее, человеческой языковой деятельности, в которой выражается мышление.

В поисках таких черт обратимся к естественным языкам и их продолжению - формализованным языкам математики. Важнейшей чертой этих языков является наличие в них иерархии понятий. Естественный язык можно представить себе в виде многоэтажной пирамиды, возвышающейся над фундаментом чувственного опыта. Элементами этой пирамиды при семантическом подходе надо считать морфемы - минимальные смысловые единицы языка. Складываясь в цепочки, морфемы образуют языковые объекты (лингемы): слова, группы слов, предложения. Языковые объекты, расположенные на самых низких этажах пирамиды, фиксируют наиболее простые, близкие к чувственному опыту понятия: "больно", "холодно", "заяц", "падать" и т.п., на основе которых строятся

более сложные и абстрактные понятия, а на их основе - еще более сложные, и т.д.; все эти понятия фиксируются языковыми объектами. Где-то на средних этажах пирамиды расположены понятия "север", "число", "работа", "чин полковника", а где-то на самом верху - "отчуждение", "гомозиготный", "бикомпактность". В естественном языке не существует точной меры сложности или абстрактности понятия, поэтому невозможно и распределить языковые объекты по этажам строго однозначно. Однако принцип образования сложных и абстрактных понятий из более простых и конкретных путем конструирования и абстрагирования, несомненно, лежит в основе построения любого языка.

Возьмем теперь какой-нибудь языковый объект, например, слово, и зададим вопрос: что значит понимать это слово? Очевидно, что физический носитель, физический вид слова не имеет сам по себе никакого значения, имеют значение лишь связи этого слова с другими словами и с "комплексами ощущений" - элементами чувственного опыта. Следовательно, понимать слово - значит уметь пройти в обратном направлении путь его построения. Понимать абстрактное понятие - значит уметь его конкретизировать в каждой заданной ситуации, понимать сложное понятие - значит уметь свести его к ряду более простых. И то, и другое означает замену языкового объекта, занимающего более высокое положение в языковой пирамиде, на ряд объектов, занимающих в ней более низкое положение. Эту операцию мы будем называть конкретизацией языкового объекта. При некоторых видах языковой деятельности мы не доводим конкретизацию языковых объектов до самого низа, до непосредственного чувственного опыта, однако, предполагается, что мы знаем, как это сделать, иначе слово не имеет для нас никакого смысла. Итак, семантика языкового объекта определяется

правилом его конкретизации, а семантика языка в целом - совокупностью правил конкретизации, которая позволяет путем ряда шагов свести каждый языковый объект к некоторым несводимым элементарным объектам ("комплексам ощущений"). Операцию конкретизации можно также определить как переход от имени к значению.

Нарисованная схема определения семантики объектов естественного языка является, без сомнения, упрощенной, однако без упрощения невозможна никакая формализация, а ведь наша задача и заключается в формализации семантических описаний. Поэтому при построении языка M мы эту схему примем за основу. Мы введем два знака: $\underline{\kappa}$ и $\underline{\cdot}$ которые будем называть конкретизационными или функциональными скобками[x], и в которые будем заключать языковый объект, подлежащий конкретизации. Так что, например, если x есть некоторая переменная величина, то $\underline{\kappa}\, x\, \underline{\cdot}$ (конкретизация x) будет изображать значение этой величины. Другой пример: объект $\underline{\kappa}\, 28+7\, \underline{\cdot}$ будет рано или поздно заменен (если только правильно определена операция сложения) на объект 35. Выполнение конкретизации - переход от имени к значению - мы объявим основной (и по существу, единственной) операцией в языке M Эту операцию будет выполнять машина C_M "понимающая" язык M. (Когда вместо "язык M" мы будем говорить "рефал", машину C_M мы будем называть "рефал-машиной"). Информацию для выполнения всевозможных конкретизаций мы будем записывать в виде предложений языка M (правил конкретизации).

---

x/ Знак $\underline{\kappa}$ мы будем также называть "знаком конкретизации", а знак $\underline{\cdot}$ - "конкретизационной точкой".

Очевидно, что явное введение конкретизационных скобок непосредственно вытекает из задач, поставленных перед языком M. В специализированных формализованных языках, опирающихся на фиксированную иерерхию понятий, обычно договариваются о такой системе обозначений, которая отражала бы положение языкового объекта в этой иерархии. Разделение производится по типу знаков, по шрифту, по положению букв в алфавите и т.п. Так, значением x может быть 2.7183, или x_0 или a но никак не наоборот. Это дает возможность разобраться, где имя, а где значение. В нашем же случае, когда мы проектируем метаязык, рассчитанный на произвольные системы понятий и обозначений, переход от имени к значению должен быть указан с помощью специальных знаков.

4. Знаки, символы, выражения

Теперь необходимо уточнить структуру объектов языка M. Подобно естественным языкам язык M является одномерным знаковым языком, то-есть, его объекты суть последовательности некоторых знаков. Под знаком мы понимаем минимальную синтаксическую единицу языка, не расчленимую на составные части, аналогично буквам или фонемам естественного языка (в письменной и устной формах, соответственно). Так как метаязык M должен быть применим к любым языкам, не следует ограничивать набор знаков языка M. В то же время нам необходимо, очевидно, какое-то число специфических знаков с фиксированным значением. Поэтому мы объявим, что все знаки языка делятся на собственные и объектные знаки. Набор объектных знаков не фиксируется. Мы будем только считать, что он ограничен. В настоящей книге мы будем использовать в качестве объектных знаков буквы

русского и латинского языков, цифры и общепринятые математические знаки, как +, −, = и т.п. Что касается собственных знаков, то в данной (неформальной) части описания языка мы будем вводить их по мере надобности. К настоящему моменту нам известно два собственных знака: $\underline{\kappa}$ и $\underline{.}$

Буквы или фонемы в естественном языке не имеют, как таковые, смысла, значения, они служат лишь материалом для построения минимальных семантических единиц - морфем: корней слов, суффиксов, предлогов и т.д. При этом морфема может состоять из одной буквы, например "в" (предлог) или из нескольких букв, например, "дуб" (корень), "из" (предлог), "ая" (окончание). В языке M минимальную семантическую единицу мы назовем <u>символом</u> и положим, что символ может изображаться как одним знаком, так и группой знаков, ограниченной тем или иным способом слева и справа. Будем в качестве ограничителя пользоваться знаком ' (кавычка), который будем рассматривать как собственный знак языка M. Итак, мы определяем символ как объектный знак или как последовательность объектных знаков, взятую в кавычки (<u>составной</u> символ). Ограничение знаков, образующих составной символ, объектными знаками, представляется вполне естественными. Из него, впрочем, есть два исключения (символы обмена), о которых - ниже.

