БИБЛИОТЕЧКА ПРОГРАММИСТА

В. Н. ПИЛЬЩИКОВ

Язык плэнер

БИБЛИОТЕЧКА
ПРОГРАММИСТА

В. Н. ПИЛЬЩИКОВ

ЯЗЫК ПЛЭНЕР

МОСКВА «НАУКА»
ГЛАВНАЯ РЕДАКЦИЯ
ФИЗИКО-МАТЕМАТИЧЕСКОЙ ЛИТЕРАТУРЫ
1983

22.18
П 32
УДК 519.6

Язык плэнер. Пильщиков В. Н.— М.: Наука. Главная редакция физико-математической литературы, 1983.— 208 с.

В книге описана одна из версий хорошо известного в области искусственного интеллекта языка программирования planner. В этом языке наряду с традиционными методами обработки символьной информации используется много новых концепций (поиск с возвратами, ассоциативная выборка данных и процедур, дедуктивные выводы и др.), учитывающих специфику задач искусственного интеллекта.

Рис. 6. Библ. 17 назв.

П $\frac{1702070000 - 154}{053(02)-83}$ 36-83

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Для большинства систем искусственного интеллекта (ИИ) характерны большие размеры и сложная структура их программ, что затрудняет их реализацию и отладку. В то же время практически все системы ИИ являются экспериментальными, с их помощью проверяются те или иные методы и гипотезы, что, естественно, требует сокращения времени от «зарождения» системы до ее полной реализации. Добиться этого можно только при адекватных инструментальных средствах — языках программирования с высокой изобразительной силой, учитывающих специфику задач ИИ.

К сожалению, в нашей стране при реализации систем ИИ нередко используются не эти языки, а языки, которые имеются «под рукой», даже язык ассемблера или фортран. В то же время существуют языки, специально предназначенные для задач ИИ. Это язык лисп (см., например, [1, 2]), созданный еще в 1960 г. и до сих пор остающийся наиболее популярным у зарубежных специалистов по ИИ, и так называемые языки для искусственного интеллекта, появившиеся в 70-х гг.: planner [3, 4], conniver [5], QA-4 [6], qlisp [7], KRL [8], FRL [9], Ф-язык [10], prolog [11] и др. Эти языки включают в себя понятия и методы, часто используемые в системах ИИ (фреймы, ассоциативный выбор данных и процедур, автоматический поиск с возвратами, дедуктивные механизмы, «демоны» и т. д.), что снимает с пользователя многие заботы технического характера и позволяет сосредоточить внимание на принципиальных аспектах разрабатываемой системы ИИ.

В книге, предлагаемой вниманию читателя, описывается реализованный в СССР диалект языка planner — одного из наиболее известных языков для ИИ. Этот язык был разработан в 1967—1971 гг. в Лаборатории искусственного интеллекта Массачусетского технологического института американским ученым Карлом Хьюиттом. В языке удачно соединены возможности языка лисп, методы анализа данных по образцам и ряд новых для алгоритмических языков концепций (поиск с возвратами, вызов процедур

по образцу, дедуктивный механизм и др.), которые обобщают применяемые в системах ИИ методы описания задач и поиска их решения. Такое сочетание, с одной стороны, делает planner мощным языком программирования для задач символьной обработки, а с другой — придает ему свойства системы ИИ, способной самостоятельно находить решение задач по их описанию. Язык ориентирован в основном на реализацию систем планирования действий робота (отсюда и название языка — «планировщик»), автоматического доказательства теорем, понимания текстов на естественном языке, вопросно-ответных систем и т. п.

Следует, однако, отметить, что в том виде, как он описан автором, язык planner так и не был реализован. Это объясняется как громоздкостью описанного языка, так и тем, что фактически была предложена некоторая схема, которую надо было еще значительно уточнять, чтобы получилось полное и строгое описание языка. Реализованы были только подмножества, диалекты языка. Первым из них был язык micro-planner [12], который представляет собой «надстройку» над лиспом, включающую минимальный набор новых концепций planner'а. Этот диалект был успешно применен для реализации ряда систем ИИ, в частности, широко известной системы Т. Винограда [13]. Другой диалект — язык popler 1.5 [14], являющийся довольно полным вариантом planner'а и включающий в себя некоторые элементы языка conniver.

Диалектом planner'а является и язык, предлагаемый вниманию читателей. Он получил название плэнер и был реализован на ЭВМ БЭСМ-6 [15, 16]. В плэнере в полном объеме сохранены основные свойства языка planner, но исключены параллельные процессы и ряд более второстепенных деталей, несколько изменены синтаксис языка и определения отдельных встроенных процедур. Все эти изменения преследовали цель упростить язык и повысить эффективность его реализации. Учитывался и практический опыт применения как самого плэнера, так и языков micro-planner и popler 1.5.

В плэнере можно выделить пять более или менее независимых частей. Каждой из них посвящена отдельная глава книги.

В гл. 1 описывается «лисповская» часть языка. Плэнер содержит в качестве своего подмножества практически весь лисп (с некоторыми модификациями) и во многом сохраняет его специфические черты; это объясняется желанием сохранить достоинства языка, широко известного среди разработчиков систем ИИ.

В гл. 2 рассматриваются образцы и их применение для анализа данных. Как показывает опыт программирования на лиспе, особенно в области ИИ, описание правил анализа данных в виде лисповских функций не всегда наглядно и компактно. В то же

время в языках обработки строк (например, в сноболе, рефале) уже давно используется более удобное средство анализа — образцы, которые представляют собой шаблоны, накладываемые на анализируемые объекты с целью определить, имеют ли эти объекты нужную структуру. Подобное же средство включено и в плэнер.

Описанное в первых двух главах книги подмножество плэнера может использоваться независимо от других его средств и представляет собой мощный язык программирования, удобный для реализации систем символьной обработки. Остальные же части плэнера ориентируют его на область ИИ. Они дают средства для описания задач (исходных ситуаций, допустимых операций, целей), решение которых должна найти система ИИ, реализуемая на плэнере, и средства, упрощающие программирование процедур поиска решения задач.

В гл. 3 описывается механизм поиска с возвратами (backtracking), который в плэнере назван режимом возвратов. Это такой способ выполнения программы, при котором ЭВМ может принимать пробные решения, а затем отказываться от них, если они завели в тупик. Во многих системах ИИ применяются методы поиска решения задач, основанные на переборе вариантов. Для того чтобы избавить разработчиков этих систем от необходимости каждый раз заново реализовывать такой перебор, в плэнер и введен режим возвратов, который берет на себя ответственность за организацию перебора. Вставляя в этот механизм подходящие операции, пользователь языка может легко получать конкретные алгоритмы перебора.

В гл. 4 рассматривается база данных, которая используется в языке для описания того внешнего мира, в котором система ИИ решает свои задачи (например, той обстановки, в которой действует робот). База данных представляет собой совокупность так называемых утверждений, каждое из которых описывает какой-то отдельный факт обстановки, а в целом они описывают всю обстановку. Язык обеспечивает хранение такой базы данных и предоставляет соответствующие операции для работы с ней.

С помощью базы данных можно представить только конкретные факты о конкретных объектах, а при описании задачи должны быть указаны и «правила игры»: логические отношения между используемыми понятиями (например, между понятиями «слева» и «справа») и операции, которые разрешено применять в задаче (например, действия робота). Такие правила описываются в плэнере в виде специальных процедур, названных теоремами. Особенностью теорем является то, что они вызываются не по имени, а по образцу: в программе в виде образца описывается некоторая цель и среди имеющихся теорем отыскивается и вычисляется та, результат работы которой соответствует данному образцу. Тем

самым автоматически выбирается «правило игры», которое обеспечивает достижение поставленной цели. Теоремы могут в свою очередь ставить цели, для достижения которых будут подыскиваться другие теоремы. Такое сведéние целей к подцелям в сочетании с режимом возвратов (нередко для достижения цели можно воспользоваться «услугами» нескольких теорем, поэтому приходится делать выбор) лежит в основе встроенного в язык дедуктивного механизма, обеспечивающего автоматический поиск решения задачи по ее описанию.

Теоремам и дедуктивному механизму посвящена заключительная глава книги.

Книга представляет собой учебное пособие по языку плэнер, содержащее полное его описание, которое сопровождается многочисленными примерами и объяснением приемов программирования на нем. Чтение книги не требует предварительного знания других языков, однако знакомство с языком лисп облегчит понимание материала. В книге нет сведений о работе с плэнер-системой, их можно найти в [16].

Автор выражает свою благодарность В. М. Юфа и М. Ю. Семенову за их полезные советы по улучшению практических свойств языка плэнер и по его реализации, а также В. Г. Сенину, прочитавшему рукопись книги и сделавшему ряд ценных замечаний. Автор особо благодарит М. Г. Мальковского за его большую помощь на всех этапах работы.

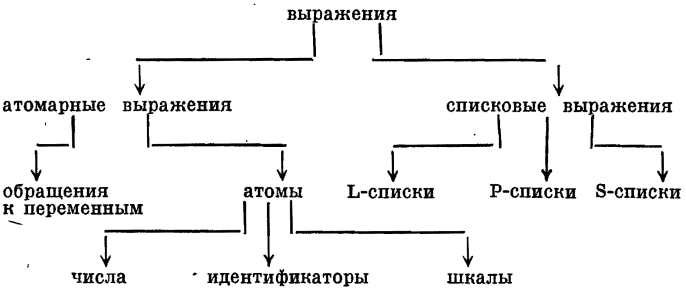
В. Н. Пильщиков

ГЛАВА 1

ФУНКЦИИ

1.1. Выражения

В языке плэнер программы и данные строятся из одних и тех же конструкций — в ы р а ж е н и й, классификацию которых можно представить в виде такой схемы:



Рассмотрим подробно указанные типы выражений.

В *алфавит* языка включаются любые символы, которые могут быть введены в ЭВМ. Символы алфавита подразделяются на следующие группы:

ограничители: ␣ ( ) [ ] ⟨ ⟩

цифры: 0 1 2 3 4 5 6 7 8 9

спецлитеры: + — . * : !

буквы: заглавные латинские и русские буквы, прочие символы (× / ÷ ↑ ∨ ∧ и др.).

Отметим, что пробел является символом алфавита и играет важную роль при записи выражений. Выше он для большей наглядности обозначен знаком ␣, однако в дальнейшем мы не будем пользоваться этим знаком, а будем просто оставлять пустой ту позицию в тексте, где должен находиться пробел.

Идентификаторы представляют собой непустые последовательности любых (кроме ограничителей) символов алфавита, начинаю-

щиеся с буквы. Идентификаторами также являются одиночные спецлитеры. Примеры идентификаторов:

АТОМ X15 =A+B= *

Числа в плэнере делятся на *целые* и *вещественные*. Целые числа представляются точно, точно выполняются и операции над ними. Представление же вещественных чисел и выполнение операций над ними в общем случае могут быть приближенными.

Запись целого числа представляет собой непустую последовательность цифр, перед которой может находится знак + или —. Например:

1983 —9 +3000000

Запись вещественного числа состоит из знака + или —, из цифр целой части, десятичной точки и цифр дробной части. Целая часть и точка должны присутствовать всегда, знак же и цифры дробной части можно опускать. Примеры:

2.71828 —0.0000005 +6. 0.6

Шкалы представляют собой последовательности битов (двоичных разрядов). Количество битов в шкалах предполагается фиксированным. Примером шкалы может служить следующая последовательность нулей и единиц:

000 000 000 000 001 010 011 100 101 110 111 000

Для краткости любая шкала записывается в языке не в двоичном виде, как выше, а в виде восьмеричного числа: каждая тройка битов шкалы заменяется на соответствующую восьмеричную цифру (000 — на 0, 001 — на 1, 010 — на 2, ..., 111 — на 7). Для того чтобы такую запись шкалы можно было отличить от записи целых десятичных чисел, перед восьмеричными цифрами ставится *. Незначащие восьмеричные нули в записи шкал можно опускать, кроме случая *0. Примеры записи шкал:

*5 *3704 *12345670 *000012345670

Последние две записи представляют одну и ту же шкалу, двоичный вид которой указан выше.

Атомы — это общее название идентификаторов, чисел и шкал; они являются простейшими выражениями языка.

Обращения к переменным. Как и в других языках программирования, *переменная* в плэнере — это объект программы, имеющий фиксированное имя и значение, которое может меняться. Именами переменных могут быть только идентификаторы, значениями — любые выражения. Никаких ограничений на типы принимаемых переменной значений в языке не накладывается, поэто-

му значением любой переменной может быть сначала идентификатор, затем список, затем число и т. д.

Во многих языках при обращении к переменной указывается только ее имя. В плэнере же при обращении к переменной необходимо поставить перед ее именем один из следующих *префиксов:*

. * : !. !* !:

Конструкция из такого префикса и имени переменной называется «обращением к переменной». Примеры обращений:

.Y :КОНСТ !*VAR !.Q+R

Для краткости мы будем, как правило, вместо словосочетания «обращение к переменной с префиксом *» употреблять термин «*-переменная», вместо «обращение к переменной с префиксом !.» — термин «!.-переменная» и т. п. Кроме того, обращения к переменным с префиксами «.», «*» и «:» будем называть *простыми обращениями к переменным*, а обращения с иными префиксами (начинающимися с восклицательного знака) — *сегментными обращениями к переменным.*

Префиксы в обращениях к переменным синтаксически отличают этот тип выражений от идентификаторов. Например, X — это сам идентификатор X, а .X — это обращение к переменной с именем X. Кроме того, префиксы указывают способы использования переменных. Так, обращение .X указывает, что нужно взять текущее значение переменной X, а обращение *X — что переменной X должно быть присвоено новое значение. Такая система обозначений очень удобна в образцах, которые играют важную роль в плэнере, поэтому она и принята на вооружение в языке.

В плэнере различаются переменные двух видов — локальные и глобальные. Локальные переменные существуют лишь внутри некоторого участка программы (например, внутри процедуры). При обращениях к таким переменным используются префиксы «.», «*», «!.» и «!*». Глобальные же переменные, которые в языке называются *константами*, существуют с момента своего определения и до завершения программы, они доступны из любой точки программы. Обращаться к константам можно только с помощью префиксов «:» и «!:».

Атомарные выражения. Это общее название атомов и обращений к переменным. Любое атомарное выражение рассматривается в языке как неделимая конструкция. Поэтому, например, атомы A, B и C не имеют никакого отношения к атому ABC. Синтаксически неделимо и выражение .VAR, хотя по смыслу в нем и можно выделить префикс и имя переменной.

Списковые выражения, или просто *списки*,— это любые (возможно, пустые) последовательности элементов списка, заключенные в круглые, квадратные или угловые скобки. *Элементами списка* могут быть любые выражения языка — как атомарные, так и списковые. Количество элементов в списке называется *длиной* списка. Примеры списков:

(ABC 5.2 *20) [PRINT (A (B))] ⟨(A)⟩ ()

В первом примере указан список из трех элементов ABC, 5.2 и *20, во втором — список из двух элементов: идентификатора PRINT и списка (A (B)), в третьем — список из одного элемента (A), а в последнем примере — пустой список, не содержащий ни одного элемента. Длины указанных списков соответственно равны 3, 2, 1 и 0.

При записи списков их элементы, а также элементы и скобки могут быть отделены друг от друга любым числом пробелов (несколько пробелов подряд эквивалентны одному пробелу). Пробелы обязательны только между двумя элементами, являющимися атомарными выражениями, во всех остальных случаях пробелы необязательны.

Таким образом, записи

(2 [A B]) (2 [A B]) (2 [A B])

изображают один и тот же список. Первая форма записи наименее экономна, а вторая — наиболее компактна. Последняя же форма записи, в которой между любыми элементами списка ставится пробел, наиболее наглядна, ее-то мы и будем использовать в дальнейшем. В такой же форме списки всегда выводятся на печать.

Отметим также, что запись любого списка, согласно определению, должна быть сбалансирована относительно скобок, поэтому в плэнере записи типа

)A(((A) [A] ⟨C (B)⟩

недопустимы.

Списки различаются по виду скобок, которыми ограничены их элементы. Списки, составленные из одних и тех же элементов, но заключенные в разные скобки, считаются различными. Например, список (A) отличен от списка [A]. Для краткости мы будем называть списки с круглыми скобками *L-списками* (*L* — от названия языка lisp, в котором все списки принято заключать в такие скобки), списки с квадратными скобками — *P-списками* (*P* — от planner), а списки, заключенные в уголки,— *S-списками* (*S* — от слова segment).

Для читателей, знакомых с языком лисп, сообщим, что в плэнере никакой список, в том числе и ( ), не эквивалентен никакому атому, в том числе и атому NIL. Нет в плэнере и аналога лисповским точечным выражениям типа (A . B).

Выражения — это общее название атомарных и списковых выражений, и это единственный вид конструкций, которые используются в плэнере для изображения программ и данных.

Отметим, что для записи плэнерских программ необходимы все описанные выше типы выражений. В то же время для представления данных обычно достаточно лишь атомов и списков с круглыми скобками. Именно на такое представление данных ориентировано большинство процедур языка. Но если нужно, то для изображения данных можно использовать и другие типы выражений; все необходимые средства для их обработки язык предоставляет.

1.2. Формы

Основными компонентами плэнерских программ являются формы. *Форма* — это выражение, которое можно вычислить и получить в результате некоторое значение. Формы делятся на *простые* и *сегментные*. Форма называется простой, если ее значением является одно выражение, и сегментной, если ее значение — последовательность выражений (*сегмент*). Тот факт, что форма f имеет значение v, будем записывать в виде $f \rightarrow v$.

В языке имеется пять типов простых форм и три типа сегментных. Ниже перечислены все эти типы и указаны правила их вычисления.

Простые формы. 1. *Атомы.* Значением атома всегда является сам этот атом. Поэтому

```
ABC  →  ABC
49.6 →  49.6
*20  →  *20
```

2. *Обращения к переменным с префиксом «.» (.-переменные).* Значением такой формы является текущее значение переменной, имя которой указано в обращении. Если, к примеру, переменная X имеет значение **B, то**

```
.X  →  B
```

Может оказаться, что в момент обращения к переменной она не описана или не имеет значение, тогда вычислить данную форму нельзя.

3. *Обращение к константам с префиксом «:»* (*:-переменные*). Результатом обращения к константе (глобальной переменной) является ее текущее значение. Например, если в программе определена константа с именем CON и значением [A B], то

:CON → [A B]

Обращение к неопределенной константе является ошибкой.

4. *Списки в круглых скобках* (*L-списки*). *L*-список является формой, только если все его элементы — формы. Значением его является L-список, составленный из значений его элементов. Так, при указанных выше значениях переменной X и константы CON имеем

(A .X :CON) → (A B [A B])

Отметим, что поскольку значениями атомов являются сами атомы, значением L-списка, составленного из атомов, является сам же этот L-список. Например:

(A 45 —2.7 *650) → (A 45 —2.7 *650)

Более того, совпадают со своими значениями и L-списки, элементами которых являются не только атомы, но и L-списки, составленные из подобных же элементов. Например:

(A (B C)) → (A (B C))
((1 2) (3 (4 5)) 6) → ((1 2) (3 (4 5)) 6)

5. *Списки в квадратных скобках* (*P-списки*). Такой список является формой, только если он представляет собой обращение к функции. Значением данной формы является результат вычисления функции при заданных в обращении аргументах. Если, например, функция с именем + складывает числа, то

[+ 2 3 4] → 9

Прежде чем перейти к сегментным формам языка, определим понятие *сегментации выражения*. Под этим понимается следующее действие: сегментация атомарного выражения дает это же самое выражение, а сегментация списка дает последовательность (сегмент) элементов этого списка, т. е. при сегментации отбрасываются внешние скобки списка. Так, результатом сегментации списка (A ⟨B⟩ C) является сегмент A ⟨B⟩ C.

Сегментные формы. 1. *Обращения к переменным с префиксом* «!.» (*!.-переменные*). Значением такой формы является сегментированное значение указанной переменной. Например, при значении A у переменной X и значении [A (B C)] у переменной Y

имеем

!.X → A
!.Y → A (B C)

2. *Обращения к константам с префиксом* «!:» (!:-*переменные*). Результатом вычисления этой формы является сегментированное значение указанной константы.

3. *Списки в угловых скобках* (*S-списки*). Такое выражение является формой, только если оно представляет собой так называемое сегментное обращение к функции. Значением данной формы является сегментированный результат вычисления функции. Например, если значением функции с именем QUOTE является ее аргумент, то

⟨QUOTE (A [B] ⟨C D⟩)⟩ → A [B] ⟨C D⟩

Никакие другие выражения языка не являются формами, попытка вычислить их приведет к ошибке.

Отметим, что сегментные формы можно использовать только в небольшом числе случаев, которые всегда будут оговариваться явно. В дальнейшем же под «формой» будем понимать «простую форму», если не указано иное.

1.3. Обращения к функциям

Функция — это процедура, в результате вычисления которой вырабатывается некоторое значение. В виде таких процедур в плэнере представляются арифметические и другие операции (типа проверки на равенство), собственно функции (типа синуса) и операторы (типа оператора присваивания, цикла). Пользователь может вычислять функцию лишь ради ее побочного эффекта, но и в этом случае она должна выдавать некоторое значение.

При обращении к функции указывается ее имя и аргументы (фактические параметры). Имя функции — всегда идентификатор. Аргументами в общем случае могут быть любые выражения, но очень часто на них накладываются определенные ограничения. Значение функции — это некоторое выражение.

В языке различаются два вида обращений к функциям — простые и сегментные. *Простое обращение к функции* с именем *fn* и аргументами a_i записывается в виде следующего P-списка:

$[fn\ a_1\ a_2\ \ldots\ a_k]$

Этому обращению в обычной символике соответствует запись $fn(a_1, a_2, \ldots, a_k)$. *Сегментное обращение к функции* с тем же именем и теми же аргументами задается в виде S-списка:

$\langle fn\ a_1\ a_2\ \ldots\ a_k \rangle$

Примеры простых (слева) и сегментных (справа) обращений к функциям:

```
[SIN .X]              ⟨SIN .X⟩
[HEAD 2 (A B C)]      ⟨HEAD 2 (A B C)⟩
[READ]                ⟨READ⟩
```

(в последней строке указаны обращения к функции, не имеющей аргументов).

При любом, простом или сегментном, обращении к функции ее вычисление происходит одинаково, различие лишь в том, что при сегментном обращении полученное значение функции еще и сегментируется. Например, если функция REV «переворачивает» список (переписывает в обратном порядке его элементы), заданный в качестве аргумента, то имеем

```
[REV (A B C)]  →  (C B A)
⟨REV (A B C)⟩  →  C B A
```

При обращении к функции разрешается в качестве первого элемента списка-обращения указывать не только имя функции, но и .-переменную или :-переменную. В таком случае берется текущее значение указанной переменной (константы), которое должно быть идентификатором, и этот идентификатор рассматривается как имя функции, к которой происходит обращение. Например, обращение [.F 5] эквивалентно обращению [SIN 5], если текущим значением переменной F является идентификатор SIN, либо эквивалентно обращению [COS 5], если переменная F имеет значение COS. Такое неявное задание имени функции позволяет в ряде случаев сократить текст программы.

Все функции языка делятся на *встроенные* и *определяемые*. Встроенные функции — это стандартные функции, правила вычисления которых известны транслятору языка. Описывать в программе такие функции, естественно, не надо. В плэнере имеется большой набор встроенных функций, в виде которых представлены базовые операции по обработке данных и основные способы объединения операций в более крупные действия. Но как бы ни был велик набор подобных функций, в каждой программе возникает потребность в новых функциях, специфичных для данной программы. Такие функции должны быть описаны самим пользователем, только после этого он имеет право обращаться к ним. Именно эти функции и называются определяемыми.

Правила определения новых функций будут объяснены позже, а пока мы рассмотрим встроенные функции языка.

При описании встроенных функций будут использоваться следующие соглашения и обозначения.

1. При описании каждой встроенной функции будет указываться только простое обращение к ней и будут объясняться правила вычисления функции именно в этом случае.

2. Следуя традиции языка лисп, мы будем делить встроенные функции на два класса — класс SUBR и класс FSUBR. К классу SUBR будем относить те функции, которые работают со значениями своих аргументов; все аргументы этих функций (в терминологии языка алгол-60) вызываются по значению. Ясно, что аргументами функций класса SUBR могут быть не любые выражения, а только формы. Все остальные встроенные функции относятся к классу FSUBR; у них либо вообще не вычисляются аргументы, либо вычисляются, но уже во время выполнения собственно функции.

Примером функции класса FSUBR может служить встроенная функция QUOTE, обращение к которой имеет следующий вид:

[QUOTE x]

где x — любое выражение языка. Эта функция не вычисляет свой аргумент, а берет его в том виде, как он задан в обращении, и выдает в качестве своего значения. Например:

[QUOTE [+ 7 5]] → [+ 7 5]

Функция QUOTE применяется в тех случаях, когда выражение x должно рассматриваться как таковое, а не как форма, которая должна быть вычислена.

При описании каждой встроенной функции (имеющей хотя бы один аргумент) мы будем указывать, к какому классу она относится. Если указан класс SUBR, то это без дополнительных пояснений будет означать, что вычисление собственно функции начинается только после того, как будут вычислены все ее аргументы (аргументы всегда вычисляются слева направо). Для функций же класса FSUBR правила вычисления их аргументов будут всегда указываться явно.

3. На аргументы многих встроенных функций накладываются определенные ограничения. При описании функций мы будем указывать эти ограничения не словами, а используя следующие обозначения для аргументов:

e — любая простая форма с произвольным значением, которое будет обозначаться E;

l — простая форма, значением которой обязательно должен быть список (с любыми скобками), обозначаемый L;

i — простая форма, значением которой должен быть идентификатор, который будет обозначаться I;

n — простая форма, значением которой должно быть число (целое или вещественное), обозначаемое N;

s — простая форма, значением которой должна быть шкала, которая будет обозначаться S.

Эти обозначения могут быть снабжены индексами e_2, n_k, E_2, N_k и т. п.

Согласно данным обозначениям запись

[ELEM n l], SUBR

означает, что функция с именем ELEM относится к классу SUBR и что у нее два аргумента. Значением первого из них должно быть число (которое мы обозначаем N), а значением второго — список (L). Любые другие значения аргументов недопустимы и ведут к ошибке.

4. Как указано выше, значениями аргументов, обозначенных символом n, могут быть как целые, так и вещественные числа. Однако иногда значение такого аргумента по смыслу обязано быть целым числом. Например, указанная выше функция ELEM выделяет N-й элемент из списка L; ясно, что число N должно быть целым. В подобных ситуациях, если в результате вычисления аргумента получилось все же вещественное число, то оно автоматически округляется до ближайшего целого числа *), которое уже и используется функцией. Так, при вычислении [ELEM 2.6 (A B C)] число 2.6 будет преобразовано в целое 3, поэтому из списка (A B C) выделяется третий элемент. В дальнейшем подобные преобразования будут подразумеваться неявно.

5. При обращении к некоторым встроенным функциям можно задавать разное количество аргументов: какие-то из аргументов можно опускать. Обозначения таких факультативных аргументов мы будем снабжать вопросительным знаком. Например, запись

[F n i? l?]

означает, что допустимо любое из следующих обращений: [F n], [F n i], [F n l], [F n i l]**). Порядок присутствующих аргументов должен соответствовать порядку, указанному в описании функции, в связи с чем обращение [F n l i] недопустимо.

В тех же случаях, когда при обращении к функции можно задавать любое количество однотипных аргументов, но не менее p, мы будем использовать запись вида

[F e_1 e_2 ... e_k], $k \geqslant p$.

---

*) Под «округлением числа x до ближайшего целого» понимается взятие целой части числа $x + 0.5$.

**) Отметим, что в плэнере нет встроенных функций типа [F n? n?], у которых отсутствие факультативных аргументов вызывало бы затруднение в определении того, какие из аргументов присутствуют, а какие — нет.

1.4. Операции над списками

Списки являются наиболее важным типом выражений языка, и первыми будут рассмотрены встроенные функции, определяющие основные операции над списками. К таким операциям относятся: выделение отдельных частей списка, удаление элементов из списка, построение новых списков.

Функция ELEM: [ELEM n l], SUBR.