Примеры символов:

$\underline{\kappa}$

=

'если'

'конец. переход к след-процедуре'

Примеры последовательностей знаков, которые не являются символами:

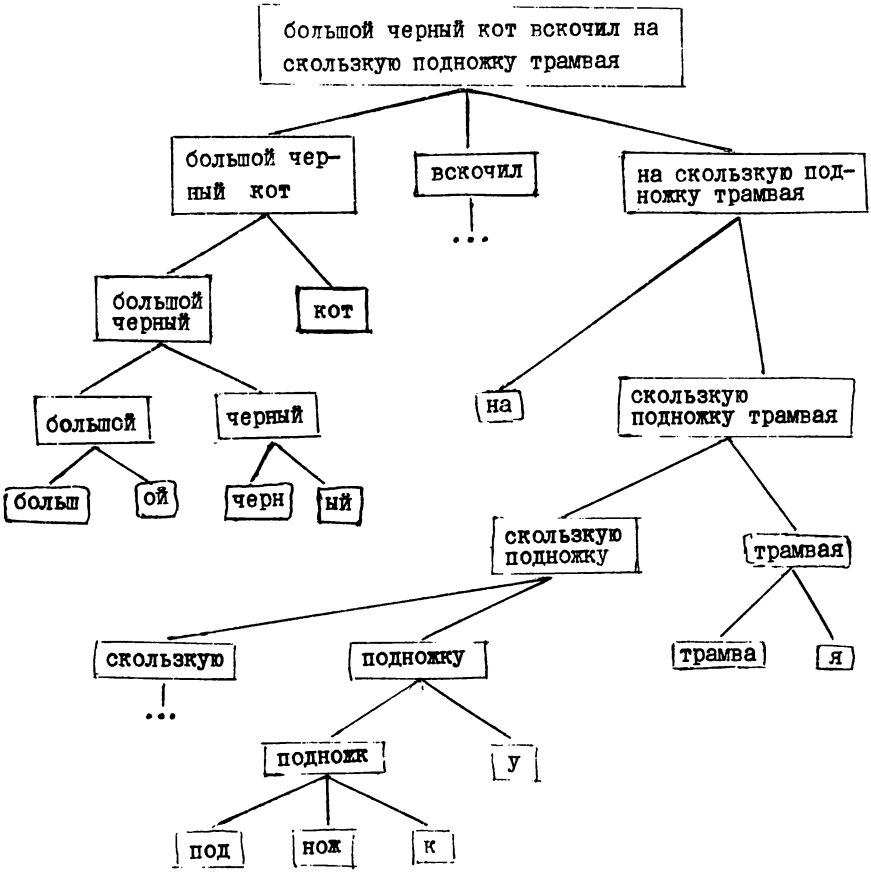
X Y z

'⊑ ≍ .'

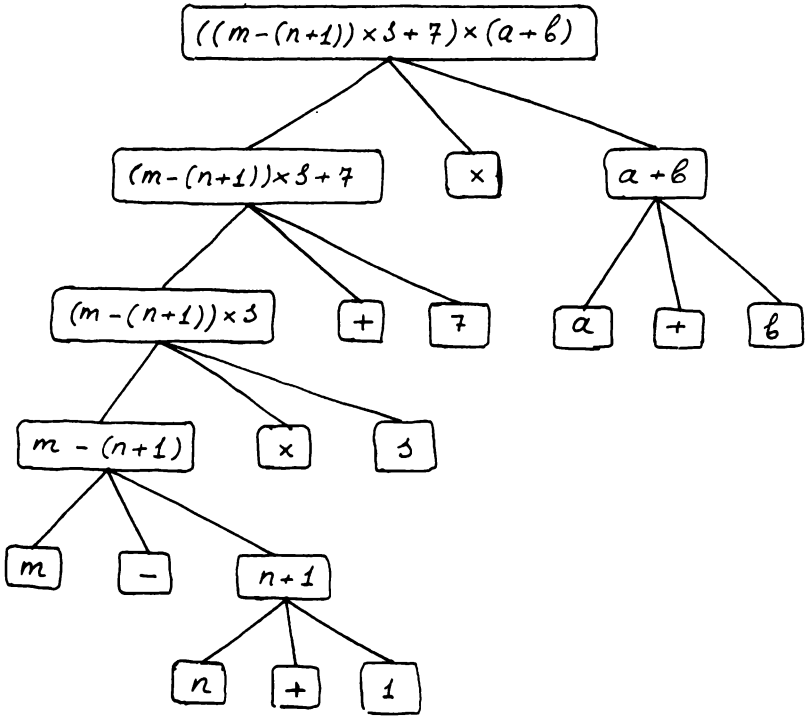
если'

"'в

В естественных языках морфемы соединяются в слова, словосочетания и предложения, имеющие определенную синтаксическую структуру. Путем синтаксического разбора можно каждый языковый объект разложить на составляющие его объекты и восстановить путь построения этого объекта из элементарных частей. То же относится и к формализованным языкам. Синтаксические средства, используемые для конструирования сложных объектов из простых (а следовательно, и для разложения сложных объектов на простые) могут быть самыми разнообразными; создавая язык M, мы не можем предполагать в его объектах какой-то определенный, специализированный синтаксис. Тем не менее, есть одна черта, общая для всех синтаксических схем, которую мы не можем не учесть при построении метаязыка: это тот факт, что синтаксический анализ (и синтез) всегда порождает определенное дерево языковых объектов (см. фиг.I.I и фиг.I.2). Простейший и наиболее привычный способ изображения такой структуры в виде линейной последовательности знаков - это использование скобок. Этот способ широко применяется во многих формализованных языках, начиная со школьной алгебры. Мы привыкли сразу выделять скобки в ряду других знаков; мы относимся к ним не как к полноправным символам, а как к специальным знакам, служащим для придания объекту иерархической структуры. Это отражено на фиг. I, 2, где скобки не указаны как элементы строк, наряду с подвыражениями и знаками операций, а проявляются только в построении дерева.



Фиг. I.I. Дерево синтаксического анализа фразы русского языка



Фиг. I.2. Дерево синтаксического анализа алгебраического выражения

Итак, для изображения синтаксических конструкций мы введем в языке M круглые скобки, которые будем рассматривать как собственные знаки языка. Круглые скобки, придающие структуру объектам работы, мы будем называть объектными, в отличие от функциональных скобок, указывающих на необходимость выполнения конкретизации. Заметим, что кавычка, которой мы пользуемся для ограничения составных символов, тоже может быть названа скобкой, а именно, символьной скобкой. Причина, по которой символьная скобка одна, а скобки двух других типов - парные, состоит в том, что при построении сложных символов мы ограничиваемся одноуровневыми структурами, поэтому можем обойтись одним ограничителем.

Теперь мы введем понятие выражения, которое можно определить как последовательность знаков, правильно построенную относительно всех трех типов скобок: символьных, объектных и функциональных.

Расшифровка этого определения такова:

1. Пустой объект есть выражение.
2. Символ есть выражение.
3. Последовательность выражений есть выражение.
4. Выражение, взятое в объектные или функциональные скобки, есть выражение.
5. Объект, который на основании предыдущего не может быть квалифицирован как выражение, не есть выражение.

Примеры выражений:

$ABC()$

```
(((’больш’ ’ой’)(’черн’ ’ый’))’кот’)(в’скоч’ ’ил’)
          (’на’(((’скольз’ к ’ую’)((’под’ ’нож’ к)у))
          (’трамва’ я)))
```

К 'ЕСЛИ' А < В 'ТО' А 'ИНАЧЕ' В

К К х (К у .)

Примеры последовательностей символов, которые не являются выражениями:

К 'СЛОЖ' А, В

)К х . (

К 'СЛОЖ' А, (В .)

Полезно также определить терм как некоторый частный случай выражения. Терм это либо символ либо выражение, взятое в объектные или функциональные скобки. Следовательно, выражение всегда есть последовательность из некоторого числа (быть может, нуля) термов.

Поскольку скобки мы рассматриваем не как объектные знаки, а как собственные знаки, как способ изображения структуры языковых объектов, мы договоримся, что язык M будет иметь дело только с выражениями. Последовательности символов, не являющиеся выражениями, то-есть, неправильно построенные относительно объектных или функциональных скобок, мы объявляем неправомочными в качестве языковых объектов, и допускаем их лишь в качестве составных частей выражений.

5. Предложения

Алгоритмические языки можно разделить на две группы. Первую группу образуют языки, которые мы назовем языками операторного типа. Элементарными единицами программы являются здесь операторы, то-есть, приказы, выполнение которых сводится к четко определенному изменению четко определенной части памяти машины. Типичным представителем этой группы является язык машины Тьюринга. Сюда же относятся языки конкретных вычислительных машин, а также такие широко распространенные языки программирования, как ФОРТРАН и АЛГОЛ. Языки второй группы мы

назовем языками сентенциального типа. Программа, написанная на таком языке, представляется в виде набора предложений (соотношений, правил, формул), которые машина, понимающая данный язык умеет каким-то образом применять к обрабатываемой информации. Язык нормальных алгоритмов А.А.Маркова является примером сентенциального языка, созданного с теоретическими целями. Язык ЛИСП может служить примером сентенциального языка для практических целей.

Оба типа алгоритмических языков имеют прообразы в естественных языках: операторные языки - в виде повелительного наклонения (приказания), сентенциальные - в виде изъявительного наклонения (описания). Сравнивая операторные и сентенциальные языки, мы не можем не обратить внимания на соотношение между использованием изъявительного и повелительного наклонения в естественных языках. Мы видим, что изъявительное наклонение является несравненно более распространенным и образует, в сущности, основу языка, в то время как повелительное наклонение предстает в виде некоторой специальной модификации. Нетрудно понять и причины этого. Если язык достаточно развит, то-есть, содержит много сложных понятий, то основная масса текстов на этом языке выполняет задачу выражения связей между понятиями, то-есть, является описаниями. При наличии этой основной массы, приказания, даже очень сложные ("подправьте-ка, пожалуйста, в этой статье литературный стиль") занимают совсем мало места, если даже допустить, что язык в целом и создавался-то для выражения подобных приказаний.

Итак, относительный вес изъявительного наклонения является мерой развитости языка. Современные вычислительные машины понимают исключительно операторные языки и находятся, таким образом, на уровне животных, язык которых содержит исключительно повелительное наклонение. Создавая проблемно-ориентированные алгоритмические языки,

люди в той или иной степени вводят в них изъявительное наклонение, приближая машину к человеку. Если мы рассмотрим, например, АЛГОЛ, то хотя этот язык и является операторным по своей структуре, наличие в нем арифметических и логических выражений является элементом изъявительного наклонения, а наличие описаний - это уже явное использование изъявительного наклонения.

Язык M будет у нас сентенциальным в своей основе и по своей природе, ибо он и задуман как язык для описания связей, соотношений между понятиями. Вся информация, которую мы желаем передать на языке M, должна быть выражена в виде правил конкретизации. Если продолжить аналогию с естественным языком, то правила конкретизации соответствуют предложениям естественного языка, поэтому мы будем также называть их предложениями (см. табл.1.1). Отделять предложения друг от друга мы будем собственным языком § (параграф), помещая его перед каждым новым предложением.

Таблица 1.1 Соответствие между объектами языка M и естественного языка

Уровень	Язык M (рефал)	Естественный язык
I	знак	фонема, буква
2	символ	морфема
3	выражение	слово, словосочетание
4	предложение	предложение

Поскольку правило конкретизации есть указание для замены одного языкового объекта на другой, предложение должно состоять из левой части (заменяемый объект) и правой части (объект, заменяющий левую часть). Для разделения левой и правой части нам понадобится еще один собственный знак. Мы возложим эту миссию на знак $\geq$ (подчеркнутая импликация). Итак, предложение, выражающее тот факт, что значение переменной величины x есть I37, мы запишем в виде:

$$\S\ \underline{K}\ x \geq \text{I37}$$

то-есть, выражение $\underline{K}\ x\ .$ (конкретизация x) должно быть заменено на I37. Конкретизационная точка $.$ в конце левой части подразумевается. Мы могли бы написать:

$$\S\ \underline{K}\ x\ . \geq \text{I37}$$

однако для краткости мы точку опускаем, вернее, сливаем её со знаком $\geq$, так что с точки зрения скобочной структуры последний играет роль правой функциональной скобки. Между знаком § и первым знаком $\underline{K}$ мы разрешим вставлять последовательность знаков, не содержащую знаков $\underline{K}$, которая будет служить номером предложения, или комментарием к нему, например:

$$\S\ 1.1\ \underline{K}\ x \geq \text{I37}$$

Теперь пришло время подумать о машине C_M , которая, используя предложения, будет выполнять конкретизации. Очевидно, что должно существовать какое-то выражение, которое является объектом работы машины C_M . Мы будем говорить, что это выражение находится в поле зрения машины C_M . Кроме того, машина C_M должна иметь поле памяти, для хранения предложений. Работа машины C_M будет складываться из последовательных шагов, каждый из которых представляет выполнение одного акта конкретизации.

Пусть в поле памяти машины C_M находится § I.I[x], а в поле зрения - выражение

$$\underline{K}\ x\ .$$

Тогда за один шаг машина C_M заменит содержимое поля зрения на выражение

I37

после чего она остановится, ибо знаков конкретизации в поле зрения больше нет, и следовательно, делать ей нечего.

Так как поле памяти содержит, вообще говоря, набор (последовательность) предложений, может оказаться, что для выполнения данной конкретизации пригодно не одно, а несколько предложений. Например, в поле памяти кроме § I.I может стоять еще предложение

$$\S\ 1.2\ \underline{K}\ x = 274$$

Неоднозначность, которая отсюда может возникнуть, устраняется следующим образом. Договоримся, что машина C_M просматривает предложения в том порядке, в котором они стоят в поле памяти, и применяет первое из них, которое окажется подходящим, после чего шаг считается выполненным.

Поле зрения машины C_M также может содержать сколько угодно конкретизационных скобок, причем они могут быть как угодно вложены друг в друга. Следовательно необходимо договориться, каким образом будет машина C_M выбирать выражение, с которого надо начинать процесс конкретизации.

Назовем ведущим знаком конкретизаций тот знак $\underline{K}$, который вместе с парным ему знаком $\underline{\ .\ }$ ограничивает выражение, подлежащее

---

x/ Нумерация предложений рефала - единая по всей книге. Если ссылок на предложение заведомо не предполагается, весь номер, или его первая часть, может опускаться. О значении точки в нумерации см. ниже.

конкретизации в первую очередь. Так как прежде чем вычислять значение функции, необходимо вычислить значения всех аргументов, введем следующее соглашение. Ведущим знаком $\underline{K}$ является первый из знаков $\underline{K}$, в области действия которого (то-есть, в последовательности знаков до парной скобки $\doteq$), нет ни одного знака $\underline{K}$ Теперь мы можем так описать работу машины C_M. В начале каждого шага машина C_M просматривает поле зрения слева направо в поисках ведущего знака конкретизации. Найдя его, она сравнивает терм, начинающийся с этого знака с левыми частями предложений, стоящих в поле памяти. Найдя подходящее выражение, она делает в поле зрения необходимую замену, после чего выполнение данного шага заканчивается и машина приступает к выполнению следующего шага.

Пусть, например, поле памяти содержит набор предложений:

$\S 1.1 \quad \underline{K} x \supseteq 137$

$\S 1.2 \quad \underline{K} x \supseteq 274$

$\S 2.1 \quad \underline{K} y \supseteq 2$

$\S 3.1 \quad \underline{K}\, 137 + 2 \supseteq 139$

а поле зрения содержит выражение:

$$\underline{K}\, \underline{K} x \doteq + \underline{K} y \doteq \doteq$$

На первом шаге ведущим знаком $\underline{K}$ будет знак $\underline{K}$, стоящий перед символом x. В результате замены поле зрение принимает вид:

$$\underline{K}\, 137 + \underline{K}\, y \doteq \doteq$$

Теперь первоочередной конкретизации подлежит выражение $\underline{\subset} y \doteq$. Первые два предложения в поле памяти оказываются неприменимыми, и применяется § 2.I. В результате выполнения второго шага поле зрения принимает вид

$$\underline{\subset} 137 + 2 \doteq$$

На третьем шаге применяется § 3.I, и в поле зрения оказывается выражение

$$139$$

которое уже не содержит знаков $\underline{\subset}$.