Значением данной функции является $|N|$-й элемент списка L, отсчитанный от начала списка, если $N > 0$, или от конца списка, если $N < 0$. Если число N равно нулю или по абсолютной величине больше длины списка L, то значение функции не определено. Примеры:

```
[ELEM 2 (A B C D)]   → B
[ELEM —1 (A B C D)]  →  D
[ELEM 5 (A B C D)]   — ошибка
[ELEM [ELEM 1 (—3 —2)]
      [QUOTE ⟨(A) (B) (C) (D)⟩]]  →  (B)
```

Поскольку функция ELEM относится к классу SUBR, то во всех этих примерах прежде всего вычисляются аргументы. Но если в первых трех примерах таких вычислений фактически нет, так как аргументы совпадают со своими значениями, то в последнем примере аргументы действительно вычисляются. Вычисление в этом примере происходит так: сначала вычисляется первый аргумент:

```
[ELEM 1 (—3 —2)]  →  —3
```

далее вычисляется второй аргумент:

```
[QUOTE ⟨(A) (B) (C) (D)⟩]  →  ⟨(A) (B) (C) (D)⟩
```

и лишь затем начинает работать собственно функция ELEM, которая из списка ⟨(A) (B) (C) (D)⟩ выбирает третий элемент от конца, т. е. элемент (B), и выдает его в качестве своего значения.

Отметим, что для плэнера характерна вложенность одних обращений к функциям в другие обращения. И поскольку до вычисления собственно функции должны быть вычислены ее аргументы, то в первую очередь всегда вычисляются внутренние обращения и лишь затем вычисляются внешние обращения.

Функция ELEM — одна из наиболее часто используемых функций языка. При этом номер выделяемого элемента обычно либо известен заранее, либо является значением какой-то переменной или константы. Для сокращения записи в подобных ситуациях разрешается применять упрощенную форму обращения к функции ELEM: если первый аргумент является числом, или переменной,

или :-переменной, то при обращении к функции можно не указывать ее имя. Например, если значением переменной I является число 2, а значением константы K — число —3, то имеем

[.I (1 2 3 4 5)] → 2
[1 [:K ((A B) C D)]] → A

При любом другом первом аргументе использовать сокращенный вид обращения к функции ELEM нельзя. Например, запись [[1 .L] (A B C)] ошибочна.

Функция ELEM выделяет элементы верхнего уровня списка. Чтобы выделить подвыражения с более глубоких уровней, надо применить композицию функций ELEM. Однако более удобной в таком случае является

Функция INDEX: [INDEX l n_1 n_2 ... n_k], SUBR, $k \geqslant 1$.

Действие этой функции эквивалентно вычислению выражения

[ELEM n_k [... [ELEM n_2 [ELEM n_1 l]] ...]]

Следовательно, функция INDEX выделяет из списка L его $|N_1|$-й элемент (отсчитанный от начала или конца списка в зависимости от знака числа N_1), затем из этого элемента, который должен быть списком, выделяет $|N_2|$-й элемент, из которого выделяет $|N_3|$-й элемент и т. д. Например:

[INDEX ((1 2 3) (4 5) (6)) 2 —1] → 5
[INDEX (AB BA) 1 2] — ошибка

Во втором из этих примеров предпринята попытка выделить элемент не из списка, а из атома, что и привело к ошибке.

Функция REST: [REST n l], SUBR.

Значением этой функции является список, полученный из списка L отбрасыванием N первых элементов, если $N \geqslant 0$, или $|N|$ последних элементов, если $N < 0$. Если число N по абсолютной величине превосходит длину списка L, то значение функции не определено. Примеры:

[REST 1 (A B C D)] → (B C D)
[REST —2 [QUOTE ⟨A B C D⟩]] → ⟨A B⟩
[REST 0 (A B)] → (A B)

Читатели, знакомые с языком лисп, легко догадаются, что лисповскому выражению (CAR X) соответствует плэнерская форма [1 .X], а (CDR X) — форма [REST 1 .X]. Плэнерские функции ELEM и REST обобщают действия лисповских функций CAR и CDR, они позволяют выделять или отбрасывать из списка любые элементы с любого его конца.

Функция HEAD: [HEAD n l], SUBR.

Данная функция противоположна функции REST: те элементы списка, которые функция REST отбрасывает, функция HEAD как раз оставляет. Точнее, значением функции HEAD является список, составленный из N первых элементов списка L, если $N \geqslant 0$, или из $|N|$ последних элементов, если $N < 0$. Значение функции не определено, если $|N|$ больше длины списка L. Примеры:

[HEAD 1 (A B C D)] → (A)
[HEAD —2 [QUOTE ⟨A B C D⟩]] → ⟨C D⟩
[HEAD 0 (A B)] → ()

Отметим, что функция HEAD, как и функция REST, сохраняет скобки исходного списка.

С помощью этих функций можно выделить из списка любую группу соседних элементов. Например, выделить с третьего по пятый элементы списка, являющегося значением переменной X, можно вычислением выражения

[REST 2 [HEAD 5 .X]]

Действительно, сначала вычисляется выражение [HEAD 5 .X], которое в качестве своего значения выдает список из пяти первых элементов списка X, а затем из этого списка функция REST отбрасывает два первых элемента. В результате получается список, составленный из третьего, четвертого и пятого элементов исходного списка. (Отметим, что фразу «отбросить элементы списка» не следует понимать как изменение списка. Функции REST и HEAD не портят исходный список, а создают новый список.)

Все описанные выше функции выделяли фрагменты списков. Рассмотрим теперь обратную операцию — построение новых списков из имеющихся частей. В отличие от языка лисп, где для этого используется несколько встроенных функций (CONS, LIST, APPEND), в плэнере построение любых новых функций осуществляется с помощью единственной, но более мощной функции FORM.

Функция FORM: $\left[\text{FORM } \{e'_1\ e'_2\ \ldots\ e'_k\}\right]$, FSUBR, $k \geqslant 0$.

Здесь под фигурными скобками понимаются любые плэнерские скобки (круглые, квадратные или угловые), а под e'_i — любые формы, простые или сегментные. Функция FORM по очереди (слева направо) вычисляет формы e'_i и из их значений строит список с теми же скобками, что и у исходного списка. Полученный таким образом список объявляется значением функции.

Если через E'_i обозначить значение формы e'_i, то действие функции FORM можно изобразить так:

$$\left[\text{FORM } \{e'_1\ e'_2\ \ldots\ e'_k\}\right] \rightarrow \{E'_1\ E'_2\ \ldots\ E'_k\}$$

Пусть, например, значением переменной X является число 1, а значением переменной Y — список (A B C). Тогда

[FORM [REST .X .Y]] → [REST 1 (A B C)]
[FORM ⟨5 !.Y .Y⟩] → ⟨5 A B C (A B C)⟩

Отметим, что в первом из этих примеров значением функции FORM является именно список [REST 1 (A B C)], а не значение этого списка, рассматриваемого как обращение к функции REST. Во втором примере элемент !.Y исходного списка «поставляет» в список-результат сразу три элемента A, B и C, так как значением сегментного обращения к переменной Y является последовательность из этих трех элементов. В то же время выражение .Y имеет своим значением список (A B C), который и попадает в таком виде в список-результат.

Функцию FORM можно применять и для построения списков с круглыми скобками, например:

[FORM ([2 .Y] .X)] → (B 1)

однако в этом случае обращение к функции FORM излишне, поскольку, как было указано в § 1.2, верно следующее:

$$(e'_1\ e'_2\ \ldots\ e'_k) \rightarrow (E'_1\ E'_2\ \ldots\ E'_k)$$

Следовательно, для того чтобы построить новый список с круглыми скобками, достаточно выписать подряд выражения, из значений которых должен состоять новый список, заключить все эти выражения в круглые скобки и полученный таким образом список вычислить.

Данное правило вычисления списков включает в себя различные конкретные способы построения списков, которые мы проиллюстрируем на примерах. Во всех этих примерах предполагается, что переменная X имеет значение [A B], переменная Y — значение (C D E), переменная Z — значение F, а переменная W — значение ( ).

Если мы хотим построить L-список, элементами которого являются значения каких-то переменных и функций (в лиспе в этих целях применяется функция LIST), то должны использовать простые обращения к данным переменным и функциям:

(.X .Y) → ([A B] (C D E))
(.X .Y .Z .W) → ([A B] (C D E) F ( ))
(A (.Z C) .W) → (A (F C) ( ))
([REST 1 .Y] [QUOTE ⟨ ⟩]) → ((D E) ⟨ ⟩)

Если элементами нового списка должны быть не сами значения переменных и функций, а элементы этих значений (это аналог лисповской функции APPEND), тогда следует использовать

сегментные обращения к переменным и функциям:

(!.X !.Y) → (A B C D E)
(!.X !.Y !.Z !.W) → (A B C D E F)
(⟨REST 1 .Y⟩ !.X) → (D E A B)

Во втором из этих примеров значением выражения !.Z является атом F, который и заносится в список-результат, а значением выражения !.W является пустой сегмент, поэтому данное выражение не вносит никаких элементов в список-результат.

Если требуется добавить новые элементы в начало существующего списка (аналог лисповской функции CONS), в середину или его конец, если требуется объединить элементы некоторых списков с добавлением новых элементов, тогда используются как простые, так и сегментные обращения к переменным и функциям:

(.X !.Y) → ([A B] C D E)
(!.Y .X) → (C D E [A B])
(.X !.Y .Z) → ([A B] C D E F)
(!.X .W !.Y) → (A B ( ) C D E)

Рассмотрим еще два примера. Если значением переменной L является список (1 2 3 4 5), тогда

(⟨HEAD 2 .L⟩ ⟨REST 3 .L⟩) → (1 2 4 5)
(⟨REST 1 .L⟩ [1 .L]) → (2 3 4 5 1)

В первом примере из списка удаляется третий элемент, а во втором примере первый элемент переносится в конец списка.

Функция LENGTH: [LENGTH *l*], SUBR.

Значением этой функции является целое число — длина списка *L*:

[LENGTH ( )] → 0
[LENGTH [QUOTE ⟨A (B C) D⟩]] → 3

1.5. Арифметические функции

Хотя язык плэнер и не ориентирован на вычислительные задачи, в нем все же имеется достаточно большой набор встроенных арифметических функций. Они рассматриваются в данном параграфе. Значениями всех аргументов этих функций должны быть числа, целые или вещественные. Значения функций — также числа.

Функция +: [+ n_1 n_2 ... n_k], SUBR, $k \geqslant 1$.

Значением функции является сумма значений ее аргументов. Это значение будет целым числом, только если значения всех аргументов — целые числа; если значение хотя бы одного аргумента вещественно, то вещественным будет и значение функции.

Примеры:

[+ 4 −3 0] → 1
[+ 4 −3 0.0] → 1.0

Функция −: [− n_1 n_2], SUBR.

Эта функция вычисляет разность $N_1 - N_2$, где N_1, напомним, обозначает значение аргумента n_1, а N_2 — значение аргумента n_2. Результат вычисления функции будет целым числом, только если числа N_1 и N_2 целые. Пример:

[− [+ 3 6] [− 4 2]] → 7

Отметим, что этому выражению в обычной математической символике соответствует запись $(3 + 6) - (4 - 2)$. Как видно, плэнерская символика менее наглядна и удобна. Однако в плэнерских программах выписывать сложные арифметические выражения приходится довольно редко, поэтому данный недостаток не сказывается слишком сильно.

Функция ×: [× n_1 n_2 ... n_k], SUBR, $k \geqslant 1$.

Значением функции является произведение чисел N_i. Оно будет целым числом лишь при условии, что все N_i — целые числа.

Функция /: [/ n_1 n_2], SUBR.

Эта функция делит число N_1 на число N_2, которое не должно равняться нулю. Значением данной функции всегда является вещественное число. Например:

[/ 20 4] → 5.0

Функция DIV: [DIV n_1 n_2], SUBR.

Значение функции — целое число $[N_1/N_2]$, где $[x]$ означает наибольшее целое, не превосходящее число x. Если N_1 и N_2 — натуральные числа, то эта функция находит неполное частное от деления N_1 на N_2. Примеры:

[DIV 1983 100] → 19
[DIV 5.7 −3.4] → −2

Функция MOD: [MOD n_1 n_2], SUBR.

Значением функции является число $N_1 - N_2 \times [N_1/N_2]$. Оно будет целым, только если N_1 и N_2 — целые числа. Для натуральных чисел эта функция вычисляет остаток от деления N_1 на N_2. Например:

[MOD 1983 100] → 83
[MOD 5.7 −3.4] → −1.1

Функция ↑: [↑ n_1 n_2], SUBR.

Функция возводит N_1 в степень N_2. Значение функции является целым числом только при условии, что N_1 — целое, а N_2 — натуральное число или 0. При вещественном N_2 и неположительном N_1, а также при нулевом N_1 и отрицательном целом N_2 значение функции не определено. Примеры:

[↑ 5 2] → 25
[↑ 5.0 0] → 1.0

Функция ABS: [ABS n], SUBR.

Значением функции является абсолютная величина числа N. Тип результата совпадает с типом числа N.

Функция ENTIER: [ENTIER n], SUBR.

Значение функции — наибольшее целое, не превосходящее число N. Например:

[ENTIER 5.8] → 5

Функция ROUND: [ROUND n], SUBR.

Значением функции является ближайшее к N целое число, т. е. наибольшее из целых, не превосходящих число $N + 0.5$. Например:

[ROUND 5.8] → 6

Функция SIGN: [SIGN n], SUBR.

В результате вычисления этой функции получается число 1, если $N > 0$, или число 0, если $N = 0$, или число —1, если $N < 0$.

Функция MAX: [MAX n_1 n_2 ... n_k], SUBR, $k \geqslant 1$.

Значение функции равно наибольшему из чисел N_i (первому из наибольших, если их несколько). Например:

[MAX 4 6.0 —2.9 6] → 6.0

Функция MIN: [MIN n_1 n_2 ... n_k], SUBR, $k \geqslant 1$.

Значение функции равно (первому) наименьшему из чисел N_i.

Функция RANDOM: [RANDOM].

При каждом обращении эта функция в качестве своего значения выдает некоторое «случайное» число из интервала (0, 1). Эти псевдослучайные числа имеют равномерное распределение.

Кроме указанных выше арифметических функций, в плэнере имеются также встроенные функции, соответствующие элементарным математическим функциям. Считая известными их определения, перечислим лишь имена этих функций в языке: SIN (синус), COS (косинус), TG (тангенс), CTG (котангенс), ARCSIN (арксинус), ARCTG (арктангенс), SQRT (квадратный корень), EXP (экспонента) и LN (натуральный логарифм). Все эти функции имеют

один аргумент и относятся к классу SUBR, их значениями всегда являются вещественные числа. Примеры:

```
[SQRT 9]  →  3.0
[SIN [ARCSIN 0.5]]  →  0.5
```

1.6. Операции над шкалами

Шкалы можно использовать для представления разных типов данных. Например, если отождествить биты 1 и 0 с логическими значениями «истина» и «ложь», то шкалу можно рассматривать как вектор логических величин. В виде шкал можно представлять и подмножества некоторого фиксированного множества. Для этого следует поставить в соответствие каждому элементу множества (например, каждой латинской букве) определенный разряд шкалы (букве A — самый правый разряд, букве B — предпоследний разряд и т. д.), тогда любое подмножество этого множества будет изображаться шкалой, в которой единицы находятся только в тех разрядах, что соответствуют элементам подмножества. Так, подмножество гласных латинских букв {A, E, I, O, U} изображается шкалой *04040421.

Отметим, что логические массивы и множества можно представлять и в виде списков, например множество гласных букв — в виде списка (A E I O U), но представление в виде шкал более экономно в смысле расхода памяти. Правда, шкалы имеют ограниченный размер, поэтому с их помощью можно представлять массивы и множества только с небольшим числом элементов.

Для работы со шкалами в плэнер введены следующие функции.

Функция $\vee$: [$\vee$ s_1 s_2], SUBR.

Напомним, что согласно принятым в § 1.3 обозначениям, значениями обоих аргументов должны быть шкалы, которые мы обозначаем как S_1 и S_2. Значение функции — также шкала, i-й разряд которой равен 1, если в i-м разряде хотя бы одной из шкал S_1 и S_2 имеется единица, и равен 0 иначе. Например:

```
[∨ *14 *12]  →  *16
```

Если шкалы рассматривать как логические массивы, то данная функция реализует поразрядное логическое сложение (дизъюнкцию), а если рассматривать их как множества, то это операция объединения множеств.

Функция $\wedge$: [$\wedge$ s_1 s_2], SUBR.

Значением этой функции является шкала, содержащая единицы только в тех разрядах, в которых обе шкалы S_1 и S_2 одновремен-

но имеют единицы. Например:

[∧ *14 *12] → *10

Данная функция реализует операцию поразрядного логического умножения (конъюнкцию), если шкалы рассматриваются как логические массивы, или операцию пересечения множеств, если шкалы изображают множества.

Функция COMP: [COMP s_1 s_2], SUBR.

Значение функции — шкала, значение i-го разряда которой равно 0, если значения i-х разрядов шкал S_1 и S_2 совпадают, и равно 1 в противном случае. Отметим, что если шкалы S_1 и S_2 полностью совпадают, то значением функции является нулевая шкала *0. Например:

[COMP *14 *12] → *6
[COMP *7042 *7042] → *0

Для логических массивов данная функция реализует операцию поразрядного сравнения (отрицания логической эквивалентности), а для множеств — симметрической разности.

Функция SHIFT: [SHIFT s n], SUBR.

Значением второго аргумента должно быть целое число N (если значением является вещественное число, то оно автоматически округляется до ближайшего целого — см. § 1.3). Значением данной функции является шкала, полученная сдвигом шкалы S на $|N|$ разрядов вправо, если $N \geqslant 0$, или влево, если $N < 0$. Разряды, вышедшие при сдвиге за пределы шкалы, теряются. Например:

[SHIFT *5071 3] → *507
[SHIFT *5071 —1] → *12162

Функция BSUM: [BSUM s], SUBR.

Значением этой функции является целое число — количество единиц в двоичном представлении шкалы S. Например:

[BSUM *04040421] → 5

С помощью данной функции можно, например, определить количество элементов, входящих в множество, изображаемое шкалой S.

Функция TOPBIT: [TOPBIT s], SUBR.

Если считать, что разряды шкалы S перенумерованы справа налево, начиная с 1, то эта функция в качестве своего значения выдает номер самой левой единицы данной шкалы или число 0, если в шкале нет единиц. Пример:

[TOPBIT *370] → 8

1.7. Предикаты

В данном параграфе рассматриваются встроенные функции, с помощью которых можно проверять условия, свойства. Эти функции вырабатывают значение «истина», если проверяемое условие выполнено, и значение «ложь» в противном случае.

Предикатом будем называть любую форму, в результате вычисления которой вырабатываются логические значения «истина» и «ложь». Значение «ложь» в языке всегда представляется пустым списком ( ). Единого же обозначения для «истины» в языке нет: любое выражение, отличное от ( ), воспринимается как обозначение «истины». Наиболее часто в роли такого обозначения выступает атом T (от английского true — истина). Например, большинство встроенных функций-предикатов, когда они принимают значение «истина», выдают в качестве своего значения именно этот атом.

Возникает вопрос: как отличать функции-предикаты от других функций, раз те и другие принимают любые значения? Ответ таков: формально они ничем не отличаются друг от друга, поэтому любую функцию (в общем случае — любую форму) можно использовать как предикат. Просто те функции, которые проверяют какие-то условия, принято называть предикатами, а их значения трактовать как логические значения. Если же функция выполняет какие-либо иные действия, то ее значениям не приписывается такой смысл, и она не рассматривается как предикат. Трактовать функцию как предикат или нет — решает пользователь.

Простейшими примерами функций-предикатов являются встроенные функции, проверяющие типы выражений. С них мы и начнем.

Функция ID: [ID *e*], SUBR.

Если *E* (значение аргумента) является идентификатором, то значение этой функции равно T («истина»). При других значениях аргумента функция принимает значение ( ) («ложь»). Например:

```
[ID X]          →  T
[ID [QUOTE X]]  →  ( )
```

Функция NUM: [NUM *e*], SUBR.

Если *E* — число, целое или вещественное, то функция имеет значение T, в противном случае — значение ( ). Например:

```
[NUM —6]          →  T
[NUM [1 (5.9 A)]] →  T
[NUM *6]          →  ( )
```

Функция INT: [INT *e*], SUBR.

Значение этого предиката равно T, если значение аргумента явля-

ется целым числом, и равно ( ) иначе. Например:

[INT 5] → T
[INT 5.] → ( )

Функция REAL: [REAL *e*], SUBR.

Если *E* — вещественное число, то функция принимает значение T, иначе — значение ( ). Например:

[REAL 5.] → T
[REAL [LENGTH (A B)]] → ( )

Функция SCALE: [SCALE *e*], SUBR.

Если *E* — шкала, то результат вычисления данной функции — атом T, в противном случае — пустой список ( ). Например:

[SCALE [COMP *20 *451]] → T

Функция ATOM: [ATOM *e*], SUBR.

Значение этого предиката равно T, если значение аргумента — атом, т. е. идентификатор, число или шкала, и равно ( ) при любых других значениях аргумента. Например:

[ATOM NIL] → T
[ATOM ( )] → ( )
[ATOM [QUOTE :A]] → ( )

Функция VAR: [VAR *e*], SUBR.

Если *E* — обращение к переменной (с любым префиксом), то значением функции является атом T, иначе — пустой список ( ). Например:

[VAR [QUOTE !*X]] → T
[VAR [QUOTE :CONST]] → T
[VAR CONST] → ( )

Функция ATOMIC: [ATOMIC *e*], SUBR.

Значение функции равно T, если E — любое атомарное выражение (атом или обращение к переменной), а иначе равно ( ).

Функция LIST: [LIST *e*], SUBR.

Если E является списком в круглых скобках, то функция принимает значение T, в противном случае — значение ( ). Например:

[LIST NIL] → ( )
[LIST ( )] → T
[LIST [QUOTE (A ⟨B⟩)]] → T
[LIST [QUOTE ⟨A (B)⟩]] → ( )

Функция LISTR: [LISTR *e*], SUBR.

Значение функции равно T, если E — любое списковое выражение,

и равно ( ), если E — атомарное выражение. Примеры:

[LISTR (A)] → T
[LISTR [QUOTE ⟨A (B)⟩]] → T
[LISTR NIL] → ( )

Функция EMPTY: [EMPTY *e*], SUBR.

Если E — пустой список (с любыми скобками), то функция имеет значение T, в противном случае — значение ( ). Например:

[EMPTY ( )] → T
[EMPTY [QUOTE ⟨ ⟩]] → T
[EMPTY (( ))] → ( )

Выше были перечислены наиболее часто используемые функции, проверяющие тип своего аргумента. В плэнере есть и другие подобные встроенные функции. Они также имеют один аргумент, также относятся к классу SUBR и также принимают значение T, если значение их аргумента имеет нужный тип, и значение ( ) в противном случае. Ниже перечислены имена всех этих функций и указаны проверяемые ими типы выражений:

LISTP — P-список, т. е. список в квадратных скобках,
LISTS — S-список, т. е. список в уголках,
VARP — простое обращение к переменной (с префиксами «.», «*» или «:»),
VAR. — .-переменная,
VAR* — *-переменная,
VAR: — :-переменная,
VARS — сегментное обращение к переменной (с префиксами «!.», «!*» или «!:»),
VAR!. — !.-переменная,
VAR!* — !*-переменная,
VAR!: — !:-переменная.

Следующие две встроенные функции позволяют проверять на равенство или неравенство любые выражения.

Функция EQ: [EQ e_1 e_2], SUBR.

Если значениями аргументов являются равные выражения, то функция принимает значение T, иначе — значение ( ). Примеры:

[EQ 5.0 5] → ( )
[EQ (A (B)) [QUOTE (A (B))]] → T
[EQ *0 0] → ( )

Функция NEQ: [NEQ e_1 e_2], SUBR.

Это проверка на неравенство: если значения аргументов не равны, то значение функции — атом T, равны — пустой список ( ). На-

пример:

[NEQ ( ) NIL] → T
[NEQ *0025 *25] → ( )
[NEQ (A B) [QUOTE [A B]]] → T

Если функции EQ и NEQ применимы к любым выражениям, то следующие четыре функции — арифметические предикаты — требуют, чтобы значениями их аргументов были числа. Эти функции относятся к классу SUBR и проверяют отношения «больше», «больше или равно», «меньше» и «меньше или равно»:

[GT n_1 n_2] — $N_1 > N_2$?
[GE n_1 n_2] — $N_1 \geqslant N_2$?
[LT n_1 n_2] — $N_1 < N_2$?
[LE n_1 n_2] — $N_1 \leqslant N_2$?

Примеры:

[GE 5.6 2.7] → T
[LT 1 1.0] → ( )

Следующая функция, проверяющая, входит ли одно выражение в другое,— пример предиката, который использует для обозначения «истины» не атом T, как это делают все выше перечисленные функции-предикаты, а другие выражения.

Функция MEMB: [MEMB *e l*], SUBR.

Значением первого аргумента может быть любое выражение *E*, а значением второго аргумента должен быть список *L* (с любыми скобками). Если *E* равно одному из элементов (верхнего уровня) списка *L*, то значением данной функции является целое число — номер (при отсчете слева направо) первого из элементов, совпадающих с *E*. Если *E* не равно никакому элементу списка *L*, то значением функции является «ложь», т. е. пустой список ( ). Таким образом, данная функция позволяет не только узнать, входит ли *E* в *L*, но и определить, на каком именно месте *E* входит в *L*. Примеры:

[MEMB B (A (B) C)] → ( )
[MEMB (B) [QUOTE ⟨A (B) C (B)⟩]] → 2

1.8. Логические функции

В предыдущем параграфе были описаны основные предикаты языка. Теперь же мы рассмотрим встроенные функции, в которых главным образом и используются предикаты. Эти функции определяют логические операции отрицания, дизъюнкции и конъюнкции, а также условные выражения.

Функция NOT: [NOT e], SUBR.

Это «отрицание». Если значение аргумента равно ( ), тогда значением функции является атом T, при любых других значениях аргумента функция принимает значение ( ). Примеры:

```
[NOT [ATOM ( )]]   →   T
[NOT [MEMB *2 (*0 *1 *2)]]   →   ( )
```

Функция OR: [OR e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Это «дизъюнкция»: функция принимает значение «истина», если хотя бы один из ее аргументов имеет значение «истина». Однако заранее аргументы функции не вычисляются, поэтому-то функция и отнесена к классу FSUBR. Точное определение функции таково. Она по очереди слева направо вычисляет свои аргументы. Если значения всех аргументов равны ( ), тогда и значение функции равно ( ). Но если нашелся аргумент, значение которого отлично от ( ), то функция с этим значением заканчивает свою работу, не вычисляя оставшиеся аргументы.

Пусть, например, значением переменной X является пустой список ( ), тогда имеем

```
[OR [ATOM .X] [EQ .X (A)]]   →   ( )
[OR .X [MEMB .X (A B ( ))] T]   →   3
[OR [ATOMIC .X] [EMPTY .X]
    [NUM [1 .X]]]   →   T
```

Отметим, что в последнем примере второй аргумент функции OR имел значение «истина», поэтому третий аргумент уже не вычислялся, так что ошибка в нем (выделение первого элемента из пустого списка) не проявилась.

Функция AND: [AND e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Это «конъюнкция»: функция принимает значение «истина», только если все ее аргументы имеют значение «истина». У этой функции аргументы также не вычисляются заранее, их вычисляет (по очереди слева направо) сама функция. Если значения всех аргументов отличны от ( ), тогда значение последнего из них становится значением функции. Но если нашелся аргумент со значением ( ), тогда функция, не вычисляя оставшиеся аргументы, прекращает свою работу со значением ( ). Примеры:

```
[AND [ATOM *5] [MEMB A (A B C)]]   →   1
[AND ( ) [REST 2 (A)]]   →   ( )
```

Функция COND. Обращение к этой функции называется *условным выражением* и имеет вид

$$\left[\text{COND}\ \left(p_1\ e_{11}\ e_{12}\ \ldots\ e_{1m_1}\right)\ \ldots\ \left(p_k\ e_{k1}\ e_{k2}\ \ldots\ e_{km_k}\right)\right],$$

где $k \geqslant 1$, $m_i \geqslant 0$. Аргументы данной функции, т. е. списки $(p_i\ e_{i1}\ e_{i2}\ \ldots\ e_{im_i})$, называются *клаузами*, а их первые элементы — простые формы p_i — *условиями*. Функция COND относится к классу FSUBR.