6. Свободные переменные

В примерах, которые мы до сих пор рассматривали, предложения, которые оказывались применимыми для выполнения конкретизации, имели левые части, в точности совпадающие с конкретизируемым выражением. Однако совершенно ясно, что далеко мы так не уйдем. Чтобы заставить машину C_M выполнить сложение I37 + 2 = I39, мы ввели в поле памяти § 3.I. Очевидно, нельзя в поле памяти запастись предложением на каждую пару складываемых чисел. Чтобы записать предложение, применимое более чем к одному конкретизируемому выражению, необходимо ввести в него свободные переменные, которые при различных применениях предложения могут принимать различные значения.

Языковые объекты, с которыми имеет дело машина C_M, это всегда выражения, включая сюда, в качестве частных случаев термы и символы. Поэтому мы введем три типа свободных переменных, соответственно синтаксическому типу тех объектов, которые могут быть их значениями. Для изображения свободных переменных введем три собственных знака - указателя свободных переменных: $\underline{E}$ - указатель выражения, $\underline{W}$ - указатель терма и $\underline{S}$ - указатель символа. Свободную переменную будем изображать парой символов, состоящей из указателя и следующего за ним произвольного объектного знака - идентификатора свободной переменной. Так, $\underline{E}1$, $\underline{E}2$, $\underline{E}a$ - свободные переменные выражения; в обычной математической системе обозначений мы записали бы их в виде E_1, E_2, E_a. Свободная переменная выражения может принять в качестве значения любое выражение. Значением свободной переменной терма (скажем, $\underline{W}1$) может быть любой терм, например, A или $(A\ B)$, но никак

не $A\,B$. Значением свободной переменной вида $\underline{S}\,a$ может быть только символ.

Разрешая использовать в левых и правых частях предложений свободные переменные, мы получим мощное изобразительное средство и завершаем конструирование основного ядра языка M . Теперь машина C_M , чтобы решить, применимо ли данное предложение к данному конкретизируемому выражению, должна определить, можно ли придать свободным переменным в левой части предложения такие значения, чтобы левая часть совпала с конкретизируемым выражением. Это действие машины C_M мы будем называть синтаксическим отождествлением. В случае успешного синтаксического отождествления свободные переменные, входящие в левую часть, принимают определенные значения (соответствующие, конечно, их синтаксическому типу), и при подстановке правой части предложения в поле зрения на место свободных переменных вписываются их значения. Если синтаксическое отождествление невозможно, предложение неприменимо.

Рассмотрим несколько примеров. Допустим, мы хотим описать понятие "первый символ выражения". Это значит, что мы хотим, чтобы, например, выполнение конкретизации:

$\underline{K}$ первый символ выражения $Z\,(A\,B)+F$.

дало бы символ Z , выполнение конкретизации

$\underline{K}$ первый символ выражения + .

дало бы символ + и т.д.

Задачу решает следующее предложение:

$\S\,4A.1$ $\underline{K}$ первый символ выражения $\underline{S1}\;\underline{E2} \geq \underline{S1}$

При отождествлении конкретизируемого выражения в первом из двух приведенных примеров свободная переменная $\underline{S1}$ принимает значение Z , а свободная переменная $\underline{E2}$ – значение $(AB)+F$.

В результате замены в поле зрения оказывается символ Z . Во втором примере $S1$ принимает значение +, а $E2$ - значение пустого выражения.

Для создания языкового объекта, имеющего значение первого символа некоторого выражения мы использовали русские слова "первый символ выражения" в их натуральном виде. С точки зрения языка M эти слова образуют выражение, состоящее из 23 символов (считая пробелы). Ясно, что это не слишком экономно, а главное, не нужно, ибо эти слова выступают в $\S 4A.1$ как единое целое. Поэтому разумно заменить эти 23 символа на один символ. Мы можем изобразить этот символ одним знаком, например, греческой буквой α , а описание этого понятия русскими словами вынести в комментарий. Тогда получим:

$\S 4B.1$ - первый символ выражения

$$K \alpha S1 E2 \supseteq S1$$

Или мы можем образовать составной символ из букв, которые входят в соответствующие русские слова, причем сократить их, чтобы не писать их слишком много букв:

$$\S 4C.1\ K\ 'ПЕРВСИМ'\ S1\ E2 \supseteq S1$$

Наконец, можно объединить эти два способа, сохранив в качестве мнемоники составной символ и поместив в комментарий его расшифровку.

Допустим, что при наличии в поле памяти одного лишь $\S 4C.1$ мы поместили в поле зрения выражение

$$K\ 'ПЕРВСИМ'.$$

или выражение

$$K\ 'ПЕРВСИМ'\ (ABC) + B33.$$

то-есть, мы хотим, чтобы машина C_M ответила на вопрос, каков первый символ пустого выражения или первый символ выражения, начинающегося со скобки. И в том, и в другом случае $\S 4C.1$ окажется неприменимым,

и машина C_M придет в состояние аварийной остановки. Она не понимает задачи; понятие 'ПЕРВСИМ' оказалось не вполне определенным. Мы можем дополнить его тем или иным способом. Простейший способ - это объявить, что в обоих сомнительных случаях первый символ есть пустое выражение. Для этого мы должны §4С.1 дополнить еще одним предложением:

§4С.2 К 'ПЕРВСИМ' Е1 ≡

которое поместить в поле памяти после §4С.1 . Здесь мы сталкиваемся с очень важным следствием определенного нами алгоритма работы машины C_M результат конкретизации существенно зависит от порядка расположения предложений в поле памяти. Если поле памяти имеет вид:

§4С.2 К 'ПЕРВСИМ' Е1 ≡

§4С.1 К 'ПЕРВСИМ' S1 E2 ≡ S1

то второе предложение (§4С.1) не будет работать никогда, ибо всегда сработает первое предложение и "первый символ выражения", определенный таким образом, всегда будет принимать пустое значение. Если же предложения стоят в таком порядке:

§4С.1 К 'ПЕРВСИМ' S1 E2 ≡ S1

§4С.2 К 'ПЕРВСИМ' Е1 ≡

то во всех случаях, когда выражение - аргумент начинается с символа, сработает §4С.1 . И только когда §4С.1 не применим, сработает §4С.2 . В таком положении §4С.2 можно прочитать так: "Во всех остальных случаях заменить конкретизируемое выражение на пусто".

Мы приходим, таким образом, к следующему правилу, касающемуся расположения предложений: сначала - частные случаи, затем - общие предложения.

7. Рекурсивные функции

Человек, который хочет описать на языке M какой-либо алгоритм – например, алгоритм выделения первого символа выражения – должен написать некоторое число предложений для поля памяти машины C_M и некоторое выражение (по существу, начальные данные) для поля зрения. Затем надо запустить машину C_M, и она выполнит требуемое преобразование. Очевидно, что для того, чтобы правильно описать алгоритм, необходимо совершенно ясно представлять, как будет работать машина C_M с данным набором предложений в поле памяти. Это ставит человека в трудное положение, ибо имитируя работу машины C_M, он должен на каждом шаге просматривать сверху донизу все предложения, стоящие в поле памяти, чтобы найти первое из них, которое окажется применимым. И если число предложений в поле памяти велико, а для сложных алгоритмов оно, несомненно будет велико, – задача написания алгоритма может оказаться практически неразрешимой. Следовательно, необходим какой-то способ, с помощью которого можно было бы на каждом шаге исключать из рассмотрения все лишние предложения, и оставить лишь небольшую, хорошо обозримую группу предложений, имеющих отношение к данному акту конкретизации. Иными словами, надо как-то разбить предложения на небольшие группы, и иметь полную уверенность в их независимости друг от друга.

Способ разрешения этой проблемы напрашивается уже из рассмотрения предложений §4С.1 и §4С.2, описывающих понятие "первый символ выражения", иными словами, алгоритм выделения первого символа выражения. В левой части обоих предложений непосредственно за знаком $\underline{К}$ следует символ 'ПЕРВСИМ'. Поэтому эти предложения могут оказаться подходящими для конкретизации только в том случае, если в конкретизируемом выражении непосредственно вслед за $\underline{К}$ стоит тот же

символ 'ПЕРВСИМ' . Допустим, что в дополнение к понятию 'ПЕРВСИМ' мы пожелали описать понятие 'ПОСЛСИМ' -выделение последнего символа выражения - с помощью следующих двух предложений:

§ 5.1 К 'послсим' Е1 ≤ 2 ≥ ≤ 2

§ 5.2 К 'послсим' Е1 ≥

Эту группу предложений можно поместить в поле памяти вместе с группой 'ПЕРВСИМ' в любом положении относительно нее. Если конкретизируемое выражение имеет вид К 'ПЕРВСИМ'<Е>. , где <Е> - произвольное выражение,[x] то можно принимать в расчет лишь группу предложений § 4С.1 и § 4С.2 ; если оно имеет вид К 'ПОСЛСИМ'<Е>., то идут в расчет только § 5.1 и § 5.2.

Итак, мы договоримся, что во всех предложениях непосредственно за знаком К в левой части будет следовать какой-либо символ. Этот символ мы будем называть детерминативом предложения. Тем самым все предложения разбиваются на группы, образованные предложениями с одним и тем же детерминативом. Порядок расположения в памяти групп предложений с различными детерминативами несущественен. Порядок расположения предложений внутри группы, вообще говоря весьма существенен, и мы будем отражать его в нумерации предложений, помещая после точки порядковый номер предложения в группе. Последовательность знаков, предшествующая точке, нумерует группу предложений с данным детерминативом. Опуская часть номера, начинающуюся с точки, мы ссылаемся на группу предложений в целом.

Группа предложений определяет функцию, определенную на некотором множестве выражений и принимающую значение из того же множества.

---

x/ С помощью угловых скобок мы вводим в описание метаязыковые по отношению к языку рефал объекты.

Действительно, пусть у нас есть некоторая группа предложений с детерминативом φ и некоторое выражение $\langle\mathcal{E}\rangle$. Поместим в поле памяти машины C_M эту группу предложений, а в поле зрения - выражение

$$\underline{\kappa}\,\varphi\langle\mathcal{E}\rangle\underline{.}$$

после чего включим машину C_M. Если машина C_M, совершив определенное число шагов, придет в состояние нормальной остановки (вызванное отсутствием в поле зрения знаков конкретизации), то выражение, оказавшееся в поле зрения, будет значением функции φ при аргументе $\langle\mathcal{E}\rangle$. Если машина C_M придет в состояние аварийной остановки, или же вообще не остановится никогда, функция φ при аргументе $\langle\mathcal{E}\rangle$ не определена. Обычной в математике функциональной записи $\varphi(\langle\mathcal{E}\rangle)$ соответствует в языке M запись $\underline{\kappa}\,\varphi\langle\mathcal{E}\rangle\underline{.}$. Это различие вызвано необходимостью отличать функциональные скобки от обычных ("объектных").

Теперь нам осталось сделать один только шаг, чтобы прийти к понятию <u>рекурсивной функции</u>. Вспомним, как мы решили проблему доопределения понятия 'ПЕРВСИМ', когда оно было определено лишь предложением §4С.1. Мы приняли решение считать первый символ пустым и в том случае, когда выражение пустое, и в том случае, когда оно начинается со скобки. Первое вполне логично, но второе заставляет подумать о более естественном решении. Естественно было бы определить первый символ выражения как первый символ первого подвыражения, не содержащего скобок, то-есть, подвыражения, образующего самую левую ветвь на дереве синтаксического анализа скобочной структуры. Например, у выражения $(AB(C))D$ первый символ будет A, а у выражения $(((X)))YZ$ первый символ - X. Что же касается выражения $()A$, то его первый символ, как первый символ пустого подвыражения, мы будем считать пустым.

Как написать предложение, порождающее нужный нам алгоритм?

Если выражение - аргумент начинается с левой скобки,то мы, очевидно, должны взять подвыражение, которое эта левая скобка - вместе с парной ей правой скобкой - ограничивает, и взять у этого подвыражения первый символ. Соответствующее предложение имеет вид:

$$\S\ 4D.2\ \underline{\subset}\ 'ПЕРВСИМ'(\underline{\in}1)\underline{\in}2 \supseteq \underline{\subset}\ 'ПЕРВСИМ'\ \underline{\in}1\ .$$

Тот факт, что $\underline{\in}1$ - не произвольная последовательность символов, а обязательно выражение, то есть последовательность, сбалансированная по скобкам, обеспечивает правильность сопоставления скобок в конкретизируемом выражении скобкам в левой части предложения. Если какая-то скобка в поле зрения отождествлена (поставлена в соответствие) с какой-то скобкой в левой части предложения, то парная ей скобка в поле зрения может отождествиться только с парной скобкой в левой части.

Полный набор предложений, описывающих понятие (функцию) 'ПЕРВСИМ' таков:

$$\S\ 4D.1\ \underline{\subset}\ 'ПЕРВСИМ'\ \underline{\subseteq}1\underline{\in}2 \supseteq \underline{\subseteq}1$$

$$\S\ 4D.2\ \underline{\subset}\ 'ПЕРВСИМ'(\underline{\in}1)\underline{\in}2 \supseteq \underline{\subset}\ 'ПЕРВСИМ'\ \underline{\in}1\ .$$

$$\S\ 4D.3\ \underline{\subset}\ 'ПЕРВСИМ' \supseteq$$

В определении функции 'ПЕРВСИМ' имеет место - благодаря $\S\ 4D.2$ - рекурсия: при замене левой части предложения на правую функция вызывает саму себя (при другом значении аргумента). Пусть в поле зрения стоит выражение

$$\underline{\subset}\ 'ПЕРВСИМ'\ ((ZY)++)AB\ .$$

на первом шаге будет применен $\S\ 4D.2$, свободная переменная $\underline{\in}1$ примет значение $(ZY)++$, а $\underline{\in}2$ - значение AB , и в поле зрения окажется выражение:

$$\underline{\subset}\ 'ПЕРВСИМ'\ (ZY)++\ .$$

После второго шага будем иметь

```
К 'ПЕРВСИМ' ZY .
```

Теперь будет применен §4D.1 с результатом

```
Z
```

Функции, определенные с помощью подстановок, включающих, в частности, возможность рекурсии, называются рекурсивными функциями. Подчеркнем, что этот термин относится к способу определения, задания функции. Рекурсия может быть не прямая, а косвенная, через посредство других функций. В следующем примере

```
§ К α Е1 ≥ К β (Е1) .
§ К β (S1 Е2) ≥ К γ Е2 .
§ К γ Е1 S2 ≥ К α Е2 .
```

функция α вызывает функцию β, которая вызывает функцию γ, а эта последняя функция снова обращается к α. Таким образом, определение функции α содержит рекурсию.