Функция по очереди вычисляет условия p_i. Если все они имеют значение «ложь», т. е. ( ), тогда и значение функции равно ( ). Но если нашлось условие p_i, значение которого отлично от ( ), тогда остальные условия уже не будут вычисляться, и функция рассматривает только данную i-ю клаузу: она по очереди вычисляет выражения e_{i1}, e_{i2} и т. д. Значение последнего выражения клаузы (им может быть и p_i, если $m_i = 0$) объявляется результатом вычисления всего условного выражения. Например:

```
[COND (( ) A) (( ) B)]  →  ( )
[COND (( ) A) (T B C)]  →  C
[COND (A) (T B)]        →  A
```

Мы рассмотрели обращение к функции COND в самом общем виде. Обычно же обращение к ней имеет более простой вид:

[COND $(p_1\ e_1)$ $(p_2\ e_2)$ ... $(p_k\ e_k)$]

Назначение такого условного выражения — выбрать одно из выражений e_i, вычислить его и взять его значение в качестве значения всего условного выражения. При этом выбирается выражение из первой клаузы с истинным условием.

Предположим, к примеру, что требуется вычислить сумму значений переменных X и Y, если эти значения являются числами, требуется построить список вида $(x + y)$, если хотя бы одно из значений x и y этих переменных является идентификатором, и требуется получить пустой список ( ) во всех остальных случаях. Сделать это можно с помощью такого условного выражения:

```
[COND ([AND [NUM .X] [NUM .Y]] [+ .X .Y])
      ([OR [ID .X] [ID .Y]] (.X + .Y))
      (T ( ))]
```

Отметим, что последнюю клаузу — (T ( )) — можно было бы и опустить, поскольку функция COND, согласно определению, и так выдаст пустой список ( ) в качестве своего значения, если два первых условия окажутся ложными.

1.9. Функция PROG

В языке имеется несколько встроенных функций, вычисление которых напоминает выполнение блоков в языке алгол-60. Такие функции мы будем называть *блочными функциями*, или просто *блоками*. Одна из них, а также сопутствующие ей функции рассматриваются в этом параграфе.

Функция PROG. Обращение к этой блочной функции имеет следующий вид:

[PROG (v_1 v_2 ... v_m) e_1 e_2 ... e_k], FSUBR, $m \geqslant 0$, $k \geqslant 1$.

Первый аргумент — список (v_1 v_2 ... v_m) — называется *описанием локальных переменных* блока, а последовательность остальных аргументов — e_1 e_2 ... e_k — называется *телом* блока. При этом простую форму e_i будем называть *меткой*, если она является идентификатором, и *оператором* в остальных случаях.

Вычисление функции PROG начинается с обработки описания локальных переменных, в результате чего вводятся в употребление переменные, областью существования которых является тело данного блока. Каждый элемент v_i описания вводит одну локальную переменную. Если v_i — идентификатор, тогда это — имя переменной, которой не присваивается никакого начального значения. Иначе v_i должен быть L-списком из двух элементов: идентификатора и любого выражения. Идентификатор определяет имя локальной переменной, а выражение (оно не вычисляется) — ее начальное значение. Например, при вычислении блока

[PROG (X (Y A) Z (W5 ⟨2 .Y⟩)) ...]

будут введены четыре локальные переменные с именами X, Y, Z и W5, причем X и Z не получат начальных значений, Y получит начальное значение A, а W5 — значение ⟨2 .Y⟩.

После введения локальных переменных функция PROG переходит к вычислению своего тела: по очереди вычисляются операторы e_i. Когда будет вычислен последний оператор e_k, функция закончит свою работу со значением, равным значению оператора e_k. При выходе из функции ее локальные переменные уничтожаются.

Отметим, что вычисленное значение любого оператора e_i, кроме последнего, игнорируется. Поэтому, хотя e_i могут быть произвольными простыми формами, в качестве e_i имеет смысл использовать только идентификаторы, играющие роль меток, и обращения к функциям, которые имеют побочные эффекты. К таким функциям относятся, например, встроенные функции SET, GO, RETURN, DO и COND.

Функция SET: [SET i e], SUBR.

Если значением второго аргумента может быть любое выражение, то значением первого аргумента должен быть идентификатор I, являющийся именем одной из существующей в данный момент переменной, например переменной ближайшего блока или какого-нибудь другого объемлющего блока. Значением функции SET яв-

ляется выражение *E*, однако главное в этой функции — ее побочный эффект: переменной с именем *I* присваивается новое значение *E*.

Например, после вычисления

[SET X [REST 1 (A B C)]] → (B C)

переменная X получит значение (B C), а после вычисления

[SET [2 (X Y Z W)] *5]

переменной Y будет присвоено значение *5. Если в первом из этих примеров имя переменной — X — задано явно, то во втором примере имя переменной — Y — было получено в результате вычисления. Таким образом, имя переменной, которой присваивается значение, может быть заранее и неизвестно. Но, отметим, в любом случае значением первого аргумента должно быть просто имя переменной, без каких-либо префиксов.

Функция SET, как правило, используется лишь ради своего побочного эффекта. Но иногда используется и вырабатываемое ею значение. Например, присвоить переменным X и Y одно и то же значение — скажем, текущее значение переменной Z — можно так:

[SET X [SET Y .Z]].

Функция SET относится к классу SUBR, поэтому вычисление внешней функции SET начинается с вычисления ее аргументов. Вычисление формы X дает этот же идентификатор X, а вычисление аргумента [SET Y .Z] дает значение переменной Z, которое внешняя функция SET и присваивает переменной X. Кроме того, при вычислении внутренней функции SET также был побочный эффект: переменная Y сменила свое значение на значение переменной Z. Таким образом, сначала Y, а затем X получили одинаковое значение. Значением всего указанного выражения является значение переменной Z, но оно уже не используется.

Функция GO: [GO *i*], SUBR.

Эта функция осуществляет переход по метке *I*. Как уже было сказано, операторы блока выполняются в том порядке, как они записаны. Функция же GO позволяет изменить этот порядок выполнения: она передает управление на оператор, следующий за меткой *I*. Например, в блоке

```
[PROG (X)
    [GO B] A [SET X 5] B [SET X 7] ...]
```

первым будет выполнен оператор [GO B], а следующим — оператор [SET X 7].

Как и все функции, функция GO имеет значение (им является *I*). Однако воспользоваться этим значением никогда не удастся, поэтому данная функция вычисляется исключительно ради ее побочного эффекта — перехода по метке. Как правило, эта метка известна заранее, поэтому чаще всего в обращениях к функции метки указываются явно. Но в общем случае метка может быть и вычислена, что можно использовать, например, для организации переключателя:

[GO [N (A B C D)]].

Здесь будет осуществлен переход по одной из меток A, B, C или D в зависимости от того, какое текущее значение имеет переменная N: 1, 2, 3 или 4.

Отметим, что в плэнерских программах в роли меток могут выступать только те идентификаторы, которые находятся на верхнем уровне блочных функций. Ни в каких других позициях, например внутри условных выражений, идентификаторы не рассматриваются как метки.

Поскольку операторами блоков могут быть обращения к любым функциям, в частности к блочным функциям, то возможна вложенность одних блоков в другие. Все переменные и метки внутреннего блока считаются локализованными в нем и недоступными извне. Поэтому использовать во внешнем блоке переменные внутреннего блока или осуществлять переход по метке внутреннего блока из внешнего запрещено. А вот переменные и метки внешнего блока доступны из внутреннего блока. Так, функция GO сначала просматривает метки ближайшего объемлющего ее блока; если нужная метка здесь есть, то на нее и осуществляется переход. Но если ее нет, тогда функция ищет эту метку во внешнем блоке. Аналогична ситуация и с переменными: если нужная переменная не описана во внутреннем блоке, то она берется из внешнего блока.

Функция RETURN: [RETURN *e*], SUBR.

Выше было сказано, что выход из блока осуществляется после того, как вычислен его последний оператор (если он, конечно, не привел к переходу по какой-либо метке). Однако очень часто необходим выход из «середины» блока. В таком случае как раз и применяется функция RETURN. Ее действие заключается в том, что она полностью завершает вычисление ближайшей блочной функции и объявляет ее значением выражение *E* — значение своего аргумента.

Функция DO: [DO e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Это «составной оператор». Функция по очереди вычисляет свои

аргументы. Значение последнего аргумента является значением функции. Например:

```
[DO [SET X 52] .X]  →  52
```

Функция DO используется в тех случаях, когда по правилам языка в каком-то месте программы должно находиться одно выражение, а по смыслу здесь нужно вычислить несколько выражений. Сделав эти выражения аргументами функции DO, мы удовлетворим обоим этим требованиям.

Отметим, что обращаться к функциям GO и RETURN можно только внутри блочных функций, в то время как обращения к функциям SET и DO допустимы и вне блоков.

Кроме рассмотренных выше операторов присваивания, перехода, выхода и составного, в блоках нужны еще и условные операторы. В роли такого оператора в языке выступает функция COND, поскольку в обращениях к ней можно указывать любые функции, в частности и функции с побочными эффектами. Например, если мы хотим присвоить переменной X значение 2, а переменной Y — значение C, когда значением переменной Z является число, хотим ничего не делать, когда значение Z — шкала, и хотим осуществить переход по метке L в остальных случаях, то это можно сделать, вычислив выражение

```
[COND ([NUM .Z] [SET X 2] [SET Y C])
      ([SCALE .Z])
      (T [GO L])]
```

Таким образом, функция COND используется в языке и как условное выражение, назначение которого — вычислить некоторое значение, и как условный оператор. В любом случае функция вырабатывает некоторое значение, просто во втором случае оно обычно игнорируется; главное здесь — побочные эффекты функций, указанных в условном операторе. Отметим, что при использовании функции COND как условного оператора удобна общая форма обращения к этой функции (см. § 1.8): если при истинности какого-то условия надо выполнить несколько действий, то в соответствующую клаузу записывается последовательность соответствующих операторов (см. первую клаузу в последнем примере); если никаких действий вообще не надо предпринимать при выполнении условия, тогда в клаузу, кроме условия, ничего не записывается (см. вторую клаузу в примере). Когда же COND используется ради вычисления какого-то значения, то, как правило, бессмысленно записывать в клаузе несколько выражений, поскольку значения всех их, кроме последнего, игнорируются. Потому-то при использовании функции COND как условного выражения в

каждой клаузе обычно указывается только условие и одно выражение.

Познакомившись со встроенными функциями, имеющими побочный эффект, мы теперь можем рассмотреть пример использования функции PROG. Предположим, что значением переменной L является список в круглых скобках, и напишем функцию PROG, которая «переворачивает» этот список, т. е. переписывает в обратном порядке его элементы. Переменная L является внешней по отношению к данному блоку.

```
[PROG (L1 (RL ( )) E)
    [SET L1 .L]
L[COND ([EMPTY .L1] [RETURN .RL])]
    [SET E [1 .L1]]
    [SET L1 [REST 1 .L1]]
    [SET RL (.E !.RL)]
    [GO L]]
```

В этом блоке вводятся три локальные переменные L1, RL и E, причем RL получает начальное значение ( ). Первый оператор блока присваивает список L переменной L1; это сделано для того, чтобы не портить значение внешней переменной L, а изменять только значение локальной переменной L1. Все последующие операторы образуют цикл, на каждом шаге которого от списка L1 отщепляется его первый элемент (сам элемент становится значением переменной E, а «хвост» списка — новым значением переменной L1), и этот элемент вставляется в нужном порядке в список RL, где постепенно накапливается «перевернутый» список. Когда список L1 станет пустым, то согласно условному оператору произойдет выход из блока со значением, равным последнему значению переменной RL.

В данном примере идентификатор L использовался и как метка, и как переменная. Это не ошибка. В плэнере имена переменных, имена меток и имена процедур могут совпадать. Это не вызывает путаницы, поскольку позиция, в которой находится идентификатор, однозначно определяет, какой объект обозначен данным идентификатором.

1.10. Циклы

С помощью описанных в предыдущем параграфе средств можно организовать любые циклические вычисления. Однако для удобства написания циклов в язык встроены специальные функции, которые мы и рассмотрим в этом параграфе.

Один из типичных циклов, используемых в плэнерских программах,— это цикл по элементам некоторого списка: определен-

ные действия должны быть повторены для каждого элемента данного списка. Примером подобного цикла может служить задача «переворачивания» списка, рассмотренная в предыдущем параграфе.

Возможный способ организации таких циклов заключается в том, что от списка постепенно отщепляется по одному элементу и для этого элемента выполняются требуемые действия. Если список является, скажем, значением переменной L, то схема этого цикла такова:

```
A [COND ([EMPTY .L] [GO B])]
    [SET E [1 .L]]
    [SET L [REST 1 .L]]
    ...
    [GO A]
B   ...
```

Здесь в начале каждого шага цикла выполняются одни и те же действия: проверяется, не пуст ли список L, и, если не пуст, выделяется его первый элемент, который присваивается некоторой переменной E, а список из остальных элементов делается новым значением переменной L. Для сокращения записи этой группы действий в язык встроена

Функция FIN: [FIN i_1 i_2], SUBR.

Значениями обоих аргументов должны быть имена переменных, описанных в одном из объемлющих блоков, причем переменная I_2 обязана иметь значение. Если им является непустой список, тогда функция присваивает первый элемент этого списка переменной I_1, а «хвост» списка, т. е. список из остальных элементов,— переменной I_2; значение функции в данном случае равно (·). При любом другом значении переменной I_2 (пустом списке или атомарном выражении) функция ничего не делает, ее значение — атом T.

Подчеркнем, что значениями аргументов функции FIN должны быть просто имена переменных, без всяких префиксов. В плэнере, вообще, действует правило: если аргумент некоторой функции — это переменная, которой функция будет что-то присваивать, то имя этой переменной указывается без префиксов.

Используя функцию FIN, описанную выше схему цикла по элементам списка L можно упростить:

```
A [COND ([FIN E L] [GO B])]
    ...
    [GO A]
B   ...
```

Например, «перевернуть» список, являющийся значением переменной L, можно так:

```
[PROG (L1 (RL ( )) E)
    [SET L1 .L]
A [COND ([FIN E L1] [RETURN .RL])]
    [SET RL (.E !.RL)]
    [GO A]]
```

Во многих случаях еще более удобным, чем функция FIN, средством для организации цикла по элементам списка является

Функция LOOP: [LOOP x l e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Первый аргумент x этой функции не вычисляется; он указывает имя переменной, которая играет роль параметра цикла. Эта переменная вводится при входе в функцию LOOP и локализуется внутри нее. Функция действует следующим образом. Она вводит переменную x и вычисляет значение второго аргумента, которое должно быть некоторым списком L. Далее функция последовательно присваивает переменной x элементы списка L и для каждого такого значения переменной x по очереди вычисляет формы e_i. Значением функции является значение формы e_k, вычисленное на последнем шаге цикла. Возможно, что значением l был пустой список, тогда цикл не выполняется, а значением функции является ( ).

Действие функции LOOP эквивалентно вычислению следующего выражения (a и b — вспомогательные переменные, имена которых отличны от названий других переменных):

```
[PROG (x a (b ( )))
    [SET a l]
A [COND ([FIN x a] [RETURN .b])]
    [SET b [DO e1 e2 ... ek]]
    [GO A]]
```

Используя функцию LOOP, «перевернуть» список L можно так:

```
[PROG ((RL ( )))
  - [LOOP E .L [SET RL (.E !.RL)] .RL]]
```

Рассмотрим еще один пример. Подсчитать количество K атомов на верхнем уровне списка L можно следующим образом:

```
[SET K 0]
[LOOP X .L
    [COND ([ATOM .X] [SET K [+ .K 1]])]]
```

В этом примере значение переменной K увеличивалось на 1 с помощью оператора [SET K [+ .K 1]]. Подобное действие, как

и уменьшение значения некоторой переменной на 1, встречается в плэнерских программах довольно часто, поэтому в язык встроены функции ADD1 и SUB1, которые сокращают запись этих действий.

Функция ADD1: [ADD1 *i*], SUBR.

Значением аргумента должен быть идентификатор — имя существующей переменной, текущим значением которой является число. Функция увеличивает на 1 это значение и делает полученное число новым значением переменной. Это же число является и значением функции. Примеры:

```
[DO [SET X 5] (.X [ADD1 X] .X)]   →  (5 6 6)
[DO [SET X Y] [SET Y —7.4]
    [ADD1 .X] .Y]   →  —6.4
```

В последнем примере значением аргумента функции ADD1 является идентификатор Y, поэтому меняется значение именно этой переменной, а не переменной X.

Функция SUB1: [SUB1 *i*], SUBR.

Эта функция действует аналогично функции ADD1, но только не увеличивает, а уменьшает на 1 значение переменной.

Функции ADD1 и SUB1 используются для организации циклов, в которых параметр цикла принимает числовые значения, изменяющиеся с шагом 1 или —1. Например, цикл, параметр K которого меняет свое значение от 1 до некоторого числа N с шагом 1, может быть организован так:

```
    [SET K 0]
A   [ADD1 K]
    [COND ([GT .K .N] [GO B])]
    ...
    [GO A]
B   ...
```

Для большего удобства в написании подобных циклов в язык встроена

Функция FOR: [FOR x n e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Первый аргумент x данной функции не вычисляется и является параметром цикла, локализуемым внутри функции. Функция вводит переменную с именем x и вычисляет значение своего второго аргумента, которое должно быть целым или вещественным числом N. Далее функция по очереди присваивает переменной x значения 1, 2, 3 и т. д. и для каждого такого значения переменной x последовательно вычисляет формы e_i. Цикл прекращается,

когда очередное значение переменной x становится строго больше числа N. Значением функции является значение выражения e_k, вычисленное на последнем шаге цикла. В случае $N < 1$ цикл не выполняется ни разу, а значением функции объявляется пустой список ( ).

Действие функции FOR эквивалентно вычислению следующего выражения (a и b — вспомогательные переменные):

```
[PROG ((x 0) a (b ( )))
   [SET a n]
A  [ADD1 x]
   [COND ([GT .x .a] [RETURN .b])]
   [SET b [DO e1 e2 ... ek]]
   [GO A]]
```

Рассмотрим несколько примеров использования функции FOR. Следующая группа операторов вычисляет факториал F целого числа N:

```
[SET F 1]
[FOR I .N [SET F [× .F .I]]]
```

Действие встроенной функции [MEMB .E .L] (см. § 1.7) можно описать в виде следующего блока:

```
[PROG ( ) [FOR K [LENGTH .L]
    [COND ([EQ [.K .L] .E] [RETURN .K])]]]
```

Функция FOR, как видно из ее определения, предназначена для организации циклов, в которых начальное значение и шаг изменения параметра цикла равны 1. Чаще всего именно такие циклы и используются в плэнерских программах. Но если требуется организовать цикл, где это не так, тогда следует отказаться от использования функции FOR либо приспособиться к ее особенностям. Например, вычислить произведение $P = N \times (N + 1) \times \ldots \times M$ с помощью этой функции можно следующим образом:

```
[SET P .N]
[FOR I [— .M .N] [SET P [× .P [+ .I .N]]]]
```

В плэнере имеется еще две встроенные функции, предназначенные для организации циклических вычислений.

Функция WHILE: [WHILE p e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Действие этой функции эквивалентно вычислению

```
A [COND (p e1 e2 ... ek [GO A])]
```

Таким образом, функция WHILE вычисляет предикат p и, если его значение отлично от ( ), последовательно вычисляет выраже-

ния e_i. Затем эти действия повторяются вновь. Цикл прекращается, когда значение предиката p становится равным ( ). Значением функции всегда является пустой список ( ).

Например, в результате вычисления

```
[SET F [SET K 1]]
[WHILE [LE .F .N]
       [ADD1 K] [SET F [× .F .K]]]
```

переменной F будет присвоено значение наименьшего из факториалов, превосходящих заданное число N.

Функция UNTIL: [UNTIL e_1 e_2 ... e_k p], FSUBR, $k \geqslant 1$.

Действие этой функции эквивалентно вычислению

A [DO e_1 e_2 ... e_k [COND (p [GO A])]]

Следовательно, функция сначала вычисляет формы e_i, а затем вычисляет предикат p. Если его значение отлично от ( ), то данные действия повторяются вновь. В противном случае функция заканчивает свою работу со значением ( ).

1.11. Функция EVAL

Следуя традициям языка лисп, плэнер имеет в числе своих встроенных функций функцию EVAL, с помощью которой можно вычислять выражения, построенные в процессе выполнения программы. Обращение к этой функции имеет следующий вид:

[EVAL e].

Функция относится к классу SUBR, поэтому при обращении к ней прежде всего вычисляется ее аргумент. Значение E этого аргумента функция рассматривает как простую форму (а E обязано ею быть) и вычисляет ее, объявляя полученное значение своим значением. Таким образом, функция EVAL вычисляет значение значения своего аргумента.

Пусть, к примеру, значением переменной X является список [+ 2 4], тогда имеем

[EVAL .X] → 6

Действительно, значением аргумента является список [+ 2 4], вычислив который, функция EVAL и получает число 6.

Вычисление выражения, являющегося значением переменной,— довольно типичное применение функции EVAL в плэнерских программах. Такие выражения либо строятся во время

выполнения программы, либо известны заранее (при написании программы), но их нельзя было вычислять раньше времени.

Другое применение функции EVAL — сразу вычислить только что построенные выражения. Например, если требуется вычислить условное выражение из двух клауз, которые заранее неизвестны, но являются значениями переменных X и Y, то это можно сделать, вычислив выражение

```
[EVAL [FORM [COND .X .Y]]]
```

Если, скажем, значением переменной X является список (( ) A), а значением переменной Y — список (T B C), то в результате вычисления указанного выражения получится атом C. В самом деле, при данных значениях переменных X и Y функция FORM строит список [COND (( ) A) (T B C)], который функция EVAL и вычисляет, получая в результате атом C.

Еще один пример применения функции EVAL. Пусть значением переменной L является непустой список, элементы которого — числа, и требуется вычислить сумму S этих чисел. Следующие обращения к функции +

```
[+ .L] и [+ !.L]
```

не дадут нам нужной суммы, а приведут лишь к ошибке. Действительно, в первом случае значением аргумента является список, а требуется число, во втором же случае в качестве аргумента задана не простая форма, а сегментная, что также недопустимо. Возможный выход — организовать цикл

```
[SET S 0]
[LOOP N .L [SET S [+ .S .N]]]
```

Однако более эффективен иной выход: сначала построить «законное» обращение к функции сложения — $[+\ l_1\ l_2\ \ldots\ l_m]$, где l_i — числа из списка L, а затем уже вычислить это выражение, обратившись к функции EVAL. Таким образом, наша задача может быть решена следующим образом:

```
[SET S [EVAL [FORM [+ !.L]]]]
```

1.12. Определение новых функций

До сих пор мы рассматривали лишь встроенные функции языка, продолжим их изучение и позже. Но сколько бы встроенных функций не было в языке, в любой более или менее сложной программе возникает необходимость в новых функциях, которые выполняют действия, специфичные именно для этой программы. Язык допускает использование таких функций. Для этого поль-

вователь должен сначала описать их, после чего он имеет право обращаться к ним наравне со встроенными функциями.

Для определения новой функции нужно обратиться к встроенной функции DEFINE класса FSUBR, указав имя, параметры и правила вычисления определяемой функции:

[DEFINE *fn* (LAMBDA *var body*)]

Функция DEFINE не вычисляет свои аргументы. Первый ее аргумент — идентификатор *fn* — указывает имя, которое выбрано для определяемой функции. Второй аргумент — список (LAMBDA *var body*) — называется *определяющим выражением* функции *fn*. Его первый элемент LAMBDA указывает, что определяется именно функция, а не процедура иного типа (сопоставитель или теорема), второй элемент *var* описывает (формальные) *параметры* определяемой функции, а третий элемент — простая форма *body* — называется *телом* функции *fn* и описывает алгоритм ее вычисления.

В результате выполнения функции DEFINE вырабатывается значение *fn*. Однако главное в работе данной функции заключается в ее побочном эффекте — в том, что она вводит в программу новую функцию с указанным именем и определяющим выражением, к которой теперь можно обращаться из любой точки программы. (Отметим, что если имя новой функции совпадает с именем другой процедуры или с именем константы, то эта процедура или константа автоматически уничтожается.)

Обращение к новой функции записывается по тем же правилам, что и обращения к встроенным функциям:

$[fn\ a_1\ a_2\ \ldots\ a_k]$ или $\langle fn\ a_1\ a_2\ \ldots\ a_k\rangle$,

где *fn* — имя функции, определенной пользователем, а a_i — аргументы (фактические параметры), при которых функция должна быть вычислена.

Сколько аргументов можно задавать при обращении к функции *fn*, какие ограничения на них накладываются — все это определяется элементом *var* из определяющего выражения данной функции. Синтаксически *var* может быть идентификатором, *-переменной или списком, составленным из идентификаторов и *-переменных.

Если *var* — идентификатор, тогда это имя единственного (формального) параметра функции. В данном случае к функции можно обращаться с любым числом аргументов, которые могут быть как простыми, так и сегментными формами. Во время вычисления обращения к функции все эти аргументы будут последовательно вычислены и из их значений будет составлен L-список, который станет значением этого единственного параметра функции.

Если *var* — идентификатор, помеченный префиксом «*», тогда этот идентификатор также является именем единственного параметра функции и при обращении к функции можно также задавать любое количество аргументов. Однако в данном случае аргументы могут быть любыми выражениями, поскольку они не вычисляются: при обращении к функции ее параметру присваивается L-список, составленный из невычисленных аргументов.

И, наконец, *var* может иметь вид $(v_1\ v_2\ \ldots\ v_k)$, $k \geqslant 0$, где v_i — либо просто идентификаторы, либо идентификаторы, помеченные звездочкой. У такой функции ровно k параметров, их имена — идентификаторы v_i (без звездочек), и при обращении к функции нужно указывать ровно k аргументов. Когда вычисляется обращение к функции, i-му параметру присваивается либо значение i-го аргумента (он должен быть простой формой), если v_i не помечен звездочкой, либо сам i-й аргумент (любое выражение) в том виде, как он задан в обращении, если v_i помечен.

Обращение к функции, определенной пользователем, вычисляется следующим образом. Сначала, если нужно, вычисляются аргументы (все или только часть их), заданные в обращении. Затем вводятся параметры функции, и им присваиваются указанным выше образом соответствующие значения, после чего вычисляется тело функции. Параметры функции локализуются в ее теле, где они рассматриваются как обычные переменные. Результат вычисления тела функции объявляется значением данного обращения к функции.

Определяемые функции могут быть рекурсивными. Никаких ограничений на рекурсии в плэнере нет. В частности, допускается косвенная рекурсия, когда одна функция обращается к другой, а та обращается (непосредственно или косвенно) к первой. При этом разрешено в теле любой определяемой функции F указывать обращения к пока еще не определенным функциям (не определенным в момент вычисления функции DEFINE, которая вводит в программу функцию F). Но, естественно, к моменту вычисления первого обращения к функции F все такие функции должны быть уже определены.

Рассмотрим описанные выше правила определения функций и вычисления обращений к ним на конкретных примерах.

В результате вычисления выражения

```
[DEFINE IF (LAMBDA (P *E1 *E2)
    [COND (.P'[EVAL .E1]) (T [EVAL .E2])])]
```

будет определена функция с именем IF, действие которой аналогично вычислению конструкции **if** P **then** E1 **else** E2 языка алгол-60. В частности, аналогия распространяется и на то, что из двух выражений E1 и E2 вычисляться всегда будет лишь одно.

Например, при значении переменной W, равном (A), обращение

```
[IF [EMPTY .W] [SET X A] [SET Y B]]
```

вычисляется следующим образом. Поскольку первый параметр функции IF не помечен, первый аргумент вычисляется; второй же и третий параметры помечены, поэтому два других аргумента не вычисляются. Следовательно, параметрам присваиваются такие значения: P:= ( ), E1:= [SET X A], E2:= [SET Y B]. Теперь вычисляется тело функции IF, т. е. условное выражение. Так как значение переменной P равно ( ), в условном выражении вычисляется выражение из второй клаузы, т. е. выражение [EVAL .E2]. Вычисление его сводится к вычислению выражения [SET Y B]. Таким образом, переменной Y присваивается значение B, а значением тела функции IF становится атом B. Этот же атом является и значением указанного обращения к функции IF.