Требование наличия рекурсии не является обязательным, это - общее понятие; функция 'ПЕРВСИМ', определенная предложениями §4C также относится к множеству рекурсивных функций. Однако для множества рекурсивных функций в целом возможность использования рекурсии имеет решающее значение. Не используя рекурсии, мы не смогли бы, например, описать функцию, которая, подобно функции §4D, обеспечивает вхождение в скобки на любую глубину. Без рекурсии можно написать предложение, обеспечивающее вход в скобки первого уровня:

```
§ К 'ПЕРВСИМ' (S1 Е2) Е3 ≥ S1
```

или второго, третьего и т.д. уровней:

```
§ К 'ПЕРВСИМ' ((S1 Е2) Е3) Е4 ≥ S1

§ К 'ПЕРВСИМ' (((S1 Е2) Е3) Е4) Е5 ≥ S1
```

или, наконец, обеспечить - путем введения в поле памяти всех этих предложений - возможность входа в скобки на глубину, не превышающую заданной; но описать функцию, входящую на любую глубину, без рекурсии невозможно. Нетрудно понять причину этого. Пусть мы имеем некоторое множество (конечное, разумеется) функций, из которых одни, быть может, выражаются через другие, но таким образом, что это не приводит к рекурсии. Следовательно, можно построить такой граф без петель, у которого вершины будут соответствовать функциям, а от вершины φ к вершине ψ стрелка будет идти в том и только в том случае, если какое-либо из предложений с детерминативом φ содержит в правой части вызов функции ψ. Теперь докажем, что для каждой функции φ существует такое число $N(\varphi)$, что выполнение конкретизации выражения вида $\underline{\kappa}\,\varphi\langle\mathfrak{C}\rangle\pm$, где $\langle\mathfrak{C}\rangle$ - произвольное выражение, не содержащее знаков $\underline{\kappa}$, потребует не больше чем $N(\varphi)$ шагов машины C_M. Для этого введем понятие ранга функции. Тем функциям, которые не вызывают никаких функций, припишем (пишу) ранг I, функции, которая вызывает ν функций рангов $r_1 \le r_2 \le \ldots \le r_\nu$, припишем ранг $r_\nu + 1$; (это возможно благодаря отсутствию у графа петель). С помощью индукции по рангу функции покажем, что для всех функций ранга r число $N(\varphi)$ не превышает некоторого максимального числа $\bar{N}(r)$. Для любой функции φ ранга I $N(\varphi)$, очевидно, равно I, ибо у них правые части предложений вообще не содержат знаков конкретизации. Следовательно, $\bar{N}(1) = 1$. Но $\bar{N}(r+1) \le k(r)\bar{N}(r)$, где $k(r)$ - максимальное число знаков конкретизации в правой части предложения с детерминативом ранга r. Таким образом, функция, определенная без рекурсии, может заставить машину C_M совершить не более чем некоторое определенное число шагов при любом аргументе.

Функции же, подобные функции § 4Д , могут потребовать для конкретизации сколь угодно большого числа шагов - в зависимости от значения аргумента.

Итак, язык М оказался языком рекурсивных функций, определенных на множестве произвольных символьных выражений. С такой областью определения связывается понятие алгоритма - в отличие от классических рекурсивных функций, определенных на множестве натуральных чисел. Поэтому мы дадим этому языку название "алгоритмический язык рекурсивных функций", сокращенно - РЕФАЛ.

Продемонстрируем близость записи алгоритмов на рефале к обычным рекурсивным определениям, используемым в математике. Возьмем классический пример - определение числа и действий над числами в рекурсивной арифметике. Определение числа основывается на объектной постоянной нуль и одноместной функции следования. Нуль мы будем обозначать, как обычно, знаком 0. Объект, следующий за произвольным объектом a , обозначают обычно через a' . Мы же, чтобы не путать знак ' (штрих) с рефал-кавычкой, будем пользоваться вместо штриха буквой 1 . Итак, 01 будет изображать 1, 011 - 2 и т.д.

Определение числа сотоит из следующих трех пунктов:

I. Нуль есть число.

2. Объект, следующий за числом, есть число, иначе говоря, $a1$ есть число, если a - число.

3. Объект, который с помощью первых двух пунктов не может быть квалифицирован как число, не есть число.

Чтобы записать это определение на рефале, мы должны ввести предикат 'ЧИСЛО' , то-есть, такую функцию, которая принимает значение 'ИСТИНА' , если ее аргумент есть число, и значение 'ЛОЖЬ' - в противном случае.

Описание этого предиката на рефале почти буквально повторяет приведенное выше словесное определение:

```
§6.1 К 'число' 0 = 'истина'
§6.2 К 'число' Ea. = К 'число' Ea.
§6.3 К 'число' Eв = 'ложь'
```

Рекурсивное определение сложения состоит из двух пунктов:

1. $+(a, 0) = a$
2. $+(a, b1) = (+(a, b))1$

(Мы записываем сложение как двуместную функцию, сохранив знак + в качестве символа функции).

Описание сложения на рефале имеет вид:

```
§7.1 К + Ea, 0 = Ea
§7.2 К + Ea, Eв. = К + Ea, Eв.1
```

Предложения § 6 и § 7 дают рефал-машине полную информацию, необходимую для выполнения алгоритмов распознавания числа и сложения чисел. Пусть, например, в поле зрения введено выражение:

```
К + 011, 0111.
```

Работа рефал-машины может быть описана следующей табличкой (табл.1.2).

Таблица I.2 Выделение рефал-машиной операции сложения 2 + 3 = 5

Номер шага i	Используемое предложение	Состояние поля зрения после i-го шага
		$\underline{K}$ + 0 1 1 , 0 1 1 1 .
I	§ 7.2	$\underline{K}$ + 0 1 1 , 0 1 1 . 1
2	§ 7.2	$\underline{K}$ + 0 1 1 , 0 1 . 1 1
3	§ 7.2	$\underline{K}$ + 0 1 1 , 0 . 1 1 1
4	§ 7.I	0 1 1 1 1 1

Задачи

I.I. Описать на рефале рекурсивное умножение.

I.2. При том частном способе записи операции следования, который мы использовали выше, становится возможным описание на рефале алгоритма сложения чисел без рекурсии. Дать это описание. Было бы возможно такое описание, если бы число, следующее за числом a, записывалось в виде $1(a)$?

I.3. Описать на рефале "усеченное вычитание" $\dot{-}$, результатом которого является абсолютная величина разности аргументов.

8. Еще о некоторых чертах рефала

Если в левую часть предложения некоторая свободная переменная входит более одного раза, то всем вхождениям этой переменной должны соответствовать в поле зрения совпадающие выражения. Пользуясь этим, можно следующим образом определить предикат равенства:

$$\S\ \underline{\underline{K}} = (\underline{\underline{E}}a)\,(\underline{\underline{E}}a) \Rightarrow \text{'ИСТИНА'}$$

$$\S\ \underline{\underline{K}} = (\underline{\underline{E}}a)\,(\underline{\underline{E}}b) \Rightarrow \text{'ЛОЖЬ'}$$

Сравниваемые выражения мы разделяем здесь не запятой, как при записи функции сложения чисел, а путем заключения их в скобки. Это связано с тем, что наш предикат рассчитан на совершенно произвольные выражения-аргументы, и может оказаться, что они содержат запятые, что может привести к неверному результату.

Символ, терм и выражение - основные синтаксические понятия рефала; с помощью соответствующих трех типов свободных переменных мы записываем левые части предложений как некие обобщенные образы, как <u>типовые выражения</u> для конкретизируемых объектов, рассматриваемых с точки зрения синтаксиса рефала, то-есть, простого скобочного синтаксиса. Однако конкретизируемые объекты могут обладать и более

сложным синтаксисом, включающим набор синтаксических понятий, специфических для языка, в рамках которого эти объекты возникли. В таких случаях бывает удобно описывать преобразования объектов в терминах данного специфического синтаксиса, то-есть, вводить в левую часть предложения свободные переменные различных типов, соответствующие данным синтаксическим понятиям. Например, в языке алгол-60 вводится множество синтаксических понятий, например: буква, знак аддитивной операции (+ или -), идентификатор, арифметическое выражение и т.п. Было бы удобно описывать трансляцию с алгола, используя свободную переменную аддитивной операции, значением которой может быть один из указанных двух знаков, или свободную переменную идентификатора и т.п. Заметим, что главный аспект синтаксического отождествления - это синтаксический анализ объекта, а элемент распознавания частного случая - лишь результат синтаксического анализа. Если, например, левая часть имеет вид

$$\S \leq \varphi(\underline{E}1)\, z\, \underline{S}2\, \underline{E}3 \geq$$

то задача отождествления состоит не только в том, чтобы убедиться, что конкретизируемое выражение имеет данный вид, но и в том, чтобы путем его синтаксического анализа придать значения переменным $\underline{E}1$, $\underline{S}2$, $\underline{E}3$, спроектировать их на определенные подвыражения конкретизируемого выражения. Только в процессе этого анализа и можно убедиться в том, что объект подходит под данный образец. Используя свободные переменные вида $\underline{S}\langle x\rangle$, $\underline{W}\langle x\rangle$, $\underline{E}\langle x\rangle$, где $\langle x\rangle$ - объектный знак (такие переменные мы будем называть базисными), мы производим анализ в рамках простого свободного синтаксиса. Этот синтаксис, несмотря на свою простоту, предоставляет весьма широкие

возможности для конструирования языковых объектов при сохранении наглядности. Когда программисту нужен внутренний язык, он может употреблять конструкции, которые легко анализируются с помощью базисных переменных. Такие конструкции легко создаются из скобок и специальных символов - синтаксических указателей. Например, число можно всегда помечать звездочкой в начале и заключать в скобки: (* 3.14159) и т.п. Тогда, чтобы распознать число и выделить его содержательную часть, достаточно в левой части предложения употребить комбинацию вида (* e1) . По этой причине даже при ограничении свободных переменных одними только базисными переменными, рефал позволяет довольно компактно и наглядно описывать сложные преобразования. Однако, переменные специальных типов все-таки создают важные дополнительные возможности (особенно, когда синтаксис диктуется извне).

В отличие от двучленных конструкций $S\langle x\rangle$ и т.д., условимся изображать переменные специального синтаксического типа трехчленными конструкциями (средний член назовем спецификатором):

$$S\,(\blacksquare)\langle x\rangle\ ,\quad W\,(\blacksquare)\langle x\rangle\ ,\quad E\,(\blacksquare)\langle x\rangle$$

Здесь заштрихованный прямоугольник обозначает какую-то информацию, необходимую для различения одних синтаксических типов от других. Какова же должна быть эта информация? Каждая свободная переменная специального синтаксического типа связана с алгоритмом распознавания или выделения объекта данного синтаксического типа. Естественно сам этот алгоритм описать на рефале, связав с ним определенную рекурсивную функцию! Тогда информация ▨ должна включать детерминатив этой функции и, может быть, еще какие-то дополнительные указания.

Пусть, например, надо распознать конструкцию, начинающуюся с идентификатора[x/], за которым следует знак +. Левая часть предложения, выполняющая эту задачу, будет иметь вид

$$\underline{<}\ \varphi\ \underline{\in}\ (\blacksquare)1 + \underline{\in} 2\ \underline{\geq}$$

Чтобы конкретизировать ▨, мы должны ввести рекурсивную функцию, описывающую процесс выделения идентификатора. Присвоим этой функции детерминатив 'ИД'. Как фиксировать результат применения этой функции? Простейший способ таков: пусть функция 'ИД' отщепляет от начала своего аргумента идентификатор и заключает его в скобки. Следовательно, конкретизация

$$\underline{<}\ 'ид'\ sum1 + 0.5/x\ \underline{.}$$

должна давать

$$(sum1) + 0.5/x$$

Необходимо также принять какое-то соглашение на тот случай, когда отщепить идентификатор от начала аргумента невозможно. Пусть функция 'ИД' приписывает в этом случае знак ¬ (логическое отрицание) к аргументу перед его началом. Например,

$$\underline{<}\ 'ид'\ (a+b)c - d\ \underline{.}$$

будет заменено на

$$\neg (a+b)c - d$$

Теперь функция 'ИД' полностью определена, и было бы нетрудно составить её описание на рефале. Если это описание введено в поле

---

x/ Напомним, что идентификатор в алголе-60 - это произвольная последовательность букв и цифр, начинающаяся с буквы.

памяти рефал-машины, то становится законным использование свободной переменной выражения типа ⟨идентификатор⟩ , то-есть, например, приведенной выше левой части предложения, в которой заштрихованный прямоугольник заменен на 'ИД'

$$\leq \varphi \, E('\text{ид}')1 + E2 \geq$$

а также

$$\leq \psi \left(E('\text{ид}')1 \times E('\text{ид}')2 \right) / E3 \geq$$

и т.п.

Чтобы рефал-машина могла использовать такие предложения, она должна предпринимать более сложные действия, чем в случае наличия лишь базисных переменных. А именно, дойдя в процессе отождествления, то-есть проектирования левой части предложения на участок поля зрения, до свободной переменной вида

$$E(\langle e \rangle)\langle x \rangle$$

где $\langle e \rangle$ - выражение (начинающееся с детерминатива), рефал-машина временно приостанавливает отождествление и приступает к выполнению конкретизации

$$\leq \langle e \rangle \langle e_0 \rangle .$$

где $\langle e_0 \rangle$ - отрезок поля зрения, на который надо спроектировать данную свободную переменную и следующие за ней (еще не спроектированные) элементы левой части предложения[x/]. Если результат конкретизации начинается с левой скобки, то-есть имеет вид

$$(\langle e_1 \rangle)\langle e_2 \rangle$$

то выражение $\langle e_1 \rangle$ объявляется проекцией (значением) свободной переменной, и процесс отождествления продолжается. Если результат начинается с символа, проектирование невозможно (символ может быть

x/ Под элементами левой части мы понимаем знаки $\leq$, $\geq$, скобки, свободные переменные и конкретные символы.

любым, в формальном описании, он обозначается через $< S_{\pi} >$; на практике мы всюду используем с этой целью знак ¬). Мы видим, что трехчленные переменные порождают еще один уровень рекурсии, ибо для определения одной лишь применимости предложения они отсылают рефал-машину к выполнению полного шага, или многих шагов. Поэтому мы будем называть их рекурсивными свободными переменными.

Если информация к рекурсивной переменной состоит из одного детерминатива (как в приведенном выше примере) мы разрешим для краткости опускать скобки в спецификаторе; таким образом, E 'ИД' I полностью эквивалентно E ('ИД')I. Распознаются эти рекурсивные переменные по тому признаку, что за указателем переменной следует не объектный знак, а составной символ.

Синтаксические типы, которые с точки зрения рефала являются, вообще говоря, выражениями, (например, идентификатор) требуют для своего определения рекурсивной функции, производящей выделение (отщепление) нужного объекта. Те же синтаксические типы, которые в смысле рефала заведомо являются символами или термами (например, знак аддитивной операции) выделяются в процессе отождествления на основании своего рефал-синтаксиса, и описывающая их рекурсивная функция должна производить лишь проверку того, что данный символ или терм подходит под определение синтаксического типа. Можно было бы договориться, что соответствующая функция приписывает к аргументу определенные символы, например 'ДА' или 'НЕТ' в зависимости от результата проверки, однако для единообразия мы будем в случае положительного ответа заключать объект в скобки, а в случае отрицательного ответа - предварять его символом ¬ Тогда рефал-машина будет реагировать на рекурсивные переменные символа и терма аналогично тому, как она реагирует на свободные переменные выражения.

Если, например, функция 'АДДОП' - "аддитивная операция"-описана следующим образом:

```
§ К 'АДДОП' + ≡ (+)

§ К 'АДДОП' - ≡ (-)

§ К 'АДДОП' S1 ≡ ¬ S1
```

то в левой части рефал-предложения можно употребить свободную переменную S 'АДДОП' I, которая будет проектироваться только на знак + или -. Механизм проектирования таков. Рефал-машина в процессе отождествления левой части, прежде чем присвоить переменной S 'АДДОП' <∝> значение очередного символа < S >, породит выражение

```
К 'АДДОП' < S > .
```

и выполнит эту конкретизацию. Только тот символ < S > будет сочтен подходящим, при котором результатом конкретизации будет выражение (в частности, конечно, символ), взятое в скобки.