Данный пример показывает, какие параметры определяемых функций следует метить звездочкой, а какие — нет. Если некоторые аргументы функции вообще не должны вычисляться либо они будут вычисляться, но только во время вычисления тела функции, тогда соответствующие параметры необходимо пометить. Если же в теле функции должны использоваться значения аргументов, тогда соответствующие параметры не метятся.

В следующем примере определяется функция, обращаться к которой можно с любым количеством аргументов, причем при обращении все эти аргументы вычисляются.

```
[DEFINE CA (LAMBDA N
    [/ [EVAL [FORM [+ !.N]]] [LENGTH .N]])]
```

Данная функция подсчитывает среднее арифметическое (числовых) значений своих аргументов. Пусть, например, значением переменной X является число 5.8, тогда обращение

```
[CA 6 .X [+ .X 2.2]]
```

вычисляется так. Поскольку в определяющем выражении данной функции в качестве описания параметров указан идентификатор N, то это означает, что все аргументы, указанные в обращении, вычисляются и что из их значений составляется список, который присваивается параметру N: N:= (6 5.8 8.0). Далее вычисляется тело функции CA — обращение к функции деления, первый аргумент который находит сумму чисел списка N (см. § 1.11), а второй — общее количество этих чисел. Таким образом, значением функции CA является частное от деления 19.8 на 3, т. е. число 6.6.

Отметим, что в общем случае правила языка не накладывают ограничений на значения аргументов определяемых функций. Однако алгоритмы вычисления этих функций обычно ограничивают

область допустимых значений аргументов. Так, значениями аргументов функции CA могут быть только числа. Подобного рода ограничения никак не фиксируются в плэнерских программах; помнить о них и придерживаться их обязан сам пользователь, если он не хочет, чтобы вычисление функции привело к ошибке.

Примером функции с произвольным числом аргументов, которые заранее не вычисляются, может служить следующая функция OR1, действие которой эквивалентно вычислению встроенной функции OR:

```
[DEFINE OR1 (LAMBDA *DIS [PROG (D)
    A [COND ([FIN D DIS] ())
            ([EVAL .D])
            (T [GO A])]])]
```

Например, обращение

```
[OR1 [ATOM .X] [EQ .X (A)]]
```

при значении (A) у переменной X вычисляется так. Параметр функции OR1 помечен, поэтому ему присваивается список из невычисленных аргументов: DIS:= ([ATOM .X] [EQ .X (A)]). Далее вычисляется тело функции OR1, т. е. функция PROG, которая вводит еще одну переменную D и организует цикл по элементам списка DIS. На первом шаге цикла переменная D получает значение [ATOM .X], а переменная DIS — значение ([EQ .X (A)]), после чего вычисляется [EVAL .D], т. е. [ATOM .X]. Поскольку эта форма имеет значение «ложь», выполняется оператор перехода из третьей клаузы условного выражения. На втором шаге цикла переменная D получает значение [EQ .X (A)], а значением переменной DIS становится пустой список ( ). Вычисление формы [EVAL .D] в этот раз сводится к вычислению [EQ .X (A)], что дает значение T, поэтому вычисление функции COND и, следовательно, функции PROG заканчивается с этим значением. Теперь уничтожается как переменная D из тела функции PROG, так и параметр DIS, и значением всего обращения к функции OR1 объявляется атом T.

Приведем также пример рекурсивной функции:

```
[DEFINE NA (LAMBDA (L) [COND
    ([ATOM .L] 1)
    ([OR [VAR .L] [EMPTY .L]] 0)
    (T [+ [NA [1 .L]] [NA [REST 1 .L]]])])]
```

Эта функция определяет, сколько атомов входит (на всех уровнях) в выражение L. Если L — атом, то значением функции объявляется 1. Если L — иное атомарное выражение или пустой список, то значение функции равно 0. Иначе L является непустым спис-

ком, и в этом случае функция NA рекурсивно применяется к первому элементу списка L, чтобы определить сколько атомов входит в этот элемент, и к «хвосту» списка L, чтобы подсчитать число атомов в этой части списка. Затем полученные числа складываются, что и дает общее количество атомов, входящих в весь список L. Например:

```
[NA (A (B 2 (C)) *70)]  →  5
```

1.13. Программа

Программа на плэнере представляет собой последовательность выражений (точнее, простых форм). Эти выражения принято называть *выражениями верхнего уровня программы*, чтобы отличить их от подвыражений, из которых они состоят. Выполнить программу — это значит последовательно вычислить выражения ее верхнего уровня.

Более точно, вычисление программы происходит так. Считывается ее первое выражение, и оно печатается на АЦПУ (при работе с плэнер-системой в пакетном режиме) или на экране терминала (в диалоговом режиме). Затем это выражение вычисляется. Полученное значение также выводится на АЦПУ или терминал. Далее все эти действия повторяются по отношению ко второму, третьему и остальным выражениям верхнего уровня программы. После вычисления последнего выражения выполнение программы прекращается.

Например, при вычислении программы, состоящей из выражений

```
[REST 1 (A B C)]
[DEFINE NATOM (LAMBDA (X) [NOT [ATOM .X]])]
[NATOM *5]
```

будет напечатано следующее:

```
[REST 1 (A B C)]
(B C)
[DEFINE NATOM (LAMBDA (X) [NOT [ATOM .X]])]
NATOM
[NATOM *5]
()
```

Из каких выражений составлять программу — это решает пользователь, но обычно структура плэнерской программы такова. Сначала выписываются обращения к функции DEFINE — определяются новые функции, необходимые для решения задачи, поставленной перед программой, а затем выписывается одно или несколько обращений к этим функциям. Более того, в программе, как

правило, выделяется одна функция, которая называется основной (или главной, ведущей) и которая решает всю задачу. Остальные же функции являются вспомогательными, они решают какие-то подзадачи и используются основной функцией. В конце программы в таком случае ставится одно обращение к ее основной функции, в качестве аргументов для которой задаются исходные данные программы.

В качестве небольшого примера плэнерской программы рассмотрим программу, которая преобразует запись арифметических выражений из инфиксной формы в префиксную. Инфиксная форма записи — это обычная форма записи формул, когда знак операции размещается между операндами: $x + y$, $x \times y + z \times w \times v$. В префиксной же записи знак операции ставится перед операндами; например, указанные выше формулы записываются в префиксной форме так: $(+\ x\ y)$, $(+\ (\times\ x\ y)\ (\times\ z(\times\ w\ v)))$. Префиксная форма записи арифметических выражений более удобна для обработки, поэтому часто до обработки выражений их преобразуют из привычной инфиксной формы в префиксную. Именно эта задача перевода нас и будет интересовать, причем для простоты мы рассматриваем преобразование не любых арифметических выражений, а только тех, которые содержат переменные, соединенные знаками $+$ и $\times$.

Назовем условно такие выражения «многочленами». Более точно их можно определить так. «Переменная» — это любой плэнерский идентификатор, «одночлен» — это либо одна переменная, либо последовательность переменных, соединенных знаком $\times$, а «многочлен» — это L-список из одного одночлена или из нескольких одночленов, соединенных знаком $+$. Примеры таких многочленов:

(X) (X + Y × Z) (X × Y + Z × W × V)

Для решения поставленной задачи мы введем три новых функции: основную функцию TRANSL и две вспомогательные функции MONO и POLY.

Функция MONO от одного аргумента решает такую задачу: она преобразует одночлен M (точнее, одночлен, заключенный в скобки) из инфиксной формы в префиксную. Если M — список из одной переменной, тогда функция в качестве своего значения выдает эту же переменную, но без скобок. Иначе M имеет вид $(x \times y)$, где x — переменная, а y — одночлен. В данном случае функция в качестве ответа выдает список $(\times\ x\ \tilde{y})$, где $\tilde{y}$ — префиксная форма записи одночлена y, полученная в результате рекурсивного применения функции MONO.

Определение функции MONO — это первое выражение нашей программы:

```
[DEFINE MONO (LAMBDA (M)
   [COND ([EQ [LENGTH .M] 1] [1 .M])
         (T (× [1 .M] [MONO [REST 2 .M]]))])]
```

Вторым выражением программы является определение функции POLY от одного аргумента — многочлена P:

```
[DEFINE POLY  (LAMBDA (P) [PROG (K)
   [SET K [MEMB + .P]]
   [COND ([NOT .K] [RETURN [MONO .P]])]
   (+ [MONO [HEAD [— .K 1] .P]]
      [POLY [REST .K .P]])])]
```

Данная функция преобразует инфиксную форму записи многочлена P в префиксную. Для этого функция прежде всего определяет, есть ли в P хотя бы один знак +. Если нет, тогда P является одночленом, поэтому функция POLY применяет к нему функцию MONO и с ее значением заканчивает свою работу. Иначе значение переменной K указывает номер первого атома + в списке P. Этот знак делит многочлен на две части: слева от него находится одночлен, справа — многочлен. Обе эти части преобразуются в префиксную форму, для чего к многочлену рекурсивно применяется функция POLY, а к одночлену — функция MONO. Далее из знака + и полученных результатов составляется список с круглыми скобками, который и представляет собой префиксную форму записи исходного многочлена P. Этот список объявляется значением функции POLY.

Функция POLY фактически полностью решает нашу задачу за одним исключением: если исходный многочлен является списком из одной переменной, скажем (X), то функция выдает в качестве ответа не (X), а просто X. Чтобы устранить этот дефект, мы вводим еще одну функцию TRANSL, которая и будет главной функцией программы. Определение данной функции — это третье выражение программы

```
[DEFINE TRANSL (LAMBDA (L)
      [COND  ([EQ [LENGTH .L] 1] .L)
             (T [POLY .L])])]
```

Указанные три выражения составляют, можно сказать, описание программы. Теперь, чтобы применить программу к каким-то исходным данным, надо вставить в программу еще одно выражение — обращение к функции TRANSL, задав в качестве аргумента тот многочлен, который мы хотим преобразовать. Так:

```
[TRANSL (X × Y + Z × W × V + U)]
```

В результате вычисления этого выражения мы получим ответ

```
(+ (× X Y) (+ (× Z (× W V)) U))
```

1.14. Переменные и константы

Любая используемая в программе переменная должна быть описана. Переменные описываются в блочной функции PROG, в функциях LOOP и FOR, в определяемых функциях и в ряде других процедур языка, которые будут рассмотрены позже. Каждая переменная локализуется в теле той функции, в которой она описана, т. е. переменная существует только во время вычисления этой функции и не существует вне ее.

Как уже было сказано, именами переменных могут быть только идентификаторы. В языке разрешается совпадение имен переменных с именами процедур, констант и меток. Любая существующая переменная может как иметь значение, так и не иметь его. Значением переменной может быть любое выражение. Если переменная не имеет значения, то возможны ограничения на ее будущее значение, о чем будет рассказано в гл. 5. Пока мы не будем учитывать эти ограничения.

В любой функции допускается использование как ее локальных переменных, так и внешних по отношению к ней переменных. Имена локальных и внешних переменных могут совпадать. В таком случае предпочтение отдается переменной, описанной в ближайшей из активных (вычисление которых еще не окончено) объемлющих процедур. Следовательно, если в процедуре F используется переменная X и она описана в F, то в F используется ее локальная переменная X. Но если X не описана в F, тогда берется переменная с этим именем из процедуры G, которая вызвала процедуру F. Если переменная X не описана и в G, тогда рассматривается процедура, вызвавшая процедуру G, и т. д.

Такое правило называется правилом *динамической идентификации переменных*, так как смысл переменных определяется динамикой выполнения программы и может меняться в зависимости от того, по какому пути идет вычисление программы. В самом деле, процедура F может быть вызвана из разных процедур: в одном случае, скажем, из процедуры G1, а в другом — из G2. Поэтому в первом случае под переменной X, используемой, но не описанной в F, понимается переменная из G1, а во втором — из G2.

В предыдущих параграфах уже был рассмотрен ряд встроенных функций языка (SET, FIN, ADD1 и SUB1), используемых при работе с переменными. Сейчас мы опишем еще несколько подобных функций.

Функция BOUND: [BOUND *i*], SUBR.

Это — функция-предикат, которая позволяет узнать, описана ли в одной из объемлющих процедур переменная с именем *I*. Если описана, то значение функции равно T, не описана — ( ).

Например, если следующий блок

```
[PROG (X (Y X))
    ([BOUND Y] [BOUND .Y] [BOUND Z])]
```

является выражением верхнего уровня программы, то его вычисление дает результат (T T ( )).

Функция HASVAL: [HASVAL *i*], SUBR.

Значением аргумента этой функции-предиката должно быть имя описанной переменной. Если в текущий момент данная переменная имеет какое-либо значение, то функция в качестве своего значения выдает атом T, а если не имеет — пустой список ( ).

Например, если функция F определена следующим образом:

```
[DEFINE F (LAMBDA (X) [PROG (Y (Z Y))
    ([HASVAL X] [HASVAL Z] [HASVAL .Z])])]
```

тогда значением выражения [F 5] будет список (T T ( )).

Функция UNASSIGN: [UNASSIGN *i*], SUBR.

Значением аргумента данной функции также должно быть имя описанной переменной. Функция «отнимает» значение у этой переменной, т. е. после вычисления функции у переменной не будет никакого значения. Значение самой функции — идентификатор *I*.

Функция VALUE: [VALUE *i*], SUBR.

И здесь значением аргумента должен быть идентификатор — имя описанной переменной, причем переменная обязана иметь значение. Именно это значение и является результатом вычисления данной функции.

Если, например, значением переменной Y является идентификатор X, а значением переменной X — число 5, то

```
[VALUE .Y]  →  5
```

Отметим, что выражение [VALUE Y] эквивалентно выражению .Y, поэтому явно указывать имя переменной при обращении к функции VALUE невыгодно. Эта функция обычно применяется в тех случаях, когда нужно получить значение переменной, имя которой заранее неизвестно, а узнается только во время выполнения программы. В качестве типичного примера приведем определение функции ADD-ONE, аналогичной встроенной функции ADD1:

```
[DEFINE ADD-ONE (LAMBDA (==)
    [SET .== [+ 1 [VALUE .==]]])]
```

Пусть существует переменная с именем X и значением 4. Тогда в результате вычисления выражения [ADD-ONE X] значение

переменной X увеличится на 1. Действительно, при вычислении этого выражения параметру == функции присвоится идентификатор X. Тело функции — это обращение к функции SET, которая присваивает переменной X (этот идентификатор является значением ее первого аргумента) значение своего второго аргумента. При вычислении этого аргумента к 1 добавляется значение выражения [VALUE .==], равное числу 4 — значению переменной X, имя которой получилось в результате вычисления формы .==. Таким образом, переменной X будет присвоено число 5, что и требовалось.

Почему для обозначения параметра функции ADD-ONE выбран необычный идентификатор ==? Ответ такой. Если бы мы обозначили этот параметр каким-то обычным идентификатором, скажем X:

```
[DEFINE ADD-ONE (LAMBDA (X)
   [SET .X [+ 1 [VALUE .X]]])]
```

то наша функция неправильно бы работала в том случае, когда при обращении к ней указывалась переменная с именем X. В самом деле, пусть по-прежнему имеется переменная X со значением 4, и пусть мы хотим увеличить это значение на 1, для чего вычисляем [ADD-ONE X]. При вычислении этого обращения к функции вводится параметр X, значением которого становится идентификатор X, поэтому при вычислении [VALUE .X] складывается следующая ситуация: значением формы .X является идентификатор X, поэтому VALUE берет значение переменной X, но какой? У нас сейчас две переменные с именем X: параметр функции ADD-ONE и переменная со значением 4. Поскольку описание параметра «ближе», чем описание переменной, то VALUE берет значение именно параметра, т. е. идентификатор X. Именно к этому идентификатору будет прибавляться 1, что, естественно, приведет к ошибке.

Таким образом, совпадение имени параметра функции ADD-ONE с именем переменной, существовавшей до обращения к функции, привело к тому, что эта переменная стала недоступной из тела функции, поэтому взять или изменить ее значение в теле функции не удастся. Избавиться от такого дефекта в определении функции ADD-ONE нельзя (если ее параметр обозначен как ==, тогда функция неприменима к переменной с именем ==). Однако выбрав в качестве названия параметра «экзотический» идентификатор ==, которым редко обозначают имена переменных, мы уменьшили вероятность проявления дефекта функции ADD-ONE.

В предыдущей части данного параграфа под «переменными» понимались «локальные переменные», которые вводятся только

на время выполнения какой-то процедуры. Но в языке имеются и переменные иного сорта — глобальные переменные, которые называются константами. Константы отличаются от локальных переменных тем, что они существуют с момента своего появления и до конца вычисления программы и что они доступны из любой точки программы.

Одно из применений констант — это обозначение каких-либо величин. Например, чтобы каждый раз не выписывать много цифр числа π, можно ввести в программе константу с именем PI, присвоив ей в качестве значения число 3.141592653, и дальше использовать в программе только краткое обозначение PI. Именно поэтому такие объекты в языке и называются константами. Однако это не означает, что раз присвоенное значение константы больше нельзя изменить. В плэнере константы могут менять свои значения, и в этом они похожи на локальные переменные. Эта возможность, а также то, что константы доступны из любого места программы, обусловливает еще одно назначение констант — служить переменными, через которые независимые процедуры программы могут обмениваться информацией. Дело в том, что на практике не всегда удобно объединять действия отдельных функций плэнерской программы в рамки какой-то одной, основной функции. Однако независимые функции все же должны как-то передавать свои данные друг другу. Именно здесь и удобны константы, которые доступны из любой функции и значения которых сохраняются независимо от того, какая функция сейчас вычисляется.

Именами констант могут быть только идентификаторы, а значениями — любые выражения. Константа без значения не существует. Обратиться к константе можно в любой момент и в любом месте программы. При обращении к константе указывается ее имя, перед которым ставится префикс «:» или «!:». Обращение с префиксом «:» (например, :PI) называется **простым обращением к константе**; значением такой формы является текущее значение константы. **Сегментное обращение к константе** задается префиксом «!:», а результатом такого обращения является сегментированное значение константы. Например, если :C → (A B ⟨C⟩), то !:C → A B ⟨C⟩.

Определить константу можно в любой момент и в любой точке программы, после чего она становится доступной во всей программе. Изменить значение константы также можно в любой момент и в любом месте. Оба этих действия выполняет

Функция CSET: [CSET *i e*], SUBR.

Значением первого аргумента должен быть идентификатор *I*, а значением второго аргумента может быть любое выражение *E*. Если константы с именем *I* в программе еще нет, тогда функция

вводит новую константу с этим именем и значением *E*. Если же константа *I* уже существовала, то функция присваивает ей новое значение — выражение *E*. Таков побочный эффект функции CSET, значением же ее является выражение *E*.

Например, после вычисления [CSET PI 3.141592653] в программе появится константа с именем PI и значением 3.141592653.

Имена констант не должны совпадать с именами процедур (функций, сопоставителей и теорем). Если определяется константа, имя которой совпадает с именем процедуры, то последняя уничтожается. Верно и обратное: определение процедуры (с помощью функции DEFINE) уничтожает константу с тем же именем.

С именами же локальных переменных (а также меток) названия констант могут совпадать. Вопрос, к чему происходит обращение — к константе или переменной, легко решается по префиксу, указанному в обращении. Например:

```
[PROG (X) [SET X A] [CSET X B]
      (.X :X)]   →   (A B)
```

В дальнейшем изложении мы оставим за локальными переменными название «переменные», а глобальные переменные будем называть только «константами».

1.15. Ввод-вывод

Как мы видели в § 1.13, исходные данные для плэнерской программы могут быть заданы как аргументы ее основной функции, а значение этой функции можно рассматривать как результат вычисления всей программы, причем этот результат автоматически выводится на печать. Однако на практике такой способ задания исходных данных и печати результата неудобен, поэтому в язык встроены процедуры ввода-вывода, позволяющие запрашивать данные для программы и печатать ее результаты в любой нужный момент.

Функция READ: [READ].

Эта функция, не имеющая аргументов, осуществляет ввод информации. Она считывает очередное выражение из числа заданных для ввода и, не вычисляя его, объявляет своим значением.

Например, если для ввода подготовлена следующая последовательность выражений:

```
(A B C)
5.2
[ELEM 1 .X]
```

то при вычислении выражений

[SET X [READ]]
[SET Y ([READ] ⟨READ⟩)]

переменной X будет присвоено значение (A B C), а переменной Y — значение (5.2 ELEM 1 .X).

Функция PRINT: [PRINT *e*], SUBR.

Данная функция выводит (например, печатает на АЦПУ) значение своего аргумента. В этом заключается побочный эффект функции, значение же ее совпадает со значением аргумента.

Например, при вычислении выражения

[DO [SET X (A B)] [SET Y [PRINT .X]]]

переменные X и Y получат одно и то же значение — список (A B), и этот же список будет выведен на печать.

Функция PRINT печатает только одно выражение, причем печатает его с новой строки. Нередко же требуется напечатать «в одну строку» сразу несколько выражений. В таких случаях применяется другая функция вывода —

Функция MPRINT: [MPRINT $e'_1\ e'_2\ \ldots\ e'_k$], SUBR, $k \geqslant 1$.

Аргументами e'_i этой функции могут быть как простые, так и сегментные формы. Функция вычисляет список $(e'_1\ e'_2 \ldots\ e'_k)$ и объявляет его значение своим результатом. Одновременно функция выводит (например, на АЦПУ) значение этого списка, но без внешних круглых скобок. Другими словами, функция печатает значения своих аргументов «в одну строку».

Если, к примеру, значением переменной ФРАЗА является список (КИСКА ЛЮБИТ МОЛОКО), а значением переменной СЛОВО — атом КИСКА, то в результате вычисления выражения

[MPRINT B " !.ФРАЗА " НЕПОНЯТНО: .СЛОВО]

получится значение (B " КИСКА ЛЮБИТ МОЛОКО " НЕПОНЯТНО: КИСКА) и будет напечатано:

B " КИСКА ЛЮБИТ МОЛОКО " НЕПОНЯТНО: КИСКА

Сделаем замечание относительно вывода вещественных чисел. При выводе они всегда печатаются с фиксированным количеством цифр в дробной части. Узнать это количество или изменить его можно с помощью следующей встроенной функции.

Функция DIGITS: [DIGITS *n*?], SUBR.

Если аргумент задан, то его значением должно быть неотрицательное целое число *N*; это число указывает то количество «дробных» цифр, с которым теперь (до следующего обращения к DIGITS) будут печататься вещественные числа. Значением функции является число *N*. Если же аргумента нет, то функция ничего не меняет, а в качестве своего значения выдает текущее количество печатаемых «дробных» цифр.

Например, при вычислении выражения

```
[DO [DIGITS 5]
    [PRINT —9.25] [PRINT 2.718281828]]
```

на печать будет выдано: —9.25000 и 2.71828.

Мы описали функции ввода-вывода. Теперь рассмотрим, откуда они вводят информацию и куда ее выводят.

В плэнере под «вводом-выводом» понимается считывание с перфокарт и печать на АЦПУ, обмен с терминалом и обмен с внешней памятью ЭВМ. Для единообразия мы будем пользоваться термином *файл* для обозначения любого набора данных на любом внешнем носителе ЭВМ. В языке предполагается, что каждый файл состоит из плэнерских выражений, считывать и записывать которые можно только последовательно, друг за другом. Файлы, из которых можно считывать, будем называть *файлами ввода*, а файлы, в которые можно записывать,— *файлами вывода*. Некоторые файлы можно использовать и как файлы ввода, и как файлы вывода.

Предполагается, что файлы имеют имена, в роли которых выступают идентификаторы. За некоторыми, *стандартными*, файлами в языке закреплены фиксированные имена: CARDS — текст на перфокартах, PAPER — текст на бумаге АЦПУ, SCREEN — информация на экране терминала. Файл CARDS используется только как файл ввода, PAPER — только как файл вывода, а SCREEN — и как файл ввода, и как файл вывода. Имена остальных, нестандартных, файлов в языке не фиксируются и определяются дополнительными, внеязыковыми соглашениями. Нестандартные файлы — это наборы данных на магнитных лентах и дисках, они могут использоваться как файлы ввода и как файлы вывода.

Прежде чем файл может быть использован в программе, его следует *открыть*. При открытии файла устанавливается его соответствие реальному набору данных и выполняются подготовительные операции по организации обмена с ним. Стандартные файлы открываются автоматически перед началом выполнения программы, а нестандартные файлы должны быть открыты в самой программе. Для этого в распоряжение пользователя предоставлена

Функция OPEN: [OPEN *file*? *type*], SUBR.

Значением первого аргумента этой функции должен быть идентификатор — имя открываемого файла, а значением второго аргумента может быть атом GET, PUT или ADD. Если в обращении к функции указаны оба аргумента и значением второго из них является атом GET, тогда функция открывает файл с указанным именем как файл ввода; тем самым теперь разрешено считывание из этого файла (начиная с его первого выражения). Если же значение второго аргумента равно PUT или ADD, то функция открывает указанный файл как файл вывода, т. е. с этого момента разрешена запись в него. При этом, если файл открывается по ADD, то новые выражения будут добавляться в конец файла, вслед за уже имеющимся текстом, а если файл открывается по PUT, тогда файл считается пустым и будет заполняться от начала. Значение функции OPEN равно T, если файл открыт, и равно (), если указанный файл не существует или является одним из стандартных, которые запрещено переоткрывать.

Например, после вычисления

[OPEN DATA GET] → T

файл с именем DATA будет открыт как файл ввода.

К функции OPEN можно обращаться и с одним (вторым) аргументом, значением которого в таком случае должен быть атом GET или PUT. В этом случае функция ничего не открывает, а в качестве своего значения выдает список из имен всех открытых в настоящий момент файлов ввода (при аргументе GET) или файлов вывода (при PUT). Ответ функции может быть, например, таким:

[OPEN PUT] → (PAPER SCREEN FILE5)

В программе одновременно может быть открыто несколько файлов ввода и несколько файлов вывода. Однако в каждый момент функции ввода (вывода) настроены на работу только с одним из открытых файлов ввода (вывода), только из него они могут сейчас считывать (записывать в него). Такие файлы называются *активным файлом ввода* и *активным файлом вывода*. Перед началом вычисления программы активные файлы устанавливаются автоматически. При работе с плэнер-системой в пакетном режиме активными объявляются файл ввода CARDS и файл вывода PAPER, т. е. функции ввода первоначально настроены на считывание с перфокарт, а функции вывода — на печать на АЦПУ. В диалоговом же режиме в роли активных файлов ввода и вывода выступает файл SCREEN, т. е. функции ввода-вывода вначале настроены на обмен с терминалом. В дальнейшем, при вычислении программы, активными могут быть объявлены и любые другие открытые файлы. Для этого используется

Функция ACTIVE: [ACTIVE *file*? *type*], SUBR.

Значением первого аргумента должно быть имя файла, а значением второго — атом GET или PUT. Если заданы оба аргумента, тогда функция объявляет указанный файл активным файлом ввода (при значении GET у второго аргумента) или активным файлом вывода (при PUT). Если это удалось сделать, значение функции равно T, не удалось (файл ранее не был открыт) — равно ( ).

В том случае, когда задан только один (второй) аргумент, функция выдает в качестве своего значения имя активного файла ввода (при аргументе GET) или активного файла вывода (при PUT). Значением функции может быть и пустой список ( ), если в настоящий момент нет активного файла соответствующего типа.

Следует подчеркнуть, что функция ACTIVE не открывает и не переоткрывает файлы, а лишь переключает «внимание» функций ввода (или вывода) с одного из уже открытых файлов на другой, заставляя эти функции работать с новым файлом. При этом, если такой файл ранее уже был активным файлом ввода (вывода), считывание (запись) из него будет продолжено с того места, на котором было приостановлено чтение (запись) в последний раз. Отметим также, что если с помощью функции OPEN (при аргументе GET или PUT) переоткрывается файл, являющийся сейчас активным, то он по-прежнему остается активным, но считывание (запись) из него теперь будет происходить от его начала.

Еще одна операция над файлами — это *закрытие* файла, в результате которой файл становится недоступным программе. Закрывает файлы

Функция CLOSE: [CLOSE *file type*], SUBR.

Если значением второго аргумента является атом GET, то функция закрывает указанный файл как файл ввода (если этот же файл был открыт и как файл вывода, то в таком качестве он не закрывается). В противном случае второй аргумент должен иметь значение PUT, и тогда указанный файл закрывается как файл вывода. Значение функции равно T, если файл удалось закрыть, и равно ( ) в противном случае.

Отметим, что если закрывается активный файл, то новый активный файл не устанавливается. Попытка вычислить в этот момент функцию ввода или вывода приведет к ошибке. Ошибкой также является и выход (при записи) за границы файла, имеющего ограниченные физические размеры (таковы, например, файлы во внешней памяти ЭВМ), или считывание из файла, в котором не осталось ни одного выражения *). Любую из этих ошибок мож-

*) При переполнении файла вывода или считывании из пустого файла ввода сначала автоматически закрывается файл, а затем уже вырабатывается сигнал ошибки.

но перехватить с помощью встроенной функции CATCH (см. § 1.16), а, кроме того, ошибки в последнем из указанных случаев можно избежать, если воспользоваться встроенной функцией:

Функция EOF: [EOF].

Это — функция-предикат, с помощью которой можно узнать, пуст или нет активный файл ввода. Если в текущий момент в этом файле есть хотя бы одно выражение *), тогда функция вырабатывает значение ( ), а если он пуст или если сейчас нет активного файла ввода, значение функции равно T.

Для иллюстрации работы с файлами и функциями ввода-вывода рассмотрим, как можно переписать все выражения файла A в конец файла B (вслед за имеющимся там текстом). Предположим, что вначале файлы A и B не были открыты и что в конце их надо закрыть, восстановив при этом исходные активные файлы ввода и вывода. Тогда решение этой задачи можно записать в виде следующего блока:

```
[PROG (АФВВ АФВЫВ)
    [SET АФВВ [ACTIVE GET]]
    [SET АФВЫВ [ACTIVE PUT]]
    [OPEN A GET] [ACTIVE A GET]
    [OPEN B ADD] [ACTIVE B PUT]
    [WHILE [NOT [EOF]] [PRINT [READ]]]
    [CLOSE A GET] [CLOSE B PUT]
    [COND (.АФВВ [ACTIVE .АФВВ GET])]
    [COND (.АФВЫВ [ACTIVE .АФВЫВ PUT])]]
```

В заключение отметим одну особенность плэнерского ввода. Как и в языке лисп, в плэнере допускается, чтобы программа и данные для нее находились в одном и том же файле. Например, при работе с плэнер-системой в пакетном режиме программа и данные обычно образуют единый набор выражений, расположенный на перфокартах **). В таких случаях транслятор языка и функции ввода считывают эти выражения вперемежку. Если, к примеру, при работе в пакетном режиме (когда активный файл ввода есть CARDS) на перфокартах были записаны выражения

```
[CSET PI [READ]]
3.14159
:PI
```

---

*) Точнее, если в файле есть хотя бы один символ, отличный от пробела. Эта оговорка необходима для случая, когда (по ошибке) оставшийся в файле текст не образует правильного плэнерского выражения.

**) В диалоговом режиме программа и данные, как правило, находятся в разных файлах.

то считывание их происходит так. Первое выражение всегда рассматривается как часть программы, поэтому его считывает и вычисляет транслятор. При вычислении этого выражения происходит обращение к функции ввода. Поскольку она настроена на чтение с перфокарт, она вводит первое из еще не считанных выражение, т. е. число 3.14159, которое и объявляет своим значением. Закончив вычисление первого выражения и напечатав его значение, транслятор считывает новое выражение — то, которое сейчас первое на входе. В нашем примере им является выражение :PI, его-то транслятор считывает и вычисляет.

1.16. Специальные функции

В этом параграфе рассматривается несколько встроенных функций, полезных при составлении сложных программ.

Иногда требуется прекратить вычисление некоторой функции, не дожидаясь обычного окончания ее работы. Например, довольно часто приходится выходить из «середины» блочной функции PROG, для чего, как известно, применяется функция RETURN. Однако с ее помощью нельзя досрочно прекратить вычисление других функций, поэтому в язык встроена более общая функция EXIT, которая позволяет завершить вычисление любой объемлющей функции.

Функция EXIT: [EXIT *e fn n*?], SUBR.

Если третий аргумент не задан, функция EXIT полностью завершает вычисление ближайшей объемлющей функции с именем *FN* (это значение аргумента *fn*) и объявляет ее значением выражение *E*. Если нужно осуществить выход не из ближайшей функции *FN*, а из второй, третьей или какой-то иной по счету объемлющей функции *FN*, то при обращении к EXIT следует задать третий аргумент, значением которого может быть любое неотрицательное целое число *N*. В таком случае произойдет выход из $(N+1)$-й по счету, считая от EXIT, объемлющей функции с именем *FN*, т. е. сначала будет «закрыто» *N* ближайших функций *FN*, а затем будет выполнено [EXIT *E FN*].

Если среди функций, объемлющих EXIT, нет нужной по счету функции *FN* или если значением аргумента *fn* не является идентификатор (как это имеет место, например, при обращении [EXIT ( ) ( )]), тогда выход осуществляется на верхний уровень программы. Тем самым полностью завершается вычисление того выражения верхнего уровня программы, в котором было данное обращение к функции EXIT, и его значением объявляется значение первого аргумента функции EXIT.

Рассмотрим для примера определение функции MN2, которая проверяет, есть ли среди элементов первого (верхнего) или второго уровней списка L хотя бы одно число:

```
[DEFINE MN2 (LAMBDA (L)
    [LOOP E1 .L [COND
        ([NUM .E1] [EXIT T LOOP])
        ([OR [ATOMIC .E1] [EMPTY .E1]] ())
        ([LOOP E2 .E1 [COND
            ([NUM .E2] [EXIT T LOOP 1])]])]])]
```

В этой функции организован цикл по элементам верхнего уровня списка L. Если среди них встретится число, то цикл сразу же прекращается: с помощью функции EXIT осуществляется выход со значением T из ближайшей функции LOOP, т. е. из тела функции MN2. Подсписки списка L анализируются с помощью внутреннего цикла. Если среди элементов этих подсписков встречается число, тогда вычисляется [EXIT T LOOP 1], т. е. прекращается работа сразу двух функций LOOP, внутренней и внешней.

В этом примере с помощью функции EXIT осуществляется выход из функций, которые объемлют ее статически (текстуально). Но с помощью функции EXIT можно прекращать вычисление и тех процедур, которые объемлют ее динамически. Имена таких процедур, как правило, заранее неизвестны, однако узнать их можно с помощью следующей встроенной функции.

Функция TRACK: [TRACK].

Значением этой функции является «след» программы — список имен всех объемлющих процедур. В нем первым элементом указывается название процедуры, которая вызвала функцию TRACK, вторым элементом — название процедуры, вызвавшей эту процедуру, и т. д. Последний элемент списка — имя самой внешней объемлющей процедуры.

Например, если функции F и G определены следующим образом:

```
[DEFINE F (LAMBDA () [PRINT [TRACK]])]
[DEFINE G (LAMBDA () [F])]
```

то при вычислении [G] на печать будет выдан список (PRINT F G), а при вычислении [PROG () [F]] — список (PRINT F PROG).

Следующей мы рассмотрим функцию, с помощью которой можно «ловить» ошибки в программе, соответствующим образом реагировать на них и затем продолжать вычисление программы. Такая возможность полезна, когда нежелательна стандартная реакция на ошибки — прекращение счета программы.

Функция CATCH: [CATCH e_1 e_2], FSUBR.

Данная функция вычисляет простую форму e_1 и, если это вычисление безошибочно, заканчивает свою работу со значением этой формы. Но если при вычислении e_1 была обнаружена ошибка, тогда данное вычисление прекращается и управление возвращается функции CATCH, которая в этом случае вычисляет «реакцию на ошибки» — простую форму e_2, а затем с ее значением заканчивает свою работу. (Отметим, что ошибки при вычислении e_2 данная функция CATCH уже не «ловит», их может «поймать» только объемлющая функция CATCH, если, конечно, такая есть.)

Например, следующая функция:

```
[DEFINE REST1 (LAMBDA (N L)
    [CATCH [REST .N .L] ()])]
```

действует аналогично встроенной функции REST, однако в том случае, когда число N по абсолютной величине превосходит длину списка L, она выдает в качестве своего значения пустой список ( ), тогда как REST сигнализирует об ошибке.

Часто, для того чтобы правильно прореагировать на ошибку, необходимо иметь информацию об ошибке. Такие сведения позволяет получить

Функция ERRINF: [ERRINF].

Значением этой функции является список вида (НОМОШ ПЛВЫР ИМЯ), где НОМОШ — номер последней ошибки в программе (каждому возможному типу ошибок поставлен в соответствие некоторый номер), ПЛВЫР — «плохое» выражение, из-за которого возникла ошибка, а ИМЯ — название той процедуры, при выполнении которой проявилась ошибка.

Например, если ошибке «аргумент — не идентификатор» поставлен в соответствие номер 4, то при вычислении выражения

```
[CATCH [DO [SET X (A)] [ADD1 .X]]
       [DO [PRINT [ERRINF]] [GO M]]]
```

будет напечатан список (4 (A) ADD1) и будет осуществлен переход по метке M.

1.17. Списки свойств

Довольно часто с идентификаторами приходится связывать некоторую информацию, которая характеризует объекты, обозначенные этими идентификаторами. Например, в программах, обрабатывающих тексты на естественном языке, приходится с каждым идентификатором, обозначающим слово естественного языка, свя-

зывать синтаксические и семантические характеристики этого слова, например: грамматический класс слова, род, одушевленность, родовое понятие и т. п. Такие характеристики обычно задаются в виде пар «название характеристики — значение характеристики». Например, для слова «стол» они могут быть такими:

грамматический класс — существительное,
род — мужской,
одушевленность — нет,
родовое понятие — мебель

и т. д.

Организовать хранение такой информации и реализовать соответствующие операции можно с помощью уже рассмотренных нами средств языка. Например, можно построить единый список, в котором для каждого идентификатора отводится один элемент, представляющий собой список, первый элемент которого — сам идентификатор, а остальные — связанные с ним характеристики и их значения. Для такого списка можно определить функции, которые позволяют находить значение нужной характеристики у заданного идентификатора, менять это значение и т. д.

Однако организация подобного списка и реализация соответствующих функций — дело хлопотное и не очень эффективное (например, замена в большом списке какого-то элемента на глубоких уровнях требует значительных затрат времени). Связывать же характеристики с идентификаторами приходится во многих программах. Поэтому в плэнер встроен аппарат так называемых списков свойств, который повышает эффективность хранения и доступа к подобной информации.

В плэнерских программах разрешается связывать с каждым идентификатором *список свойств*, который представляет собой совокупность *свойств*. Каждое свойство — это пара: *название* (*индикатор*) свойства и *значение* свойства. Названиями свойств могут быть только идентификаторы, а значениями — любые выражения. Например, названиями могут быть идентификаторы ЦВЕТ, ВЕС, КООРДИНАТЫ, а значениями — выражения БЕЛЫЙ, 50, (2 6 1).

В каждый момент с любым идентификатором программы связано только конечное число свойств, однако можно считать, что у идентификатора есть свойства с любыми названиями. Дело в том, что в языке принято следующее соглашение: отсутствие свойства эквивалентно наличию этого свойства со значением ( ). В частности, для уничтожения свойства достаточно дать ему значение ( ).

Для работы со списками свойств в языке имеется ряд встроенных функций. При их описании, а также в дальнейшем изложении мы будем пользоваться следующими обозначениями: через *ind* будем обозначать простую форму, значением которой должен

быть идентификатор (обозначаемый *IND*), играющий роль названия свойства, а через *v* — простую форму с любым значением, обозначаемым *V* и используемым в качестве значения свойства.

Для задания начального списка свойств идентификатора или полной замены его прежнего списка свойств на новый используется

Функция PLIST: [PLIST *i l*], SUBR.

Значением второго аргумента этой функции должен быть список вида $(IND_1\ V_1\ IND_2\ V_2\ \ldots\ IND_k\ V_k)$, $k \geqslant 0$, где IND_i — идентификаторы, а V_i — любые выражения. Функция PLIST уничтожает прежний список свойств идентификатора *I*, являющегося значением первого аргумента, и связывает с идентификатором *k* новых свойств с названиями IND_i и значениями V_i. Значение функции равно значению ее второго аргумента.

Например, после вычисления

```
[PLIST HEAD ( )]
```

у идентификатора HEAD не останется ни одного свойства, а после вычисления

```
[PLIST АТОМ (РОД ФУНКЦИЯ КЛАСС SUBR
    ЧИСЛОАРГ 1 ТИПАРГ (ЛЮБОЕ ВЫРАЖЕНИЕ))]
```

с идентификатором АТОМ будет связано четыре свойства: одно — с названием РОД и значением ФУНКЦИЯ, другое — с названием КЛАСС и значением SUBR, третье — с названием ЧИСЛОАРГ и значением 1, четвертое — с названием ТИПАРГ и значением (ЛЮБОЕ ВЫРАЖЕНИЕ).

Доступ ко всему списку свойств идентификатора или к его отдельным свойствам обеспечивает

Функция GET: [GET *i ind*?], SUBR.

Если второго аргумента нет, то в качестве своего значения функция выдает весь список свойств идентификатора *I* в виде списка $(IND_1\ V_1\ \ldots\ IND_k\ V_k)$, $k \geqslant 0$. При этом свойства со значением ( ) в данный список не включаются, а порядок расположения свойств произволен. В том случае, когда заданы оба аргумента, функция выдает значение только одного свойства — с названием *IND*; в частности, будет выдано ( ), если свойства с таким названием нет.

Если, например, идентификатор АТОМ обладает указанными выше свойствами, то

```
[GET АТОМ]  →  (ТИПАРГ (ЛЮБОЕ ВЫРАЖЕНИЕ)
    КЛАСС SUBR РОД ФУНКЦИЯ ЧИСЛОАРГ 1)
[GET АТОМ КЛАСС]  →  SUBR
[GET АТОМ ТИПРЕЗ]  →  ( )
```

Для изменения значения какого-нибудь одного свойства идентификатора применяется

Функция PUT: [PUT *i ind v*], SUBR.

Эта функция заменяет в списке свойств идентификатора *I* прежнее значение свойства с названием *IND* на новое значение, которым является выражение *V*. Если такого свойства не было, то оно добавляется, а если *V* — это пустой список ( ), то данное свойство уничтожается. Значение функции равно *V*.

Например, при указанных выше свойствах идентификатора АТОМ после вычисления

```
[DO [PUT АТОМ РОД СОПОСТАВИТЕЛЬ]
    [PUT АТОМ КЛАСС ( )]
    [PUT АТОМ ЧИСЛОАРГ 0]
    [PUT АТОМ ТИПАРГ ( )]]
```

у индентификатора АТОМ окажется только два свойства: одно — с названием РОД и значением СОПОСТАВИТЕЛЬ, а другое — с названием ЧИСЛОАРГ и значением 0.

1.18. Преобразование типов данных

До сих пор мы считали, что атомарные выражения являются неделимыми конструкциями. Для большинства задач такая точка зрения наиболее естественна и удобна. Однако в некоторых случаях выгодно все же рассматривать эти конструкции как составные. Поэтому в язык встроен ряд функций, которые не считают атомарные выражения неделимыми объектами и которые позволяют преобразовать одни атомарные выражения в другие.

Функция ATL: [ATL *i*], SUBR.

Эта функция разбивает идентификатор, являющийся значением ее аргумента, на отдельные символы, из которых состоит печатное наименование идентификатора, и превращает их в односимвольные атомы: цифры — в целые числа, остальные символы — в идентификаторы. L-список из этих атомов объявляется значением функции. Например:

```
[ATL СЛОВОМ]  →  (С Л О В О М)
[ATL B5+C.]  →  (B 5 + C .)
```

Функция ATL полезна, например, в программах, обрабатывающих тексты на естественном языке. В этих программах приходится определять, в каком падеже и числе входит в исходный текст каждое существительное, прилагательное и т. п. Если бы идентификаторы, представляющие эти слова, нельзя было разделить на составляющие их буквы, тогда в программе пришлось бы хранить формы каждого слова во всех падежах единственного и множественного чисел. Функция же ATL позволяет преобразовывать любое слово-идентификатор в список из его букв, который уже можно анализировать (например, находить основу и окончание слова) средствами языка. Поэтому в программе можно хранить только основы слов и общую для всех слов таблицу окончаний.

Если функция ATL полезна при анализе слов, то следующая функция полезна при их синтезе, например при соединении основы слова с окончанием.

Функция LTA: [LTA *l*], SUBR.

Значением аргумента должен быть непустой список, элементами которого являются атомы. Функция «склеивает» эти атомы в один идентификатор, который она и выдает как свое значение. Примеры:

```
[LTA (С Л О В О М)]  →  СЛОВОМ
[LTA (СЛОВ ОМ)]     →  СЛОВОМ
[LTA (—52 А *7)]    →  —52А*7
[LTA (* Х 1)]       →  *Х1
[LTA (5 6.79)]      →  56.79000
```

(замечание: у вещественных чисел учитывается столько цифр из дробной части, сколько их в текущий момент выводится на печать — см. § 1.15).

Подчеркнем, что значением функции LTA всегда является идентификатор, даже если он внешне имеет вид запрещенного атома (как в третьем примере выше) или похож на другие типы выражений (как в последних двух примерах).

Рассмотрим одно из возможных применений функции LTA. В некоторых программах возникает потребность во «внутренних» названиях для их объектов, причем заранее количество таких имен неизвестно, так что запасти их сразу нельзя. Во многих версиях языка лисп в подобных ситуациях предлагается встроенная функция GENSYM, которая при каждом обращении к ней выдает в качестве своего значения некоторый новый идентификатор, отличный от всех остальных идентификаторов программы. В плэнере такой встроенной функции нет, но она легко определяется с помощью функции LTA. Для этого можно, например, ввести кон-

станту с именем NGENSYM, дать ей начальное значение 0 и затем определить функцию GENSYM следующим образом:

```
[DEFINE GENSYM (LAMBDA ()
    [DO [CSET NGENSYM [+ :NGENSYM 1]]
        [LTA (:NGENSYM *)]])]
```

Обращение к этой функции имеет вид [GENSYM]. При первом обращении функция выдает в качестве своего значения идентификатор 1*, при втором — идентификатор 2*, при сорок пятом — 45* и т. п.

Следующие две функции преобразуют шкалы в целые числа и наоборот.

Функция STN: [STN *s*], SUBR.

Эта функция преобразует шкалу *S*, рассматриваемую как запись целого числа в восьмеричной системе, в соответствующее десятичное целое число. Например:

[STN *21] → 17

Функция NTS: [NTS *n*], SUBR.

Значением аргумента данной функции должно быть неотрицательное целое число. Оно преобразуется в шкалу, представляющую запись этого числа в восьмеричной системе. Например:

[NTS 17] → *21

С помощью следующей встроенной функции можно изменять внешние скобки у любых списков, можно заменять в обращениях к переменным одни префиксы на другие, вообще отбрасывать префиксы или присоединять префиксы к именам переменных.

Функция ETE: [ETE e_1 e_2], SUBR.

Значениями обоих аргументов этой функции должны быть одновременно либо списки, либо идентификаторы и/или обращения к переменным. В первом случае значением функции является список, составленный из элементов списка E_1, но имеющий скобки списка E_2, т. е. у списка E_1 скобки заменяются на скобки списка E_2. Во втором случае префикс выражения E_1 заменяется на префикс выражения E_2. Более точно, в качестве своего значения функция выдает обращение к переменной, составленное из имени, взятого из E_1, и префикса, взятого из E_2 (идентификаторы здесь условно рассматриваются как обращения к переменным с пустым префиксом). Примеры:

```
[ETE (A (B C)) [QUOTE < >]]  →  <A (B C)>
[ETE [QUOTE *X] [QUOTE !.A]]  →  !.X
[ETE [QUOTE .X] A]  →  X
[ETE X [QUOTE :A]]  →  :X
```

Отметим, что функция ETE осуществляет чисто синтаксическое преобразование, поэтому не требуется, чтобы переменные, обращения к которым указаны в качестве аргументов, были описаны в программе.

Функция ETE, как правило, применяется тогда, когда во время выполнения программы надо построить плэнерские выражения, которые затем будут вычислены с помощью функции EVAL. Здесь обычно возникает такая проблема: как, имея имя некоторой переменной, построить обращение к ней с некоторым префиксом? Функция LTA в данной ситуации не поможет, так как в результате вычисления, скажем, выражения [LTA (. X)] получается конструкция .X, которая внешне похожа на .-переменную, но которая ею не является. А вот функция ETE здесь как раз и пригодится:

[ETE X [QUOTE .C]] → .X

причем выражение .X действительно является .-переменной.

1.19. Пример программы

Рассмотрим задачу преобразования выражений языка лисп в эквивалентные плэнерские выражения с последующим их вычислением. Например, лисповское выражение

```
(SEXPR MEMBER (LAMBDA (A L)
    (COND ((EQ L ( )) NIL)
          ((EQ A (CAR L)) T)
          (T  (MEMBER A (CDR L))))))
```

должно быть преобразовано в плэнерское выражение

```
[DEFINE MEMBER (LAMBDA (A L)
    [COND ([EQ .L ( )] ( ))
          ([EQ .A [1 .L]] T)
          (T [MEMBER .A [REST 1 .L]])])]
```

которое и следует вычислить.

Для простоты мы ограничимся «базовым» лиспом и будем предполагать, что лисповские выражения записаны без ошибок, что точечные выражения лиспа не используются и что встроенные лисповские функции переопределять нельзя.

Ниже перечислены правила преобразования допускаемых нами лисповских выражений (слева) в эквивалентные плэнерские выражения; в этих правилах $\tilde{x}$ обозначает простую плэнерскую форму, эквивалентную лисповскому выражению x, а $\tilde{\tilde{x}}$ — сегментный вариант этой формы.

число → *число*
NIL, ( ) → ( )
T → T
другие идентификаторы → *.-переменные*
(QUOTE x) → x
(CAR x) → [1 $\widetilde{x}$]
(CDR x) → [REST 1 $\widetilde{x}$]
(CONS x y) → ($\widetilde{x}$ $\widetilde{y}$)
(ATOM x) → [ATOM $\widetilde{x}$]
(EQ x y) → [EQ $\widetilde{x}$ $\widetilde{y}$]
(COND (p_1 e_1) ... (p_k e_k)) →
→ [COND ($\widetilde{p_1}$ $\widetilde{e_1}$) ... ($\widetilde{p_k}$ $\widetilde{e_k}$)]
(SEXPR f (LAMBDA (v_1 v_2 ... v_n) e)) →
→ [DEFINE f (LAMBDA (v_1 v_2 ... v_n) $\widetilde{e}$)]
(f a_1 a_2 ... a_n) → [f $\widetilde{a_1}$ $\widetilde{a_2}$ ... $\widetilde{a_n}$]
((LAMBDA (v_1 v_2 ... v_n) e) a_1 a_2 ... a_n) →
→ [DO [DEFINE f (LAMBDA (v_1 v_2 ... v_n) $\widetilde{e}$)]
[f $\widetilde{a_1}$ $\widetilde{a_2}$ ... $\widetilde{a_n}$]]

Относительно последнего правила преобразования следует сделать замечание. В лиспе при обращении к функции разрешается вместо ее имени записывать определяющее выражение этой функции. В плэнере так делать нельзя, поэтому подобное лисповское обращение заменяется на два действия — на определение функции с данным определяющим выражением и с каким-нибудь именем и на последующее обращение к этой функции по имени.

Указанные правила преобразования мы кладем в основу следующей плэнерской программы, которая решает поставленную задачу:

```
[DEFINE COMP (LAMBDA (L)
    [COND ([ATOM .L] [COMPATOM .L])
          ([EQ .L ( )] ( ))
          (T [COMPLIST .L])])]
```

Эта функция осуществляет преобразование одного лисповского выражения L: если L — атом, тогда к L применяется функция COMPATOM, если L — непустой список, то применяется функция COMPLIST, а если L — пустой список, тогда эквивалентное плэнерское выражение определяется сразу.

```
[DEFINE COMPATOM (LAMBDA (A)
    [COND ([NUM .A] .A)
          ([EQ .A T] T)
```

```
        ([EQ .A NIL] ())
        (T [ETE .A [QUOTE .C]])])]
```

Функция COMPATOM преобразует лисповские атомы в эквивалентные плэнерские выражения: числа и атом T не меняются, атом NIL, который в лиспе эквивалентен пустому списку, преобразуется в (), а все остальные атомы (в лиспе это переменные) преобразуются в плэнерские .-переменные.

```
[DEFINE COMPLIST (LAMBDA (L) [PROG (F)
    [FIN F L]
    [COND ([EQ .F QUOTE] [1 .L])
          ([EQ .F CAR]
             [FORM [1 [COMP [1 .L]]]])
          ([EQ .F CDR]
             [FORM [REST 1 [COMP [1 .L]]]])
          ([EQ .F CONS]
             [COMPCONS [1 .L] [2 .L]])
          ([EQ .F COND] [COMPCOND .L])
          ([EQ .F SEXPR]
             [COMPDEFUN [1 .L] [2 .L]])
          ([LIST .F] [COMPLAMBDA .F .L])
          (T [FORM [.F ‹COMPARGS .L›]])])]]
```

Функция COMPLIST преобразует непустые списки, которые представляют собой обращения к лисповским функциям. COMPLIST выделяет все особые случаи преобразования и для них либо сама, либо с помощью других функций программы строит эквивалентные плэнерские выражения. Последняя клауза условного выражения соответствует общему случаю преобразования, при котором имя лисповской функции не меняется, а последовательность ее аргументов заменяется на последовательность эквивалентных плэнерских выражений. Так преобразуются обращения к лисповским встроенным функциям ATOM и EQ, а также обращения ко всем определяемым функциям.

Преобразование списка из лисповских выражений в список из соответствующих плэнерских выражений осуществляется функцией COMPARGS:

```
[DEFINE COMPARGS (LAMBDA (L) [COND
    ([EMPTY .L] ())
    (T ([COMP [1 .L]]
          ‹COMPARGS [REST 1 .L]›))])]
```

Следующие четыре функции нашей программы осуществляют трансляцию конкретных лисповских функций.

```
[DEFINE COMPCONS (LAMBDA (A1 A2) [DO
    [SET A1 [COMP .A1]]
    [SET A2 [COMP .A2]]
    [COND ([VAR. .A2]
              (.A1 [ETE .A2 [QUOTE !.C]]))
          ([LISTP .A2]
              (.A1 [ETE .A2 [QUOTE < >]]))
          ([LIST .A2] (.A1 !.A2))]])]
```

Эта функция преобразует обращение к лисповской функции CONS. COMPCONS прежде всего получает плэнерские эквиваленты A1 и A2 обоих аргументов функции CONS, а затем строит L-список, первый элемент которого — это A1, а второй (или остальные) получается так. Если A2 — это .-переменная или простое обращение к функции, то в L-список вставляется сегментный вариант этого обращения. Если же A2 является списком в круглых скобках, то в L-список вставляются сразу все элементы списка A2, т. е. результат вычисления лисповской функции CONS здесь выписывается явно. Возможен еще один вариант, когда A2 является атомом (например, числом или T). В таком случае в лиспе образуется точечное выражение, но поскольку мы их не рассматриваем, то данный вариант игнорируется.

```
[DEFINE COMPCOND (LAMBDA (L) [PROG ((CL ()))
    [LOOP E .L [SET CL (!.CL [COMPARGS .E])]]
    [FORM [COND !.CL]]])]
```

Функция COMPCOND транслирует лисповские условные выражения, т. е. обращения к функции COND.

```
[DEFINE COMPDEFUN (LAMBDA (F D)
    [FORM [DEFINE .F
        (LAMBDA [2 .D] [COMP [3 .D]])]])]
```

Функция COMPDEFUN преобразует обращение к лисповской функции SEXPR, которая определяет новую функцию с именем F и определяющим выражением D, в соответствующее обращение к плэнерской функции DEFINE.