Пример левой части, использующей рекурсивную переменную символа:

```
К φ Е1 S 'АДДОП' A Е2 ≡
```

Более полное и строгое определение рекурсивных переменных мы отложим до Формального описания.

Если левая часть предложения содержит более одной свободной переменной выражения, то может случиться, что существует несколько вариантов приписывания свободным переменным значений, которые приводят к отождествлению конкретизируемого выражения с левой частью предложения. Пусть, например, в поле памяти рефал-машины находится предложение с левой частью

```
K Ф E1, W2 E3 =
```

а в поле зрения - выражение

```
K Ф A, B B, (?) C Д .
```

Оно может быть отождествлено с левой частью таким образом, что E1 примет значение A, W2 - значение B, а E3 - значение B, (?)СД. Возможен и второй вариант: E1 принимает значение A, BB, W2 - значение (?), E3 - значение СД. Следовательно, необходимо договориться, как поступает рефал-машина при наличии такой неоднозначности. Соглашение, которое мы примем, состоит в том, что в подобных ситуациях рефал-машина приписывает свободным переменным выражения тем более короткие значения, чем раньше (левее) эти переменные появляются в левой части предложения. В нашем примере рефал-машина выберет первый вариант. Иначе говоря, рефал-машина просматривает левую часть предложения слева направо и каждую встречающуюся свободную переменную выражения набирает последовательно - терм за термом - до первой возможности продвинуться дальше в процессе отождествления. Пользуясь этим, мы можем, например, записать алгоритм устранения запятых из выражения (без вхождения в скобки) следующим образом:

```
§ K 'УСТРЗАП' E1, E2 = E1 K 'УСТРЗАП' E2 .

§ K 'УСТРЗАП' E1 = E1
```

В первом предложении мы вынесли переменную E1 из области действия знака K, ибо она заведомо не содержит запятых (E1 отщепляется до первой запятой).

Принцип просмотра левой части предложения слева направо в процессе отождествления можно было бы при Формальном описании рефала

выдержать с абсолютной, догматической неукоснительностью. Однако это вряд ли целесообразно. Рассмотрим в качестве примера проектирование левой части

$$\underline{K}\ \text{'ПОСЛСИМ'}\ \underline{E}1\ \underline{S}2\ \geq$$

на выражение

$$\underline{K}\ \text{'ПОСЛСИМ'}\ a\,b\,(c+d)\,b\,x\ .$$

Согласно алгоритму работы рефал-машины синтаксическое отождествление начинает осуществляться после того, как в поле зрения фиксируется ведущий знак $\underline{K}$ и парная ему точка $.$, которые сопоставляются знакам $\underline{K}$ и $\geq$ в левой части предложения. Следовательно, исходная позиция проектирования может быть изображена следующим образом:

$$\underline{K}\ \text{'ПОСЛСИМ'}\ \underline{E}1\ \underline{S}2\ \geq$$

$$\underline{K}\ \text{'ПОСЛСИМ'}\ a\,b\,(c+d)\,b\,x\ .$$

Продвигаясь в левой части слева направо, проектируем детерминатив 'ПОСЛСИМ' на тождественный ему символ в поле зрения. Теперь, если строго соблюдать принцип просмотра слева направо, надо было бы начать с того, что придать переменной $\underline{E}1$ пустое значение, переменную $\underline{S}2$ спроектировать на символ a и убедиться, что так отождествления не получается. Затем надо придать $\underline{E}1$ значение a $\underline{S}2$ – значение b убедиться, что отождествление снова зашло в тупик, и так удлинять (набирать) значение переменной $\underline{E}1$, пока $\underline{S}2$ не окажется спроектированным на последний символ x

Ясно, что транслятор, работающий таким образом был бы никуда негодным. Но и для описательных целей такой процесс неудобен. Отщепление последнего символа удобнее представлять как движение справа налево от конкретизационной точки. Поэтому в Главе 2 мы определим

синтаксическое отождествление как последовательность актов проектирования элементов левой части, придерживаясь следующего общего принципа: когда можно спроектировать элемент без удлинения базисных свободных переменных выражения Ξ<α> , это проектирование производится; когда это сделать невозможно, начинается набор значений переменных Ξ<α> . Кроме того, базисным переменным символа и терма отдается предпочтение, в смысле очередности проектирования, перед всеми рекурсивными переменными. Понятие "можно спроектировать" требует, разумеется, уточнения. Мы считаем, что "можно спроектировать" символ, когда он примыкает слева или справа к уже спроектированному элементу, и что "можно спроектировать" скобку, когда парная ей скобка уже спроектирована. Эти возможности порождают аналогичные возможности проектирования свободных переменных символа и терма. Например, в приведенном выше примере переменная S 2 может быть спроектирована благодаря наличию проекции у знака ≥ Аналогично функциональным скобкам, обычные (объектные) скобки выполняют роль опорных точек при проектировании. При прочих равных возможностях предпочтение отдается элементу, расположенному левее.

Перечисленные принципы, уточненные и дополненные в Формальном описании, порождают определенную последовательность проектирования элементов левой части, которая, вообще говоря, не совпадает с последовательностью расположения элементов между знаками К и ≥ Номер элемента в этой последовательности мы назовем его проекционным номером. В нашем примере проекционные номера распределятся следующим образом:

```
          1     3  2
К 'послсим' Ξ1 S 2 ≥
```

Приведем и прокомментируем более сложный пример:

```
    1 2    4      5 6 12  11  10  9 7 8  3
K φ A W(α S X)1 EA (E Z  W2 EA W3 ) EC S X =
```

Проекционный номер 3 имеет переменная S X а не W(α S X)1 ей отдается предпочтение, так как она - базисная. Смысл этого становится виден, когда мы приступаем к проектированию рекурсивной переменной номер 4. В её спецификатор входит свободная переменная S X Это не вызывает трудностей, так как она уже приняла определенное значение. Для проверки заданного терма <W> на годность в качестве значения рекурсивной переменной рефал-машина выполняет конкретизацию:

K α <S$_x$> <W> .

где <S$_x$> - значение переменной S x . Если бы в момент проектирования переменной W (α S X) 1 переменная S X еще не имела значения, проектирование было бы невозможно. Такая ситуация квалифицируется как синтаксическая ошибка.

Переменная EA сначала принимает пустое значение, а затем удлиняется, пока не станет возможным отождествление. Такие переменные называются открытыми.

Спроектировав скобку 6, мы тут же проектируем скобку 7. Теперь у переменной EC оказываются спроектированными оба конца, почему ей и присваивается номер 8. Такие переменные называются закрытыми. Продвигаясь от скобки 7 влево мы проектируем W 3 , а затем EA. Последнее возможно потому, что переменная EA уже приняла определенное значение. Данное ее вхождение является повторным. Поэтому надо только проверить, действительно ли к терму W3 примыкает слева заданное выражение. Проектируя W2 , мы обнаруживаем, что E Z также оказы-

вается спроектированным (закрытая переменная).

Упомянем, наконец, еще об одной черте рефала. Чтобы обеспечить возможность самопреобразования программы, вводится два символа обмена - '§' и '≥' (ради экономии собственных знаков они изображаются через посредство уже введенных знаков). Символы обмена выступают в роли детерминативов функций, которые выполняются рефал-машиной без описания с помощью предложений. Функция '§' делает в поле зрения доступным набор предложений, находящихся в поле памяти. Функция '≥' заносит в поле памяти указанный набор предложений. Так как не все знаки, которые встречаются в предложениях, могут быть записаны в поле зрения, при обмене между полем памяти и полем зрения производится метакодовое преобразование.

СОДЕРЖАНИЕ

№ Т 10000 от "14" VII 1971 г. Заказ № 786 Тираж 250 экз.

Ордена Ленина институт прикладной математики
Москва, Миусская пл., 4