И, наконец, следующая функция транслирует лисповский список, первым элементом которого является LAMBDA-выражение:

```
[DEFINE COMPLAMBDA (LAMBDA (D A) [PROG (F)
    [SET F [GENSYM]]
    [FORM [DO [COMPDEFUN .F .D]
              [FORM [.F <COMPARGS .A>]]]]])]
```

В этой функции используется обращение к функции GENSYM (см. § 1.18), которая «поставляет» некоторое имя для функции

с заданным определяющим выражением. Поскольку GENSYM не относится к числу встроенных функций плэнера, она также должна быть описана в нашей программе:

```
[CSET NGENSYM 0]
[DEFINE GENSYM (LAMBDA ( )
    [DO [CSET NGENSYM [+ :NGENSYM 1]]
        [LTA (:NGENSYM *)]])]
```

Итак, мы определили все необходимые функции, с помощью которых осуществляется трансляция одного лисповского выражения. Если мы хотим отранслировать целую лисповскую программу (последовательность выражений), расположенную в некотором файле FILE, и хотим вычислить ее с распечаткой результатов на АЦПУ, то в нашу программу следует включить определение еще одной, основной функции:

```
[DEFINE COMPILE (LAMBDA (FILE) [PROG (L)
    [OPEN .FILE GET] [ACTIVE .FILE GET]
    [ACTIVE PAPER PUT]
    [WHILE [NOT [EOF]]
           [PRINT [EVAL [COMP [READ]]]]]
    [CLOSE .FILE GET]])]
```

Последним выражением нашей плэнерской программы должно быть обращение к функции COMPILE, аргументом для которой указывается имя файла (скажем, ЛИСП), в котором реально записана лисповская программа:

```
[COMPILE ЛИСП]
```

Вычисление этого выражения и вызовет выполнение описанной выше плэнерской программы, т. е. трансляцию и счет лисповской программы.

ГЛАВА 2

ОБРАЗЦЫ

2.1. Основные понятия

Обработка данных подразумевает, как правило, анализ этих данных и их преобразование. И то и другое в плэнере можно осуществлять с помощью функций, рассмотренных в предыдущей главе. Однако для проведения анализа данных в языке есть более удобное средство — *образцы*. Достоинством образцов является то, что они в более наглядной и более лаконичной форме, чем функции, описывают правила анализа. Поэтому программы, в которых используются образцы, получаются более простыми и компактными, чем программы, в которых используются только функции.

Синтаксически образец — это плэнерское выражение, а содержательно его можно рассматривать как шаблон, который как бы накладывается на анализируемое выражение, чтобы определить, имеет ли оно требуемую структуру. Процесс такого наложения называется *сопоставлением* образца с выражением. Возможны два исхода сопоставления: *удача* и *неудача*. Сопоставление удачно, если анализируемое выражение имеет требуемую структуру; в этом случае говорят, что образец и выражение *соответствуют* друг другу. В противном случае сопоставление неудачно, выражение не соответствует образцу.

При сопоставлении образца с выражением возможны побочные эффекты: некоторым переменным из образца могут быть присвоены новые значения. Не требуется, чтобы образец полностью определял структуру анализируемого выражения. Образец может накладывать ограничения только на некоторые части выражения, другие же фрагменты выражения могут быть произвольными. В то же время очень часто необходимо знать конкретный вид таких фрагментов. Для этого в соответствующие позиции образца вставляются переменные, которым при сопоставлении и будут присвоены интересующие нас фрагменты выражения. Таким образом, с помощью образцов можно не только убедиться,

что анализируемое выражение имеет определенную структуру, но и узнать заранее неизвестные его части.

Образцы делятся на *простые* и *сегментные*. Простые образцы всегда сопоставляются с одним выражением. К таким образцам относятся:

— атомы;
— простые обращения к константам и переменным;
— простые обращения к функциям;
— L-списки, элементами которых являются (любые) образцы;
— простые обращения к процедурам, называемым сопоставителями.

Сегментные образцы всегда сопоставляются с последовательностями выражений (сегментами). Самостоятельно эти образцы не используются и могут встречаться лишь как элементы образцов-списков или как аргументы функций и сопоставителей. К сегментным образцам относятся:

— сегментные обращения к константам и переменным;
— сегментные обращения к функциям;
— сегментные обращения к сопоставителям.

Никакие другие плэнерские выражения использовать в качестве образцов нельзя.

В дальнейшем мы будем обозначать образцы через *pat* (возможно, с индексами: pat_1, pat_i и т. п.) и будем, если не сказано иное, под «образцом» понимать «простой образец».

Сопоставление образца с выражением осуществляет

Функция IS: [IS *pat e*], FSUBR.

Свой первый аргумент — простой образец *pat* — функция IS не вычисляет. Она вычисляет только второй аргумент — простую форму *e*, значением которой может быть любое выражение. Данное выражение и сопоставляется с образцом *pat*. Если сопоставление удачно, функция вырабатывает значение T, неудачно — значение ( ).

2.2. Простые образцы

В этом параграфе описываются правила сопоставления простых образцов всех типов, кроме образцов-сопоставителей, которые будут рассмотрены позже.

Образцам-атомам соответствуют только равные им атомы. Поэтому

```
[IS ПЛЭНЕР ПЛЕНЭР]  →  ( )
[IS 5 [+ 2 3]]  →  T
```

Если в качестве образца указано простое обращение к функции, то вычисляется значение функции, и оно сравнивается с со-

поставляемым выражением. В случае равенства сопоставление удачно, иначе — неудачно. Например:

```
[IS [ATL СЕЗАМ] (С Е З А М)]  →  Т
[IS [+ 2 3] 5.0]  →  ()
```

Образцу, являющемуся простым обращением к константе, соответствует только выражение, равное значению константы. Пусть, к примеру, константа PI имеет значение 3.14159, тогда

```
[IS :PI 3.14159]  →  Т
[IS :PI (A)]  →  ()
```

Если в качестве образца указана .-переменная и эта переменная имеет значение, то такому образцу соответствует только выражение, совпадающее с текущим значением переменной. Например:

```
[DO [SET X (A B)] [IS .X (A B)]]  →  Т
[DO [SET A *20] [IS .A 16]]  →  ()
```

Во всех рассмотренных выше случаях сопоставление фактически сводилось к проверке на равенство. В следующих же двух случаях сопоставление сводится к присваиванию переменным новых значений.

Если в качестве образца указана .-переменная и эта переменная в текущий момент не имеет значения, то данный образец соответствует любому выражению. Как побочный эффект сопоставления, переменная получает значение, которым становится сопоставляемое с нею выражение. Например, если переменная X не имела значения, то при сопоставлении

```
[IS .X (A + B)]  →  Т
```

переменной X будет присвоено значение (A + B).

Таким образом, как образец .-переменная ведет себя двояко. Если она имеет значение, то оно и используется при сопоставлении, а если переменная не имеет значение, то при сопоставлении ей присваивается значение.

Довольно часто в процессе сопоставления необходимо присваивать переменным новые значения независимо от того, имели они ранее значения или нет. В таких случаях следует обращаться к переменным с префиксом «*». Как образец *-переменная соответствует любому выражению, причем при сопоставлении переменная получает новое значение, которым становится сопоставляемое с нею выражение. Так, после вычисления

```
[DO [SET X PL/1]
    [IS *X (PASCAL ADA)]]  →  Т
```

переменная X будет иметь значение (PASCAL ADA).

Все рассмотренные нами случаи сопоставления очень просты, и их легко реализовать с помощью функций. Поэтому рассмотренные выше образцы обычно используются не самостоятельно, а как компоненты более сложных образцов — образцов-списков.

Образцу, являющемуся L-списком, соответствует только L-список, элементы которого соответствуют элементам данного образца. При этом требуется, чтобы существовало соответствие для каждого элемента как образца-списка, так и анализируемого списка. Например, сопоставление

[IS (ОГНИ *X) (ОГНИ МОСКВЫ)]

удачно, поскольку есть соответствие для каждого элемента сопоставляемых списков, а сопоставление

[IS (ОГНИ *X) (ОГНИ НОЧНОЙ МОСКВЫ)]

неудачно, так как нет соответствия для третьего элемента анализируемого списка (напомним, что простому образцу, в частности *X, может соответствовать одно выражение, но не сегмент).

При описании точных правил сопоставления L-списков будем различать два случая: когда все элементы образца-списка являются простыми образцами и когда среди этих элементов есть хотя бы один сегментный образец. В данном параграфе рассматривается только первый случай.

Образец-список, составленный из одних простых образцов, требует, чтобы сопоставляемый с ним список имел ту же длину, что и он сам, и чтобы элементы обоих списков, находящиеся в одинаковых позициях, соответствовали друг другу. Более точно, сопоставление L-списков одинаковой длины сводится к последовательному сопоставлению сначала первых элементов списков, затем вторых элементов, затем третьих элементов и т. д. Если все эти сопоставления удачны, то удачным считается и сопоставление списков в целом; при этом все побочные эффекты, имевшие место при проверке соответствия элементов, сохраняют свою силу и по окончании сопоставления списков. Но если окажется, что сопоставление какой-то пары элементов неудачно, то сопоставление списков сразу же прекращается; оно считается неудачным, и при этом все побочные эффекты сопоставления предыдущих элементов уничтожаются.

Таким образом, при сопоставлении переменные образца получают новые значения только тогда, когда сопоставление образца с выражением удачно в целом, при неудачном же сопоставлении переменные образца не меняют своих значений. Это — общее правило сопоставления образцов.

Проиллюстрируем сказанное выше конкретными примерами. Сначала рассмотрим случаи неудачного сопоставления:

```
[IS (A + .X) [QUOTE ⟨A + B⟩]]  →  ()
[IS (*X — СТОЛИЦА ФИДЖИ)
    (СУВА — СТОЛИЦА)]  →  ()
[IS (ЯЗЫК АДА) (ЯЗЫК АДЫ)]  →  ()
```

В первом из этих примеров неудача сопоставления обусловлена тем, что анализируемое выражение не является списком в круглых скобках, во втором примере — разной длиной сопоставляемых списков, а в третьем примере — несоответствием вторых элементов списков.

В следующих примерах, где предполагается, что переменная Y имеет значение 1828, а константа С — значение ФИДЖИ, все требования к удачному исходу сопоставления выполнены:

```
[IS (.Y — [+ .Y 82]) (1828 — 1910)]  →  T
[IS (*X ГОВОРИЛ: *Y)
    (К.ПРУТКОВ ГОВОРИЛ: БДИ!)]  →  T
```

и переменная X получает значение К.ПРУТКОВ, а переменная Y — значение БДИ!

```
[IS (:C НАСЕЛЯЮТ (*N ТЫСЯЧ))
    (ФИДЖИ НАСЕЛЯЮТ (600 ТЫСЯЧ))]  →
T,  N:=600
```

Рассмотрим процесс сопоставления списков в последнем случае. Оба сопоставляемых списка имеют одинаковую длину, поэтому функция IS осуществляет последовательное сопоставление их элементов:

```
[IS :C ФИДЖИ]  →  T
[IS НАСЕЛЯЮТ НАСЕЛЯЮТ]  →  T
[IS (*N ТЫСЯЧ) (600 ТЫСЯЧ)]  →  T,  N:= 600
```

(в последнем случае рекурсивно применяется правило сопоставления списков). Поскольку все эти сопоставления удачны, то удачным является и сопоставление списков в целом. А раз так, то за переменной N сохраняется значение 600, полученное ею во время сопоставления третьих элементов исходных списков.

Другой пример. Пусть переменная Y имеет значение ФУТЛЯР. Тогда сопоставление

```
[IS (*X — .Y РОЯЛЯ) (АРФА — СКЕЛЕТ РОЯЛЯ)]
```

неудачно, поэтому переменная X не меняет своего значения. В самом деле, это сопоставление сводится к последовательному выполнению следующих сопоставлений:

```
[IS *X АРФА]  →  T,  X:=АРФА
[IS — —]  →  T
[IS .Y СКЕЛЕТ]  →  ()
```

Поскольку обнаружилось несоответствие между третьими элементами списков, сопоставление списков прекращается и оно объявляется неудачным. При этом присвоенное переменной X значение АРФА уничтожается и восстанавливается прежнее значение переменной, которое она имела до начала сопоставления.

Изменение значений у переменных образца в процессе сопоставления можно рассматривать как условное (временное) присваивание значений, которые станут постоянными значениями только в случае удачного исхода всего сопоставления образца с выражением. Однако это не означает, что такими временными значениями нельзя пользоваться. Пока выполняется сопоставление, временные значения переменных рассматриваются как текущие значения, которыми можно воспользоваться, обратившись к переменным с префиксом «.». Такая возможность использована, например, в образце (*X .X), которому соответствует любой L-список из двух равных элементов. Действительно, рассмотрим сопоставление

```
[IS (*X .X) (FIFTY FIFTY)]
```

При сопоставлении первых элементов *X и FIFTY переменная X условно получает значение FIFTY, которое и используется при сопоставлении вторых элементов .X и FIFTY. Так как сопоставление списков удачно, значение FIFTY сохраняется за переменной X и по окончании сопоставления.

Манипулируя префиксами «.» и «*», можно построить более сложные примеры того же типа. Предположим, что до начала сопоставления переменная X имела значение 1; тогда имеем

```
[IS (.X + *X × .X − *X / [+ .X 1])
    (1 + 2 × 2 − 3 / 4)]   →   T,   X:=3
```

Здесь при сопоставлении первых элементов используется значение переменной X, равное 1. При сопоставлении третьих элементов переменная X получает новое значение 2, которое используется при сопоставлении пятых элементов. При сопоставлении же седьмых элементов переменная X еще раз меняет свое значение; им становится число 3, которое используется в функции сложения и которое сохраняется у переменной после сопоставления.

2.3. Сегментные образцы

Теперь мы рассмотрим правила сопоставления образцов-списков, среди элементов которых есть сегментные образцы.

Сегментные образцы, как уже отмечалось, сопоставляются с последовательностями выражений (сегментами). При этом сопоставление таких образцов регламентируется *правилом десегмен-*

тации, которое гласит, что сопоставление сегментного образца с сегментом всегда осуществляется так, как если бы выполнялось сопоставление соответствующего простого образца с L-списком, составленным из элементов данного сегмента.

Например, сегментный образец !:C соответствует сегменту $a_1\ a_2\ \ldots\ a_k$ тогда и только тогда, когда простой образец :C соответствует списку $(a_1\ a_2\ \ldots\ a_k)$, т. е. когда значение константы равно этому списку. Таким образом, если в образце-списке встретилось сегментное обращение к константе и ее значением является L-список из k элементов, то данный сегментный образец требует, чтобы k очередных элементов анализируемого списка образовывали последовательность, совпадающую с сегментированным значением константы. Только при этом условии будет существовать соответствие для данного сегментного образца. Если же в анализируемом списке осталось меньше k элементов или если сегмент из очередных k элементов этого списка не равен сегментированному значению константы, то соответствия данному сегментному образцу нет. Например, при значении (+ 2 +) у константы C имеем

```
[IS (1 !:C 3) (1 + 2 + 3)]  →  T
[IS (1 !:C 3) (1 + 2)]      →  ()
[IS (1 !:C 3) (1 — 2 + 3)]  →  ()
```

Следует подчеркнуть, что, согласно правилу десегментации, совпадения сегментированного значения константы с сегментом очередных элементов анализируемого списка недостаточно для удачи сопоставления. Требуется также, чтобы значением константы был L-список. Если же ее значением является атомарное выражение или список не в круглых скобках, то сегментному обращению к этой константе не соответствует ни один сегмент. В самом деле, такое значение не равно никакому L-списку, поэтому и простое обращение к константе не соответствует никакому L-списку, а раз так, то, согласно правилу десегментации, сопоставление сегментного обращения к константе с любым сегментом оказывается неудачным. Таким образом, если константа P имеет значение [+ 2 +], а константа Q — значение A, то

```
[IS (1 !:P 3) (1 + 2 +3)]  →  ()
[IS (!:Q B) (A B)]         →  ()
```

Аналогично сегментным обращениям к константам ведут себя !.-переменные, имеющие значения, и сегментные обращения к функциям. Здесь для удачного исхода сопоставления также требуется, чтобы значением переменной или функции был L-список и чтобы последовательность элементов этого списка совпадала с сопоставляемым сегментом. Например, если переменная X имеет

значение (КОЗЬМА ПРУТКОВ), а Y — значение (СМОТРИ В КОРЕНЬ), то

```
[IS (!.X ГОВОРИЛ: !.Y)
    (КОЗЬМА ПРУТКОВ ГОВОРИЛ:
    СМОТРИ В КОРЕНЬ)]  →  T
[IS (⟨REST 2 .Y⟩ *Z)
    (КОРЕНЬ УРАВНЕНИЯ)]  →  T, Z:= УРАВНЕНИЯ
[IS (⟨3 .Y⟩ *Z) (КОРЕНЬ УРАВНЕНИЯ)]  →  ()
```

Теперь рассмотрим, как сопоставляются !*-переменные и !.-переменные, не имеющие значений. Согласно правилу десегментации, сопоставление образца !*X с сегментом $a_1\ a_2\ \ldots\ a_k$ эквивалентно сопоставлению образца *X со списком $(a_1\ a_2\ \ldots\ a_k)$. Отсюда следует такое правило сопоставления !*-переменной или !.-переменной, не имеющей значения: этому сегментному образцу соответствует любой сегмент, причем при сопоставлении переменная получает новое значение, которым является L-список, составленный из элементов того сегмента, с которым сопоставлялся данный образец. Например, сопоставления

```
[DO [UNASSIGN X]
    [IS (ОГНИ !.X) (ОГНИ МОСКВЫ)]]
[IS (ОГНИ !*Y) (ОГНИ НОЧНОЙ МОСКВЫ)]
```

удачны, и при этом значением переменной X становится список (МОСКВЫ), а значением переменной Y — список (НОЧНОЙ МОСКВЫ).

Поскольку !*-переменной (как и !.-переменной, не имеющей значения) соответствует любой сегмент, то, естественно, возникает вопрос: а какой именно сегмент анализируемого списка ставится в соответствие этому сегментному образцу? Ответ такой: сегмент подбирается так, чтобы было соответствие и для других элементов сопоставляемых списков.

Если в образце-списке всего одна !*-переменная, то выбор сегмента не вызывает затруднений. Например, при сопоставлении

```
[IS (1 !*X*Y) (1 + 2 + 3 + 4)]
```

на долю образца !*X приходится сегмент + 2 + 3 +, так как только в этом случае есть соответствие и всем другим элементам образца-списка и анализируемого списка. Тем самым по окончании сопоставления переменная X будет иметь значение (+ 2 + 3 +), а переменная Y — значение 4.

Если !*-переменных в образце-списке несколько, то выбор сегментов усложняется, но и здесь он во многих случаях очевиден. Так, при сопоставлении

```
[IS (!*X — !*Y) (1 + 2 + 3 — 4)]
```

на долю образца !*X приходится сегмент 1 + 2 + 3, а на долю образца !*Y — сегмент из одного элемента 4. Только в этом случае второй элемент образца-списка будет иметь соответствие в анализируемом списке. Следовательно, после сопоставления значением переменной X станет список (1 + 2 + 3), а значением переменной Y — список (4). В то же время при сопоставлении

```
[IS (!*X — !*Y) (1 + 2 + 3 + 4)]
```

ни при каком распределении сегментов между образцами !*X и !*Y не будет соответствия второму элементу образца-списка, потому данное сопоставление неудачно.

Еще несколько примеров подобного типа:

```
[IS (!*X + !*Y + !*Z)
    (+ A — B + C)]   →   T,
    X:= ( ), Y:= (A — B), Z:= (C)
[IS (!*X *P !*Y .P !*Z)
    (1 — 2 + 3 + 4)]   →   T,
    X:= (1 — 2), P:= +, Y:= (3), Z:= (4)
[IS (*P !*X .P !*Y) (1 + 2 + 3)]   →   ( )
```

В ряде случаев возможно несколько вариантов удачного исхода сопоставления списков. Например, сопоставление

```
[IS (!*X + !*Y) (1 + 2 + 3 + 4)]
```

является удачным при любом из следующих вариантов распределения сегментов между образцами !*X и !*Y:

```
        1  и  2 + 3 + 4
    1 + 2  и  3 + 4
1 + 2 + 3  и  4
```

Какой из этих вариантов выбирается при сопоставлении? Это важно знать, поскольку от выбора варианта зависит, какие значения получат переменные X и Y.

В подобных ситуациях из всех возможных вариантов удачного распределения сегментов между !*-переменными (или !.-переменными, не имеющими значений) всегда выбирается тот, в котором на долю самой левой !*-переменной образца-списка приходится наикратчайший сегмент. Если после этого все равно осталось несколько вариантов, то из них выбирается тот, где на долю второй по порядку !*-переменной приходится наикратчайший сегмент. Если снова нет однозначности, тогда выбирается вариант с наикратчайшим сегментом для третьей по порядку !*-переменной, и т. д.

Согласно этому правилу, в нашем последнем примере выбирается вариант, в котором образцу !*X поставлен в соответ-

ствие сегмент из одного элемента 1, а образцу !*Y — сегмент 2 + 3 + 4. Следовательно, по окончании сопоставления переменная X будет иметь значение (1), а переменная Y — значение (2 + 3 + 4).

Другие примеры подобного типа:

```
[IS (!*X !*Y) (A + B + C)]  →  T,
    X:= ( ), Y:= (A + B + C)
[IS (!*X + !*Y + !*Z)
    ((A + B) + C — D + E + F)]  →  T,
    X:= ((A + B)), Y:= (C — D), Z:= (E + F)
[IS (!*X !*Y (!*Z + !.X) !*U)
    (B + C — (A + B + C) — (D + B))]  →  T,
    X:= (B), Y:= (+ C — (A + B + C) —),
    Z:= (D), U:= ( )
```

Отметим, что во втором из этих примеров образцу !*X противопоставляется сегмент из одного элемента (A + B), а не конструкция (A, поскольку атом + с верхнего уровня образца-списка не может соответствовать никакому атому с более глубоких уровней анализируемого списка. Более того, конструкция (A не является сегментом, ибо представляет собой последовательность символов, но не последовательность правильных плэнерских выражений.

Сказанного выше и приведенных примеров, по-видимому, достаточно для понимания правил сопоставления образцов-списков, содержащих сегментные элементы. Однако для большей четкости приведем точный алгоритм такого сопоставления.

Пусть образец-список P, среди элементов которого есть сегментные образцы, сопоставляется с выражением E. Если E не является L-списком или если длина списка E меньше числа несегментных элементов списка P, то сопоставление P с E неудачно. Иначе осуществляется последовательный слева направо просмотр элементов списка P. Если очередной элемент из P — это простой образец, то он сопоставляется с очередным элементом из E, а если этот элемент является сегментным образцом, то он сопоставляется (с учетом правила десегментации) с последовательностью очередных элементов из E, которая определяется следующим образом.

Если данный элемент является последним сегментным образцом в P, то берется такой сегмент из E, чтобы на долю оставшихся элементов списка P приходилось по одному элементу из оставшейся части списка E.

Для сегментного образца, не являющегося последним в P, выбор сегмента зависит от типа этого образца. Для !.-переменной, имеющей значение, и для сегментного обращения к константе

или функции выбирается сегмент, длина которого равна длине значения данной переменной, константы или функции. Но это делается только в том случае, если значением переменной, константы или функции является L-список и если в оставшейся части списка E хватает элементов для оставшихся несегментных элементов образца-списка P. Если эти условия не выполнены, соответствия данному сегментному образцу нет.

Для остальных типов сегментных образцов (для !.-переменных, не имеющих значений, для !*-переменных и для сегментных обращений к сопоставителям) сегмент определяется подбором. Сначала сегментный образец сопоставляется с пустым сегментом. Если это сопоставление неудачно, образец сопоставляется с сегментом из одного (очередного) элемента списка E. Если и это не привело к удаче, к сегменту добавляется следующий элемент из E, затем — еще один элемент и т. д. Такое наращивание продолжается до тех пор, пока не будет найден сегмент, который соответствует данному сегментному образцу, либо станет невозможным наращивание этого сегмента (в списке E не хватит элементов для оставшихся несегментных элементов из P). В первом случае соответствие сегментному образцу найдено, во втором — соответствия нет.

Последовательное слева направо сопоставление списков P и E продолжается до тех пор, пока не будет найдено соответствие всем элементам списка P или пока не встретится элемент из P, для которого нет соответствия в списке E. В первом случае сопоставление списков заканчивается, оно удачно, и при этом сохраняют свою силу все побочные эффекты сопоставления. Во втором же случае происходит возврат назад — к последнему из рассмотренных ранее сегментных элементов (с верхнего уровня) списка P, для которого возможно наращивание сопоставляемой с ним последовательности элементов списка E. Причем при возврате уничтожаются все побочные эффекты, которые имели место как во время предыдущего сопоставления этого сегментного образца, так и после. Затем к сопоставляемому с этим образцом сегменту добавляется очередной элемент списка E, и процесс сопоставления возобновляется с этой точки.

Может оказаться, что в списке P уже не осталось ни одного сегментного образца, для которого возможно наращивание сопоставляемого с ним сегмента. В таком случае сопоставление списков прекращается — оно неудачно. При этом все его побочные эффекты уничтожаются.

Поясним описанный алгоритм сопоставления на следующем примере:

[IS (!*X *Y .X !*Z) (A + B × (A + B) / C)]

Здесь образцу !*X сначала ставится в соответствии пустой сег-

мент, поэтому переменная X получает значение ( ). Затем образец *Y сопоставляется с атомом A, в результате чего переменной Y присваивается значение A. Теперь сопоставляются .X и +. Это сопоставление неудачно (переменная X имеет сейчас значение ( )), поэтому переменные X и Y «теряют» свои значения ( ) и A и происходит возврат к первому элементу образца-списка. На этот раз образцу !*X ставится в соответствии сегмент из одного элемента A, и переменная X получает теперь значение (A). После этого образец *Y удачно сопоставится с атомом +, однако образцу .X вновь не будет соответствия. Поэтому снова происходит возврат к образцу !*X, и т. д. В конце концов этому образцу будет поставлен в соответствие сегмент A + B, и переменная X получит значение (A + B). Далее образец *Y удачно сопоставится с атомом ×, в результате чего переменной Y присвоится значение ×. Сопоставление образца .X с очередным элементом анализируемого списка — с подсписком (A + B) — на этот раз удачно, поэтому далее ищется соответствие образцу !*Z. Поскольку этот сегментный образец является последним в образце-списке, ему ставится в соответствие весь остаток анализируемого списка, т. е. сегмент из элементов / и C. Следовательно, переменная Z получает значение (/ C). На этом сопоставление списков завершается, оно удачно. За переменными X, Y и Z сохраняются значения, полученные ими во время сопоставления: (A + B), × и (/ C).

Отметим, что из описанного алгоритма сопоставления списков вытекает следующее правило: если имеется несколько вариантов удачного сопоставления простого образца с выражением, то всегда рассматривается только один из них. В частности, при сопоставлении списков невозможен возврат к сегментным образцам, находящимся не на верхнем уровне образца-списка. В связи с этим сопоставление

[IS ((!*X A !*Y) .X) ((A B A) (A B))]

неудачно. Действительно, при сопоставлении первых элементов (подсписков) переменная X получает значение ( ), а переменная Y — значение (B A), после чего сопоставление вторых элементов оказывается неудачным. Но несмотря на это, другой возможный вариант удачного сопоставления подсписков, в котором переменная X получила бы значение (A B), а переменная Y — значение ( ), не рассматривается.

2.4. Примеры использования образцов

Приведем два примера использования образцов для анализа структуры данных.

Первый пример — это символьное дифференцирование многочленов. Под «многочленом» будем понимать «одночлен» или по-

следовательность «одночленов», соединенных знаками +. «Одночлен» — это «переменная» (плэнерский идентификатор), либо неотрицательное число, либо «степень» вида $x \uparrow n$, где x — «переменная», а n — целое число, большее или равное 2, либо последовательность таких конструкций, соединенных знаками $\times$. Весь многочлен заключается в круглые скобки. Примеры таких многочленов:

(2), (X $\times$ Y), (6 $\times$ X + X $\uparrow$ 3 $\times$ Y + X $\uparrow$ 2)

Следующая функция строит список, представляющий собой запись производной многочлена P по переменной V:

```
[DEFINE DIF (LAMBDA (P V) [PROG (T) [COND
    ([NOT [MEMB .V .P]] (0))
    ([IS (.V) .P] (1))
    ([IS (!*T + !*P) .P]
       (⟨DIF .T .V⟩ + ⟨DIF .P .V⟩))
    ([IS (!*T × !*P) .P]
       (⟨DIF .T .V⟩ × !.P + !.T × ⟨DIF .P .V⟩))
    ([IS (.V ↑ 2) .P] (2 × .V))
    ([IS (.V ↑ *T) .P]
       (.T × .V ↑ [— .T 1]))] ])]
```

В первой клаузе условного выражения проверяется, входит ли в многочлен P переменная, по которой ведется дифференцирование. Во второй клаузе выявляется случай, когда многочлен состоит только из этой переменной. В остальных клаузах с помощью образцов определяется структура многочлена: сумма ли это, одночлен или степень (особо выделен случай степени с показателем 2, чтобы избежать появления в производной степеней с показателем 1). Этими же образцами многочлен разбивается на отдельные компоненты. Например, у многочлена (6 $\times$ X + X $\uparrow$ 3 $\times$ Y + X $\uparrow$ 2) выделяются первое слагаемое T = (6 $\times$ X) и остальные слагаемые P = (X $\uparrow$ 3 $\times$ Y + X $\uparrow$ 2). После этого по соответствующей формуле дифференцирования строится список, являющийся производной многочлена. Так, для указанного выше многочлена применяется формула $(T + P)' = (T' + P')$, причем для получения производных T′ и P′ рекурсивно применяется сама функция DIF (считаем, что значением V является атом X):

⟨DIF .T .V⟩ → 0 $\times$ X + 6 $\times$ 1
⟨DIF .P .V⟩ →
3 $\times$ X $\uparrow$ 2 $\times$ Y + X $\uparrow$ 3 $\times$ 0 + 2 $\times$ X

Далее эти производные соединяются знаком + и заключаются в круглые скобки, что и дает производную исходного многочлена:

(0 $\times$ X + 6 $\times$ 1 + 3 $\times$ X $\uparrow$ 2 $\times$ Y +
X $\uparrow$ 3 $\times$ 0 + 2 $\times$ X)

Как видно, полученную производную следовало бы упростить, однако мы не будем заниматься этой проблемой.

Следующий пример, который мы рассмотрим,— это задача интерпретации программ, записанных на некотором модельном языке. Синтаксис этого языка описывается следующими металингвистическими формулами (в них для большей наглядности пробелы указываются явно, уголки же используются как метасимволы, а не как плэнерские скобки):

⟨программа⟩ ::= (⟨описание⟩␣⟨операторы⟩)
⟨описание⟩ ::= (VAR␣⟨переменные⟩)
⟨переменные⟩ ::= ⟨пусто⟩|⟨переменная⟩␣⟨переменные⟩
⟨операторы⟩ ::= ⟨пусто⟩|⟨оператор⟩␣⟨операторы⟩
⟨оператор⟩::= (⟨непомеченный оператор⟩) |
(⟨метка⟩␣:␣⟨непомеченный оператор⟩)
⟨непомеченный оператор⟩::= ⟨оператор перехода⟩ |
⟨оператор вывода⟩ | ⟨оператор присваивания⟩ |
⟨условный оператор⟩
⟨оператор перехода⟩::= GOTO␣⟨метка⟩
⟨оператор вывода⟩::= OUTPUT␣⟨выражение⟩
⟨оператор присваивания⟩::= ⟨переменная⟩␣=␣⟨выражение⟩
⟨условный оператор⟩::= IF ⟨условие⟩ THEN (⟨непомеченный оператор⟩) ELSE (⟨непомеченный оператор⟩)
⟨условие⟩::= (⟨выражение⟩␣⟨знак отношения⟩␣⟨выражение⟩)
⟨знак отношения⟩::= ≠ | =
⟨выражение⟩::= ⟨число⟩ | ⟨переменная⟩ |
⟨переменная⟩␣+␣⟨число⟩

Под «меткой» и «переменной» понимается любой плэнерский идентификатор, под «числом» — любое плэнерское число.

Семантика описанного языка естественна и очевидна.

Пример программы на этом языке:

```
((VAR S I) (S = 0) (I = 0)
(L : I = I + 1) (S = S + 2)
(IF (I ≠ 20) THEN (GOTO L) ELSE (OUTPUT S)))
```

Интерпретатор этого языка мы опишем в виде нескольких функций. Основной из них является функция INTERP, аргумент которой — интерпретируемая программа.

```
[DEFINE INTERP (LAMBDA (PROG)
   [PROG (VARS (AL ( )) OPS OP LAB)
      [COND ([IS ((VAR !*VARS) !*PROG) .PROG])
            (T [ERR 1 ( )])]
      [LOOP V .VARS
```

```
            [COND ([ID .V]
                       [SET AL (!AL (.V ()))])
                  (T [ERR 2 .V])]]
        [SET OPS .PROG]
    L [COND ([FIN OP OPS] [RETURN END])]
        [IS (*LAB : !*OP) .OP]
        [EVOP .OP]
        [GO L]])]
```

Эта функция строит список AL, в котором будут храниться переменные интерпретируемой программы и их значения. Первоначально этот список имеет вид $((v_1\ ())\ (v_2\ ())\ \ldots\ (v_k\ ()))$, где v_i — переменные, описанные в начале интерпретируемой программы, а ( ) означает отсутствие значения у переменной. В дальнейшем в этот список будут заноситься значения, получаемые переменными v_i. Например, когда переменной v_1 будет присвоено значение 5, список AL будет преобразован к виду $((v_1\ 5)\ (v_2\ ())\ \ldots)$.

Возможно, что описание переменных в интерпретируемой программе задано неправильно. Тогда функция INTERP передает управление функции ERR, которая печатает сообщение об ошибке и прекращает интерпретацию.

После обработки описания функция INTERP начинает последовательное выполнение операторов интерпретируемой программы. Список всех операторов этой программы хранится в переменной PROG, а список операторов, начиная с того, который должен быть выполнен в текущий момент,— в переменной OPS. Если список OPS пуст, тогда функция INTERP завершает интерпретацию и выдает значение END. Иначе от списка OPS отделяется первый оператор OP, из которого удаляется метка, если она есть. Получившийся непомеченный оператор выполняется с помощью функции EVOP. Далее описанный цикл повторяется.

```
[DEFINE EVOP (LAMBDA (OP)
    [PROG (E L V B OP1 OP2) [COND
        ([IS (OUTPUT !*E) .OP]
            [PRINT [EVEXP .E]])
        ([IS (GOTO *L) .OP] [GOTO .L])
        ([IS (*V = !*E) .OP]
            [PUTVAL .V [EVEXP .E]])
        ([IS (IF *B THEN *OP1 ELSE *OP2) .OP]
            [COND ([EVBOOL .B] [EVOP .OP1])
                  (T [EVOP .OP2])])
        (T [ERR 3 .OP])] ])]
```

Функция EVOP интерпретирует действия непомеченного оператора. Для этого она определяет тип оператора и выполняет сама (для оператора вывода и условного оператора) или с помощью других функций (для операторов перехода и присваивания) действия данного оператора. Если оператор записан неправильно, выдается сообщение об ошибке и интерпретация всей программы прекращается.

```
[DEFINE GOTO (LAMBDA (L) [PROG (B OP E)
    [COND ([NOT [ID .L]] [ERR 4 .L])
          ([IS (!*B (.L : !*OP) !*E) .PROG]
             [SET OPS [REST [LENGTH .B]
             .PROG]])
          (T [ERR 5 .L])] ])]
```

Функция GOTO интерпретирует оператор перехода. Она находит в списке PROG всех операторов интерпретируемой программы оператор, помеченный меткой L, и присваивает переменной OPS список операторов, начиная с найденного (переменные PROG и OPS являются внешними по отношению к функции GOTO и берутся из функции INTERP). Тем самым на следующем шаге интерпретации именно этот оператор и будет выбран функцией INTERP для выполнения. Если в операторе перехода в качестве метки указан не идентификатор или если указанной меткой не помечен ни один оператор программы, то выдается сообщение об ошибке, после чего интерпретация программы прекращается.

```
[DEFINE EVBOOL (LAMBDA (B) [PROG (E1 E2)
    [COND ([IS (!*E1 = !*E2) .B]
           [EQ [EVEXP .E1] [EVEXP .E2]])
          ([IS (!*E1 ≠ !*E2) .B]
             [NEQ [EVEXP .E1] [EVEXP .E2]])
          (T [ERR 6 .B])] ])]
```

Функция EVBOOL вычисляет условия в условных операторах.

```
[DEFINE EVEXP (LAMBDA (E) [PROG (N)
    [COND ([IS (*E) .E] [COND ([NUM .E] .E)
                              (T [GETVAL .E])])
          ([AND [IS (*E + *N) .E] [NUM .N]]
             [+ [GETVAL .E] .N])
          (T [ERR 7 .E])] ])]
```

Функция EVEXP вычисляет значения выражений интерпретируемой программы. Отметим, что выражения подаются на вход функции в виде L-списков.

Следующие две функции предназначены для работы со списком AL, где хранятся переменные интерпретируемой программы и их текущие значения. Функция GETVAL находит текущее значение указанной переменной, а функция PUTVAL заменяет прежнее значение на новое. В этих функциях учитывается возможность следующих ошибок: в качестве имени переменной указан не идентификатор; указанной переменной нет в списке AL; запрашивается значение переменной в тот момент, когда она его не имеет.

```
[DEFINE GETVAL (LAMBDA (V) [PROG (S VAL)
    [COND ([NOT [ID .V]] [ERR 2 .V])
          ([IS (!*S (.V *VAL) !*S) .AL]
              [COND (.VAL) (T [ERR 9 .V])])
          (T [ERR 8 .V])] ])]
[DEFINE PUTVAL (LAMBDA (V VAL) [PROG (B E X)
    [COND ([NOT [ID .V]] [ERR 2 .V])
          ([IS (!*B (.V *X) !*E) .AL]
              [SET AL (!.B (.V .VAL) !.E)])
          (T [ERR 8 .V])] ])]
```

И, наконец, последняя функция интерпретатора — это функция ERR, которая реагирует на ошибки, встретившиеся в интерпретируемой программе. Данная функция по номеру ошибки выдает на печать название ошибки и фрагмент программы, в котором была обнаружена ошибка, после чего осуществляет выход из функции INTERP (со значением ABEND), т. е. прекращает работу интерпретатора.

```
[DEFINE ERR (LAMBDA (N F)
    [DO [MPRINT <.N :ERRORS> : !.F]
        [EXIT ABEND INTERP]])]
```

Названия ошибок перечислены в списке, являющемся значением константы ERRORS. Определение этой константы также входит в описание интерпретатора.

```
[CSET ERRORS
    ((НЕТ ОПИСАНИЯ)
    (ИМЯ ПЕРЕМЕННОЙ — НЕ ИДЕНТИФИКАТОР)
    (НЕПРАВИЛЬНЫЙ ОПЕРАТОР)
    (МЕТКА — НЕ ИДЕНТИФИКАТОР)
    (НЕОПИСАННАЯ МЕТКА)
    (НЕПРАВИЛЬНОЕ УСЛОВИЕ)
    (НЕПРАВИЛЬНОЕ ВЫРАЖЕНИЕ)
    (НЕОПИСАННАЯ ПЕРЕМЕННАЯ)
    (ПЕРЕМЕННАЯ БЕЗ ЗНАЧЕНИЯ))]
```

Запуск интерпретатора осуществляется обращением:

[INTERP *prog*]

где *prog* — программа, которую надо проинтерпретировать.

2.5. Сопоставители

Хотя рассмотренные в предыдущих параграфах образцы и предоставляют пользователю весьма богатые возможности для анализа данных, во многих случаях их все же недостаточно. Например, нельзя построить образец, с помощью которого можно было бы проверить, является ли анализируемое выражение E списком, первый элемент которого — атом (любой). Правда, такой анализ можно провести, используя образцы и функции:

[AND [IS (*X !*Y) .E] [ATOM .X]]

однако с помощью только одних образцов этот анализ провести нельзя. Введение в язык сопоставителей как раз и обусловлено стремлением расширить возможности образцов так, чтобы в виде образцов можно было описать любое правило анализа данных.

Сопоставители — это особый класс процедур, обращаться к которым можно только в образцах. Синтаксически обращения к сопоставителям записываются так же, как и обращения к функциям: *простое обращение к сопоставителю* с именем g и аргументами a_i задается в виде P-списка

$[g\ a_1\ a_2\ \ldots\ a_k]$

а *сегментное обращение к сопоставителю* — в виде S-списка:

$\langle g\ a_1\ a_2\ \ldots\ a_k \rangle$

Как и в случае функций, при обращении к сопоставителю разрешается в качестве первого элемента списка-обращения указывать .-переменную или :-переменную. В этих ситуациях имя сопоставителя, к которому происходит обращение, задается значением данной переменной или константы, которое обязано быть идентификатором.

Однако в отличие от функций сопоставители не вырабатывают никаких значений. Выполнение сопоставителя заключается в проверке, обладает ли объект, с которым он сопоставляется, определенным свойством или нет. При простом обращении сопоставитель сопоставляется с одним выражением, при сегментном — с последовательностью выражений. Если проверяемое выражение или сегмент обладает нужным свойством, сопоставление этого объекта с сопоставителем считается удачным, не обладает — неудачным.

Каждый сопоставитель проверяет «свое» свойство. Например, в языке имеется сопоставитель ATOM без аргументов, который проверяет свойство «быть атомом». Поэтому простое обращение к этому сопоставителю, т. е. образец [ATOM], соответствует только атомам:

```
[IS [ATOM] A]  →  T
[IS [ATOM] (A)]  →  ()
[IS ([ATOM] !*X) (A B C)]  →  T,  X:= (B C)
```

(Отметим, что в последнем примере использован образец, позволяющий провести анализ, о котором шла речь в начале параграфа.)

Другой пример — сопоставитель [LIST 2], который соответствует любому L-списку из двух элементов:

```
[IS [LIST 2] (A B)]  →  T
[IS (*X [LIST 2]) (A (B))]  →  ()
```

Сегментный вариант обращения к этому сопоставителю соответствует любому сегменту из двух элементов:

```
[IS (A ⟨LIST 2⟩ D) (A B C D)]  →  T
```

В общем случае свойства, которые проверяют сопоставители, зависят от некоторых параметров. Аргументы в обращениях к сопоставителям как раз и задают конкретные значения этих параметров. Так, сопоставитель LIST в общем случае проверяет свойство «быть L-списком», а его аргумент уточняет данное свойство, накладывая ограничение на длину списка. Например, образец [LIST 5] соответствует только L-спискам из пяти элементов, а образец [LIST 0] — только пустому списку ().

Кроме сопоставителей, проверяющих свойства, в языке есть сопоставители, которые позволяют задавать различные комбинации образцов. К примеру, сопоставитель AUT, аргументами которого являются образцы, определяет «дизъюнкцию» этих образцов: он соответствует любому объекту, который соответствует хотя бы одному из его аргументов-образцов. В частности, образец [AUT () [LIST 2]] соответствует как пустому списку (), так и любому L-списку из двух элементов, а образец [AUT + — ×] соответствует атому +, или атому —, или атому ×.

На сопоставители распространяются все правила сопоставления образцов с выражениями, рассмотренные нами в предыдущих параграфах. Так, выполнение процедуры-сопоставителя может сопровождаться побочными эффектами; например, во время сопоставления

```
[IS [AUT [ATOM] (A *X)] (A B)]  →  T
```

переменная X получит значение B. Но как и раньше, такие побочные эффекты сохраняют свою силу только в случае удачного исхода сопоставления в целом. Поэтому при неудачных сопоставлениях

```
[IS [AUT (*X + 0) (0 + *X)] (0 — 2)]
[IS ([AUT [ATOM] (¬ *X)] ∧ .X) ((¬ A) ∧ B)]
```

переменная X не изменит своего значения, хотя в процессе сопоставления ей и присваивалось (условно) значение 0 в первом примере или значение A во втором примере.

На сопоставители распространяется и правило, которое гласит, что из всех возможных вариантов удачного сопоставления образца с выражением всегда рассматривается только один (первый) вариант. Например, при сопоставлении

```
[IS [AUT (A + *X) (*X + B)] (A + B)]
```

учитывается только вариант удачного сопоставления первого аргумента-образца со списком (A + B), а вариант удачного сопоставления образца (*X + B) с этим списком уже не рассматривается. Поэтому после сопоставления переменная X будет иметь значение B.

И, наконец, на сопоставители распространяется правило десегментации: сегментное обращение к сопоставителю соответствует последовательности выражений тогда и только тогда, когда L-список из этих элементов соответствует простому обращению к данному сопоставителю. Например, образец ⟨LIST 3⟩ соответствует сегменту A B C, поскольку образец [LIST 3] соответствует списку (A B C). Удачным является и сопоставление

```
[IS (A ⟨AUT (+) (+ *X +)⟩ C) (A + B + C)]
```

так как второму элементу образца-списка ставится в соответствие сегмент + B + и при этом сопоставление образца [AUT (+) (+ *X +)] со списком (+ B +) удачно; значением переменной X становится атом B. В то же время сопоставление

```
[IS (A ⟨AUT — +⟩ B) (A + B)]
```

неудачно. Действительно, образец ⟨AUT — +⟩ здесь сопоставляется с сегментом из одного элемента +, а это сопоставление неудачно, поскольку неудачно сопоставление образца [AUT — +] со списком (+). Неудачным является и сопоставление

```
[IS (⟨ATOM⟩ !*X) (A B C)],
```

так как образцу [ATOM] не соответствует никакой список и, следовательно, образцу ⟨ATOM⟩ не соответствует ни один сегмент.

Как и функции, сопоставители делятся на *встроенные* и *определяемые*. Правила вычисления встроенных сопоставителей заранее известны в языке, поэтому их не надо определять, а можно прямо обращаться к ним. Эти сопоставители осуществляют проверку простейших свойств и задают основные способы объединения образцов. На базе встроенных сопоставителей и других образцов можно определять новые сопоставители, описывая в виде них любые желаемые правила анализа данных.

В следующем параграфе рассматриваются встроенные сопоставители языка, а в § 2.7 будут описаны правила определения новых сопоставителей.

2.6. Встроенные сопоставители

Как и встроенные функции, встроенные сопоставители делятся на два класса — SUBR и FSUBR. К классу SUBR относятся сопоставители, все аргументы которых вызываются по значению. Следовательно, аргументы таких сопоставителей должны быть простыми формами, а выполнение сопоставителей начинается только после того, как вычислены значения всех их аргументов. Остальные встроенные сопоставители относятся к классу FSUBR. Их аргументами, как правило, являются простые образцы, а они, естественно, не вычисляются.

При описании каждого встроенного сопоставителя мы будем указывать его класс, вид простого обращения к нему и правило сопоставления при таком обращении. Использоваться будут те же обозначения, что и при описании встроенных функций. Символом *pat* будут обозначаться только простые образцы.

Первая группа встроенных сопоставителей, которые мы рассмотрим,— это сопоставители, проверяющие тип сопоставляемых с ними выражений. Ниже перечислены те из них, которые проверяют типы атомарных выражений; слева указаны названия этих сопоставителей, а справа — тип тех выражений, которым и только которым они соответствуют. Все эти сопоставители не имеют аргументов.

ID — идентификаторы,
INT — целые числа,
REAL — вещественные числа,
NUM — любые числа,
SCALE — шкалы,
ATOM — атомы,
VAR: — :-переменные,
VAR. — .-переменные,
VAR* — *-переменные,
VARP — простые обращения к переменным,

VAR!: — !:-переменные,
VAR!. — !.-переменные,
VAR!* — !*-переменные,
VARS — сегментные обращения к переменным,
VAR — любые обращения к переменным,
ATOMIC — атомарные выражения.

Примеры:

```
[IS [ATOMIC] ()]  →  ()
[IS ([VARP] [NUM])
    [QUOTE (:PI 3.14)]]  →  T
[IS (A ⟨ID⟩ C) (A B C)]  →  ()
```

Следующие четыре сопоставителя относятся к классу FSUBR и проверяют типы списковых выражений:

[LIST *pat*?] — L-списки (с круглыми слобками),
[LISTP *pat*?] — P-списки (с квадратными скобками),
[LISTS *pat*?] — S-списки (с угловыми скобками),
[LISTR *pat*?] — любые списки.

Все эти сопоставители действуют аналогично, поэтому ограничимся описанием правила сопоставления только для сопоставителя LIST. Если при обращении к этому сопоставителю не задан аргумент, то сопоставитель соответствует любому L-списку. Если же аргумент задан, то сопоставитель соответствует только тем L-спискам, длина которых сответствует образцу *pat*. Таким образом, проверив, что с ним сопоставляется L-список, сопоставитель определяет его длину и полученное число сопоставляет с *pat*. Если это сопоставление удачно, то считается, что сопоставитель LIST соответствует анализируемому списку, неудачно — не соответствует. Примеры:

```
[IS [LISTS] [QUOTE ⟨A B⟩]]  →  T
[IS [LISTR 3] (A (B C))]  →  ()
[IS [LIST *X] (A (B C))]  →  T,  X:=2
[IS (A ⟨LIST 0⟩ B) (A B)]  →  T
[IS (A ⟨LISTP⟩ E) (A B C D E)]  →  ()
```

Неудача сопоставления в последнем примере обусловлена тем, что сопоставление образца ⟨LISTP⟩ с сегментом B C D, согласно правилу десегментации, сводится к сопоставлению образца [LISTP] с L-списком (B C D), а оно неудачно. (Напомним, что сегменты всегда заключаются в к р у г л ы е скобки.)

Имена перечисленных выше, а также ряда других встроенных сопоставителей совпадают с именами встроенных функций. Естественно, возникает вопрос: что имеется в виду — функция или сопоставитель, когда в программе встречается обращение к про-

цедуре, скажем, с именем ATOM? Ответ такой. Если данное обращение указано в том месте, где могут находиться только формы, то оно рассматривается как обращение к функции (образцы не являются формами, их нельзя вычислить). Поэтому выражение [SET X [ATOM .Y]] законно, а выражение [SET X [ATOM]] ошибочно (у функции ATOM должен быть один аргумент). В образцах же предпочтение отдается сопоставителям: если в том месте, где должен находиться образец, встретилось обращение к процедуре, которая может быть как функцией, так и сопоставителем, то оно рассматривается как обращение к сопоставителю. Поэтому выражения [IS [ATOM] .X] и [IS ([ATOM] B) (A B)] законны, а выражения [IS [ATOM .X] T] и [IS ([ATOM A] B) (A B)] ошибочны, так как у сопоставителя ATOM не должно быть аргументов. Таким образом, в позиции образца нельзя обращаться к функции, имя которой совпадает с именем сопоставителя. (Отметим, что одноименными могут быть только встроенные процедуры. Определение же новой процедуры с именем, которое имела другая, встроенная или определяемая, процедура, приводит, как известно, к уничтожению последней.)

Обращение к следующему встроенному сопоставителю «выпадает» из общего синтаксиса обращений к процедурам:

[]

Этот пустой P-список воспринимается в языке как обращение к сопоставителю, которому соответствует любое выражение. Такой образец используется тогда, когда надо указать, что в соответствующей позиции анализируемого списка должен находиться какой-то один элемент, а какой именно — не играет роли. Например, проверить, является ли значение переменной X L-списком из четырех элементов, первый из которых — атом A, третий — атом C, а два других — любые, можно так:

[IS (A [] C []) .X]

Сегментное обращение к этому безымянному сопоставителю записывается в виде пустого S-списка: ⟨⟩. Этому образцу соответствует любой сегмент. Поэтому

[IS (*X ⟨⟩ .X) (A B B A)] → T, X:=A
[IS (⟨⟩ [REAL] ⟨⟩) (A 5 B C 2.4)] → T

В отличие от образца [], образец ⟨⟩ используется тогда, когда надо указать, что в соответствующем месте анализируемого списка может находиться л ю б о е количество элементов, в том числе и ни одного. Поэтому

[IS (A []) (A)] → (), но [IS (A ⟨⟩) (A)] → T
[IS (A []) (A B)] → T и [IS (A ⟨⟩) (A B)] → T
[IS (A []) (A B C)] → (), но [IS (A ⟨⟩) (A B C)] → T

Отметим также разницу между образцами [ ] и *X и образцами ⟨ ⟩ и !*X. Каждый образец первой пары соответствует любому одному выражению, а каждый образец второй пары — любому сегменту. Однако при сопоставлении образцов [ ] и ⟨ ⟩ нет побочных эффектов, в то время как при сопоставлении образцов *X и !*X переменной X присваивается некоторое значение. Поэтому образцы *X и !*X применяются тогда, когда нам надо не только указать, что в соответствующем месте анализируемого выражения должно что-то находиться, но и надо это «что-то» выделить, присвоить переменной. Если же это «что-то» нас не интересует, то лучше использовать образцы [ ] и ⟨ ⟩. Например, если мы хотим узнать только одно — есть ли среди элементов списка L хотя бы один подсписок (с любыми скобками) из трех элементов, то это лучше сделать с помощью сопоставления

[IS (⟨ ⟩ [LISTR 3] ⟨ ⟩) .L],

а не сопоставления

[IS (!*X [LISTR 3] !*X) .L],

поскольку в последнем случае приходится вводить лишнюю вспомогательную переменную X.

Следующая группа встроенных сопоставителей — это так называемые арифметические сопоставители. Они относятся к классу SUBR. Сначала они проверяют, является ли числом сопоставляемое с ними выражение E. Если нет, сопоставление неудачно. Иначе это число сравнивается со значением их аргумента — с числом *N*. Если проверяемое отношение (ниже оно указано справа) выполняется, тогда сопоставление удачно, не выполняется — неудачно.

[LT *n*] — $E < N$?
[LE *n*] — $E \leqslant N$?
[GT *n*] — $E > N$?
[GE *n*] — $E \geqslant N$?

Примеры:

[IS [LT 4] A] → ( )
[IS [LT 4] 2.8] → T
[IS [LT 5.2] 7] → ( )
[IS [LIST [GE 2]] (A B)] → T

Третья группа встроенных сопоставителей — это логические сопоставители.

Сопоставитель NON: [NON *pat*], FSUBR.

Это «отрицание» образца *pat*. Сопоставитель NON соответствует

сопоставляемому с ним выражению тогда и только тогда, когда это выражение не соответствует образцу *pat*. Например:

```
[IS [NON [ATOM]] A]   →   ()
[IS (⟨LIST *N⟩ [NON A] ⟨ ⟩)
    (A A B)]   →   T,   N:=2
```

Отметим, что при любом исходе сопоставления выражения с данным сопоставителем все побочные эффекты этого сопоставления уничтожаются. Действительно, если образец *pat* не соответствует анализируемому выражению, то побочных эффектов нет, а если *pat* соответствует выражению, тогда сам сопоставитель NON сигнализирует о неудаче, поэтому побочные эффекты все равно будут отменены. Например:

```
[IS [NON (*X A)] (A B)]   →   T
[IS [NON *X] A]   →   ()
```

и в обоих случаях переменная X не изменит своего значения.

Сопоставитель ET: [ET $pat_1 \ldots pat_k$], FSUBR, $k \geqslant 1$.

Это «конъюнкция» образцов pat_i: сопоставитель ET соответствует анализируемому выражению, только если оно соответствует всем образцам pat_i. Более точно, сопоставитель просматривает слева направо свои аргументы-образцы и по очереди сопоставляет их с выражением. Если все эти сопоставления удачны, то считается, что сопоставитель соответствует этому выражению. Но если сопоставление какого-то образца pat_i с выражением неудачно, то оставшиеся образцы не рассматриваются, побочные эффекты сопоставления предыдущих аргументов-образцов уничтожаются и сопоставление ET с выражением завершается неудачей.

Примеры:

```
[IS [ET [INT] [GE 0]] 5]   →   T
[IS [ET [LIST 3] (A !*X)]
    (A B C)]   →   T,   X:=(B C)
[IS [ET (*X + *Y) ([ID] + [NUM])]
    (A + B)]   →   ()
[IS [ET *X [ABS ,X]] A] — ошибка
```

Отметим, что в предпоследнем примере переменные X и Y не меняют своих значений, а ошибка в последнем примере вызвана тем, что полученное переменной X нечисловое значение A использовалось как аргумент функции ABS, что запрещено правилами языка.

Наиболее часто сопоставитель ET используется в тех случаях, когда требуется некоторой переменной X присвоить в качестве значения анализируемый объект, но только при условии, что этот

объект обладает свойством, которое проверяется некоторым образцом *pat*. Сделать это можно, построив образец [ET *pat* *X], который сначала проверяет, обладает ли анализируемый объект нужным свойством, а затем, если объект прошел проверку, присваивает его переменной X. Например, найти среди элементов списка L самое левое число и присвоить его переменной X можно так:

[IS (⟨ ⟩ [ET [NUM] *X] ⟨ ⟩) .L]

Сопоставитель SAME. Обращение к этому сопоставителю имеет следующий вид:

[SAME (v_1 v_2 ... v_m) pat_1 ... pat_k], FSUBR,
$m \geqslant 0$, $k \geqslant 1$.

Сопоставитель SAME действует аналогично сопоставителю ET, но на время своей работы вводит локальные переменные согласно их описанию (v_1 v_2 ... v_m), что делается так же, как в функции PROG (см. § 1.9). Например:

[IS [SAME (X) [LIST 4] (*X ⟨ ⟩ .X)]
(A B C A)] → T

Сопоставитель AUT: [AUT pat_1 ... pat_k], FSUBR, $k \geqslant 1$. Это «дизъюнкция» образцов pat_i. Сопоставитель просматривает слева направо свои аргументы-образцы и по очереди сопоставляет их с анализируемым выражением. Если все эти сопоставления неудачны, то соответствия между сопоставителем и выражением нет. Если же нашелся образец pat_i, которому соответствует анализируемое выражение, тогда сопоставитель, не рассматривая оставшиеся аргументы, завершает свою работу с удачей. Пример:

[IS [AUT (*X B) (A *X) (*X C)]
(A C)] → T, X:=C

Рассмотрим на примере сопоставителя AUT принцип построения сегментных образцов; такое построение нередко вызывает затруднения у программирующих на плэнере.

Предположим, что требуется написать образец, которому соответствует пустой сегмент или сегмент из одного элемента +. Ясно, что этот образец должен быть сегментным, раз он сопоставляется с сегментами, но вот сообразить сразу, каков конкретно этот образец, наверное, трудно. Выйти из затруднения помогает правило десегментации, согласно которому простой вариант данного образца должен соответствовать пустому списку ( ) или списку (+). Построить же такой простой образец легко — это [AUT ( ) (+)]. Теперь, возвращаясь к сегментному варианту, получаем искомый образец: ⟨AUT ( ) (+)⟩.

Примеры использования этого образца:

[IS (⟨AUT ( ) (+)⟩ A — B) (A — B)] → T
[IS (⟨AUT ( ) (+)⟩ A — B) (+ A — B)] → T
[IS (⟨AUT ( ) (+)⟩ A — B) (— A — B)] → ( )

Таким образом, сегментный образец сначала следует строить так, как если бы он был простым образцом, который сопоставляется с L-списками, и лишь затем следует перейти к сегментному варианту, т. е. заменить внешние квадратные скобки простого образца на угловые (ничего не меняя при этом «внутри» образца).

Сопоставитель WHEN. Это условный сопоставитель. Он относится к классу FSUBR. Обращение к нему имеет следующий вид:

$$\left[\text{WHEN } \left(pat_1\ pat_{11} \ldots pat_{1m_1}\right) \ldots \left(pat_k\ pat_{k1} \ldots pat_{km_k}\right)\right],$$

где $k \geqslant 1$, $m_i \geqslant 0$. Данный сопоставитель действует так. Сначала он по очереди сопоставляет анализируемое выражение с образцами pat_i — первыми элементами своих аргументов (клауз). Если выражение не соответствует ни одному из них, то соответствия между сопоставителем и выражением нет. Если же нашелся образец pat_i, которому соответствует анализируемое выражение, тогда остальные клаузы уже не рассматриваются и действие сопоставителя сводится к последовательному сопоставлению выражения с оставшимися образцами из этой i-й клаузы, а точнее, к сопоставлению выражения с образцом $\left[\text{ET } pat_{i1}\ pat_{i2} \ldots pat_{im_i}\right]$ (при $m_i = 0$ такое сопоставление считается удачным).

Например, образец

[WHEN ([ID] A) ([NUM] [GE 2] [LE 5]) ([])]

из всех идентификаторов соответствует только идентификатору A, из чисел — только числам от 2 до 5, а из выражений иных типов — чему угодно.

В последнюю группу встроенных сопоставителей языка входят сопоставители LINEAR, STAR, ONE-OF, PAT, BE и HAS.

Сопоставитель LINEAR. Обращение к нему таково:

$$\left[\text{LINEAR } pat'_1\ pat'_2 \ldots pat'_k\right],\ \text{FSUBR},\ k \geqslant 1.$$

В отличие от всех других встроенных сопоставителей, у этого сопоставителя аргументами могут быть как простые образцы, так и сегментные. Сопоставителю LINEAR соответствуют списки с любыми скобками, при этом правила его сопоставления аналогичны правилам сопоставления образца-списка $\left(pat'_1\ pat'_2 \ldots pat'_k\right)$ Таким образом, данному сопоставителю соответствует любой список, если такой же список, но в круглых скобках, соответствует указанному образцу-списку. Например:

[IS [LINEAR INDEX !*X [VAR]]
[QUOTE ⟨INDEX 1 2 ;A⟩]] → T, X:= (1 2)

Сопоставитель STAR: [STAR *pat*], FSUBR.

Данный сопоставитель соответствует любому списку, каждый элемент которого соответствует образцу *pat*. Более точно, сопоставитель STAR, проверив, что анализируемое выражение является списком (с любыми скобками), сопоставляет образец *pat* по очереди с каждым элементом этого списка. Пустым спискам данный сопоставитель соответствует всегда.

Например, образец [STAR [ID]] соответствует любому списку, все элементы которого — идентификаторы, а образец ⟨STAR [ID]⟩ соответствует любому сегменту из идентификаторов:

```
[IS [STAR [ID]] (A B C)]  →  T
[IS [STAR [ID]] (A B 2.4 C)]  →  ()
[IS (!*X ⟨STAR [ID]⟩) (1 2 A B)]  →  T,  X:= (1 2)
```

Напомним, что, согласно общему принципу сопоставления, побочные эффекты удачного сопоставления сохраняются, а в случае неудачи они уничтожаются. Поэтому

```
[IS [STAR *X] (A B C)]  →  T,  X:= C
[DO [UNASSIGN X]
    [IS [STAR .X] (A A)]]  →  T,  X:= A
[DO [UNASSIGN X]
    [IS [STAR .X] (A B)]]  →  ()
```

(в последнем примере переменная X остается без значения).

Сопоставитель ONE-OF: [ONE-OF *l*], SUBR.

Значением аргумента должен быть список. Сопоставитель соответствует анализируемому выражению тогда и только тогда, когда это выражение равно одному из элементов данного списка. Например:

```
[DO [SET X (A B C)] [IS [ONE-OF .X] D]]  →  ()
[IS (!*X (ONE-OF (+ —)] ⟨ ⟩)
    (A — B + C)]  →  T,  X:= (A)
```

Сопоставитель PAT: [PAT *e*], SUBR.

Значением аргумента должно быть выражение, которое может выступать в роли простого образца. PAT сопоставляет этот образец с анализируемым выражением. Если это сопоставление удачно, то сопоставитель PAT соответствует выражению, неудачно — не соответствует.

Данный сопоставитель применяется в тех случаях, когда с анализируемым выражением надо сопоставить образец, который заранее неизвестен, а является значением какой-нибудь переменной или строится во время вычисления программы.

Пусть, к примеру, значением переменной X является список (A *Y). Тогда

[IS [PAT .X] (A B)] → T, Y:= B

В то же время

[IS .X (A B)] → ()

Действительно, если в первом примере значение переменной X рассматривается как образец, который сопоставляется со списком (A B), то во втором примере это значение просто сравнивается, а эти два списка не равны.

Сопоставитель BE: [BE *e*], SUBR.

Данный сопоставитель «игнорирует» сопоставляемое с ним выражение и интересуется только значением своего аргумента. Если оно равно (), то сопоставитель не соответствует никакому выражению, а при других значениях аргумента сопоставитель соответствует любому выражению.

Сопоставитель BE используется обычно в тех случаях, когда в процессе сопоставления надо что-то выполнить с помощью функций. Например, при сопоставлении

[IS (< > [ET [ATOM] *X [BE [PRINT .X]]] < >)
((A) (B C) D E)] → T

переменной X будет присвоено значение D и этот атом будет выдан на печать.

В § 1.16 была описана встроенная функция EXIT, с помощью которой можно прекратить выполнение любой объемлющей процедуры. Там был рассмотрен выход из функций, здесь же мы рассмотрим выход из сопоставителей.

Пусть дано обращение к функции EXIT:

[EXIT *e fn n*?]

и пусть значением аргумента *fn* является идентификатор *FN* — имя некоторого сопоставителя, объемлющего данное обращение. Тогда функция EXIT завершает работу этого сопоставителя, объявляя выполняемое им сопоставление неудачным, если значением аргумента *e* является пустой список (), или удачным, если значение аргумента *e* отлично от (). В случае «неудачного» выхода все побочные эффекты, имевшие место с начала выполнения сопоставителя, отменяются, при «удачном» же выходе побочные эффекты сохраняются.

Третий аргумент функции EXIT указывает, из какого по счету объемлющего сопоставителя с именем *FN* осуществляется выход. Если этого аргумента нет или если его значение *N* равно

нулю, тогда имеется в виду ближайший объемлющий сопоставитель *FN*, при других значениях — $(N+1)$-й по счету сопоставитель *FN*. Если среди объемлющих процедур нет нужного сопоставителя, тогда функция EXIT осуществляет выход на верхний уровень программы (см. § 1.16).

С учетом сказанного образец

```
[SAME (X) [WHEN ([ATOM] [NUM])
                (*X [BE [EXIT .X SAME]])]]
```

из всех атомов соответствует только числам, а из выражений иных типов — чему угодно, кроме пустого списка ( ).

Сопоставитель HAS. Обращение к нему имеет вид

[HAS ind_1 pat_1 ... ind_k pat_k], FSUBR, $k \geqslant 1$.

Этот сопоставитель соответствует только идентификаторам, причем таким, чьи списки свойств удовлетворяют следующему требованию: значения свойств с названиями IND_i (это значения аргументов ind_i) должны соответствовать образцам pat_i. Более точно сопоставитель HAS действует так. Если анализируемое выражение — не идентификатор, сопоставление неудачно. Иначе сопоставитель вычисляет значение IND_1 своего аргумента ind_1, находит в списке свойств идентификатора свойство с названием IND_1 и значение этого свойства (либо пустой список ( ), если такого свойства нет) сопоставляет с образцом pat_1. Если данное сопоставление удачно, то сопоставитель выполняет аналогичные действия по отношению к следующей паре аргументов — ind_2 и pat_2, и т. д. В случае успеха всех этих проверок сопоставление идентификатора с сопоставителем HAS считается удачным. Но если хотя бы одна проверка оказалась неудачной, тогда HAS сразу прекращает свою работу, отменяет все побочные эффекты предыдущих проверок и сигнализирует о неудаче своего сопоставления.

Пусть, к примеру, идентификатор CONS имеет свойство с названием TYPE и значением SUBR, а также свойство с названием NARG и значением 2. Тогда

```
[IS [HAS NARG [GT 2]] CONS]   →   ( )
[IS ([HAS TYPE SUBR NARG *N] ⟨LIST .N⟩)
    (CONS A B)]   →   T,   N:=2
```

2.7. Определение новых сопоставителей

Для определения нового сопоставителя следует обратиться к встроенной функции DEFINE:

[DEFINE *g* (KAPPA *var pat*)].

Здесь *g* — идентификатор, имя определяемого сопоставителя,

а список (КАРРА *var pat*) — *определяющее выражение* этого сопоставителя: элемент КАРРА указывает, что определяется сопоставитель (а не функция или теорема), элемент *var* — это описание *параметров* сопоставителя, а простой образец *pat* — его *тело*.

Описание параметров сопоставителя задается так же, как и описание параметров определяемой функций (см. § 1.12). Если *var* — идентификатор, то это — имя единственного параметра сопоставителя. Обращаться к такому сопоставителю можно с любым числом аргументов, которые должны быть формами, простыми или сегментными; все эти аргументы будут вычислены и L-список из их значений будет присвоен параметру сопоставителя. Если *var* — идентификатор, помеченный звездочкой, то у сопоставителя также один параметр и любое число аргументов, но в данном случае параметру присваивается L-список из невычисленных аргументов. И, наконец, *var* может иметь вид $(v_1\ v_2\ \ldots\ v_m)$, $m \geqslant 0$, где v_i — либо просто идентификаторы, либо идентификаторы, помеченные звездочкой. При обращении к такому сопоставителю надо задавать ровно m аргументов, при этом если параметр v_i не помечен, то ему присваивается значение i-го аргумента, который должен быть простой формой, а если помечен — присваивается сам i-й аргумент в том виде, как он задан в обращении.

При обращении к сопоставителю, определенному пользователем, выполняются следующие действия. Сначала, если нужно, вычисляются аргументы (все или только часть их), заданные в обращении. Затем вводятся параметры сопоставителя, и им присваиваются соответствующие значения. После этого осуществляется сопоставление тела сопоставителя, т. е. образца *pat*, с анализируемым объектом (с объектом, который поставлен в соответствие данному обращению к сопоставителю). Параметры сопоставителя локализуются в его теле, где они рассматриваются как обычные переменные. Если сопоставление тела сопоставителя с анализируемым объектом удачно, то считается, что сопоставитель соответствует этому объекту, неудачно — не соответствует.

Определения сопоставителей могут быть рекурсивными. В теле сопоставителей допускаются обращения к еще не определенным процедурам, однако к моменту первого обращения к сопоставителю все эти процедуры должны быть уже определены в программе.

Проиллюстрируем сказанное на конкретных примерах:

```
[DEFINE NATOM (KAPPA () [NON [ATOM]])]
```

У этого сопоставителя нет параметров, поэтому обращаться к нему надо без аргументов: [NATOM]. Сопоставитель соответствует любым выражениям, кроме атомов. Действительно, сопоставление образца [NATOM] с любым выражением сводится к сопоставлению тела этого сопоставителя, т. е. образца [NON [ATOM]], с этим же

выражением, а данному образцу соответствуют только неатомы.

```
[DEFINE LG (KAPPA (MIN MAX)
    [LISTR [ET [GE .MIN] [LE .MAX]]])]
```

Сопоставитель LG соответствует любому списку, длина которого не меньше MIN и не больше MAX. Рассмотрим, например, сопоставление

```
[IS [LG 2 5] (A B C)]
```

Выполнение сопоставителя LG начинается с того, что параметру MIN присваивается значение 2, а параметру MAX — значение 5. Далее осуществляется сопоставление образца

```
[LISTR [ET [GE .MIN] [LE .MAX]]]
```

со списком (A B C). Оно удачно, поэтому удачным является и сопоставление образца [LG 2 5] со списком (A B C), т. е. функция IS вырабатывает значение T.

```
[DEFINE CONTAINS (KAPPA (*P)
    (< > [PAT .P] < >))]
```

Образцу [CONTAINS *pat*] соответствуют только L-списки, причем такие, у которых хотя бы один элемент верхнего уровня соответствует образцу *pat*. Например:

```
[IS [CONTAINS [ID]] (5 6 A 2)]  →  T
```

Действительно, значением параметра P при этом обращении к сопоставителю является выражение [ID]. Поэтому сопоставление тела сопоставителя — образца (< > [PAT .P] < >) — со списком (5 6 A 2) удачно.

```
[DEFINE PALYNDROM (KAPPA ( )
    [AUT ( ) [LIST 1]
        [SAME (X) (*X <PALYNDROM> .X)]])]
```

Данный сопоставитель соответствует L-спискам, у которых элементы, равноудаленные от их концов, равны. Определение сопоставителя рекурсивно: он соответствует либо пустому L-списку, либо L-списку из одного элемента, либо L-списку, у которого первый и последний элементы равны, а середина соответствует этому же сопоставителю. Поэтому

```
[IS [PALYNDROM] (A B A)]  →  T
[IS [PALYNDROM]
    ((A + B) — C — (A + B))]  →  T
[IS [PALYNDROM] (A B C A)]  →  ( )
```

Рассмотрим подробнее, как осуществляется первое из этих сопоставлений. Параметров у сопоставителя PALYNDROM нет,

поэтому сразу начинается сопоставление его тела с анализируемым списком:

```
[IS [AUT ( ) [LIST 1]
    [SAME (X) (*X <PALYNDROM> .X)]] (A B A)]
```

Так как первый и второй образцы из AUT не соответствуют списку (A B A), с этим списком сопоставляется третий образец:

```
[IS [SAME (X) (*X <PALYNDROM> .X)]
    (A B A)]
```

Первый и последний элементы анализируемого списка равны друг другу, поэтому данное сопоставление будет удачным, если удачно сопоставление сегментного образца <PALYNDROM> с сегментом из одного элемента B, а это сопоставление, согласно правилу десегментации, сводится к сопоставлению

```
[IS [PALYNDROM] (B)]
```

Снова вызывается сопоставитель PALYNDROM, но теперь его тело сопоставляется со списком (B):

```
[IS [AUT ( ) [LIST 1]
    [SAME (X) (*X <PALYNDROM> .X)]] (B)]
```

Поскольку список (B) соответствует второму образцу из AUT, то это сопоставление удачно. Следовательно, удачным является и сопоставление образца [PALYNDROM] со списком (B), а тем самым удачно и наше исходное сопоставление.

```
[DEFINE ET1 (KAPPA *ARGS [WHEN
    ([BE [EMPTY .ARGS]])
    ([PAT [1 .ARGS]]
        [PAT [FORM [ET1 <REST 1 .ARGS>]]])])]
```

Данный сопоставитель аналогичен встроенному сопоставителю ET. При обращении к нему можно задавать любое количество аргументов-образцов, список из которых присваивается параметру ARGS. Например, при обращении [ET1 [INT] [GE 1] [LE 9]] значением параметра станет список ([INT] [GE 1] [LE 9]). Если первый из заданных образцов соответствует анализируемому выражению, то ET1, чтобы проверить соответствие этого выражения и остальным образцам, строит с помощью функции FORM обращение к себе с остальными аргументами-образцами (в нашем примере — это обращение [ET1 [GE 1] [LE 9]]) и получившийся образец сопоставляет с анализируемым выражением. Если список ARGS пуст, а это признак того, что сопоставления всех

исходных образцов с анализируемым выражением оказались удачными, то сопоставление ET1 с выражением завершается, согласно первой клаузе условного сопоставителя WHEN, удачей.

2.8. Пример использования сопоставителей

Чтобы продемонстрировать возможности сопоставителей, опишем анализатор выражений языка лисп, который может служить дополнением к рассмотренной в § 1.19 программе, преобразующей лисповские выражения в эквивалентные плэнерские выражения при условии, что лисп-программа записана без ошибок.

При построении анализатора мы будем исходить из следующих предположений. Рассматривать будем то же подмножество лиспа, что и в § 1.19. На вход анализатору будет подаваться последовательность выражений, составляющих связную лисп-программу. Считаем, что файл, где записаны эти выражения, уже открыт и является активным файлом ввода и что функции вывода настроены на печать на АЦПУ или терминал. Анализатор по очереди считывает выражения лисп-программы, проверяет их и о каждой замеченной ошибке выдает сообщение на печать. Анализатор распознает не все ошибки; он, например, не может обнаружить неправильный баланс скобок или запрещенные нами точечные выражения, так как подобные конструкции недопустимы с точки зрения синтаксиса плэнера и попросту не могут быть считаны плэнерской функцией ввода. Ошибки, которые распознает анализатор,— это неправильное количество аргументов в обращениях к лисповским функциям, обращение к неописанным функциям и т. п.

В своей работе анализатор пользуется списками свойств плэнерских идентификаторов, являющихся названиями лисповских функций. Свойство NARG указывает на количество аргументов у соответствующей функции, а свойство TYPE указывает на класс функции — SUBR или FSUBR, при этом все определяемые лисп-функции мы будем относить к классу SUBR (напомним, что в нашем подмножестве лиспа все аргументы определяемых функций вызываются по значению). Для названий встроенных лисп-функций эти свойства задаются заранее, а для определяемых — при появлении в лисп-программе их описаний.

Переменные, которые использует анализатор, имеют следующий смысл. LV — это список лисп-переменных, к которым разрешено обращаться в анализируемой лисповской функции; мы будем предполагать, что в теле любой определяемой функции можно использовать только ее локальные переменные и запрещено использование внешних переменных. DE — это «флажок»,

который указывает, анализируется ли сейчас тело какой-нибудь определяемой функции (DE = T) или нет (DE = ()); по этому флажку анализатор узнает, допустимы ли сейчас обращения к еще не описанным функциям (что допускается в теле определяемых функций) или нет. UNDEF — список обращений как раз к таким неописанным функциям; анализ этих обращений откладывается на конец, когда станут известными все функции, определенные в лисп-программе. ERR — это флажок, который имеет значение (), пока в лисп-программе не обнаружено ошибок, и принимает значение T при первой же ошибке.

Ниже приводится описание нашего анализатора, снабженное краткими пояснениями.

```
[DEFINE ANALYSER (LAMBDA ()
   [PROG ((LV ()) (DE ()) (UNDEF ())
         (ERR ()) N)
      [PLIST QUOTE (TYPE FSUBR NARG 1)]
      [PLIST CAR (TYPE SUBR NARG 1)]
      [PLIST CDR (TYPE SUBR NARG 1)]
      [PLIST CONS (TYPE SUBR NARG 2)]
      [PLIST ATOM (TYPE SUBR NARG 1)]
      [PLIST EQ (TYPE SUBR NARG 2)]
      [WHILE [NOT [EOF]]
         [IS [L-FORM] [READ]]]
      [LOOP E .UNDEF [COND
         ([IS ([HAS TYPE SUBR NARG *N]
            <LIST .N>) .E])
         (T [PRINT-ERROR .E])]]
      .ERR])]
```

Это основная функция анализатора. Прежде всего она заносит в списки свойств названий встроенных лисп-функций информацию об их классе и числе аргументов. Для идентификаторов COND в SEXPR списки свойств не задаются, так как обращения к функциям с этими названиями будут анализироваться по особым правилам, без использования списков свойств. Затем в WHILE-цикле осуществляется считывание лисп-выражений и их проверка. В LOOP-цикле анализируются те обращения к лисп-функциям, которые были занесены в список UNDEF. В те моменты, когда анализатору встречались эти обращения, указанные в них функции еще не были определены, поэтому анализатор мог тогда проверять лишь правильность аргументов, заданных в этих обращениях. Теперь же он проверяет правильность числа аргументов и то, были ли в лисп-программе определены функции, указанные в этих обращениях.

Сообщения о любом неправильном лисп-выражении печатает следующая функция:

```
[DEFINE PRINT-ERROR (LAMBDA (E)
    [DO [MPRINT INCORRECT FORM: .E]
        [SET ERR T]])]
```

Все остальные процедуры нашего анализатора — это сопоставители, проверяющие правильность записи различных типов лисповских выражений.

```
[DEFINE L-FORM (KAPPA ( )
    [SAME (E) *E
        [AUT [L-CONST] [L-VAR]
            [FUNCALL] [LAMBDA-CALL]
            [BE [PRINT-ERROR .E]]] ])]
```

Лисп-форма — это или константа, или переменная, или обращение к функции, или особое обращение с LAMBDA-выражением. Если анализируемое выражение — не форма, то об этом сообщается на печать.

```
[DEFINE L-CONST (KAPPA ( )
    [AUT T NIL ( ) [NUM]])]
```

Лисп-константа — это T, NIL, ( ) или число.

```
[DEFINE L-VAR (KAPPA ( )
    [ET [ID] [ONE-OF .LV]])]
```

Переменной (согласно нашим предположениям) является идентификатор из текущего списка LV.

```
[DEFINE FUNCALL (KAPPA ( ) [SAME (N)
    ([ID] <LIST *N>)
    [WHEN
        ((COND < >) [LIST [GE 2]]
            ([ ] <STAR ([L-FORM] [L-FORM])>))
        ((SEXPR < >) [NEWFUN])
        (([HAS TYPE SUBR] < >)
            ([HAS NARG .N] <STAR [L-FORM]>))
        (([HAS TYPE FSUBR] < >)
            ([HAS NARG .N] < >))
        ([BE DE] *N
            ([ ] <STAR [L-FORM]>)
            [BE [SET UNDEF (!.UNDEF .N)]])]])]
```

Обращение к функции должно быть списком в круглых скобках,

начинающимся с идентификатора. Обращение к функции COND должно содержать хотя бы один аргумент, и все аргументы обязаны быть списками из двух форм. Обращение к функции SEXPR анализируется с помощью сопоставителя NEWFUN. Для обращений к другим функциям, встроенным или уже определенным, проверяется правильность числа заданных аргументов, а для функций класса SUBR проверяется и то, являются ли их аргументы формами. Если же встретилось обращение к неописанной функции и мы сейчас находимся внутри тела некоторой определяемой функции, т. е. DE = T, тогда проверяется, являются ли аргументы этого обращения формами, после чего обращение заносится в список UNDEF. Если DE = ( ), то обращение к неописанной функции недопустимо, и такому обращению сопоставитель FUNCALL не соответствует.

```
[DEFINE NEWFUN (KAPPA ( ) [SAME (F N)
    (SEXPR [ET [ID] *F] [DEXPR *N])
    [BE [PLIST .F (TYPE SUBR NARG .N)]]])]
```

Этот сопоставитель проверяет правильность обращений к функции SEXPR. Проверив, что в анализируемом обращении первый аргумент является идентификатором, а второй — правильным определяющим выражением, сопоставитель заносит в список свойств этого идентификатора — имени определяемой функции — число параметров функции (оно узнается при выполнении сопоставителя DEXPR) и признак того, что функция относится к классу SUBR.

```
[DEFINE DEXPR (KAPPA (*P) [SAME (LV (DE T))
    (LAMBDA [ET [STAR [ID]] *LV] [L-FORM])
    ([ ] [LIST [PAT .P]] [ ])])]
```

Проверяя, правильно ли построено определяющее выражение функции, сопоставитель DEXPR вводит локальные переменные LV и DE, которые будут использоваться сопоставителем L-FORM при анализе тела этого определяющего выражения. DE получает значение T, а значением LV становится список параметров из определяющего выражения. Этим сообщается сопоставителю L-FORM, что он работает внутри определяющего выражения и что здесь допустимы обращения только к переменным, являющимся параметрами анализируемой функции. Если определяющее выражение записано правильно, то образец, заданный как аргумент сопоставителя DEXPR, сопоставляется с числом, равным длине списка параметров.

```
[DEFINE LAMBDA-CALL (KAPPA ( ) [SAME (K)
    ([DEXPR*K] <ET [LIST .K]
                [STAR [L-FORM]]>)])]
```

В обращении с LAMBDA-выражением число аргументов должно равняться числу параметров, описанных в LAMBDA-выражении, и все эти аргументы должны быть формами.

На этом завершается описание анализатора. Запуск его осуществляется обращением к функции ANALYSER:

[ANALYSER].

Значение этого выражения равно ( ), если в анализируемой лисп-программе не было обнаружено ошибок, и равно T, если была найдена хотя бы одна ошибка.

ГЛАВА 3

РЕЖИМ ВОЗВРАТОВ

3.1. Основные понятия

Многие задачи обработки символьной информации и искусственного интеллекта (например, символьное интегрирование, синтаксический анализ, доказательство теорем, планирование действий роботов, игровые задачи) решаются методом перебора. Эти задачи не имеют эффективного алгоритма решения, и единственная возможность решить их — это рассмотреть различные варианты, которые потенциально могут быть решениями, в надежде найти вариант, который действительно решает задачу. Конечно, при этом стремятся с помощью эвристик сократить число рассматриваемых вариантов, но в основе такого способа решения лежит все же перебор вариантов.

Существует несколько разновидностей метода перебора. Наиболее часто на практике применяется перебор в глубину, когда в каждой точке ветвления выбирается один из вариантов и он исследуется до конца. Если выбранный путь не приводит к успеху, то осуществляется возврат к ближайшей точке ветвления, в которой еще остались нерассмотренные альтернативы, здесь выбирается новый вариант, и теперь исследуется уже он (см. рис. 1). Для того чтобы упростить программирование такого метода перебора, в язык плэнер введен специальный способ выполнения программ — *режим возвратов* (backtracking). Суть его в следующем.

В любой плэнерской программе могут быть *развилки* — точки, где необходимо сделать выбор из нескольких *альтернатив*. Программа выбирает в развилке одну из альтернатив и продолжает свои вычисления. В какой-то момент может быть установлено, что сделанный выбор неудачен, тогда вырабатывается *неуспех* — специальный сигнал, по которому программа прекращает обработку выбранной альтернативы и возвращается назад к развилке. Одновременно уничтожаются следы работы программы на неуспешном пути вычислений; восстанавливаются прежние значения перемен-

ных, прежние значения списков свойств и т. п. Таким образом, при появлении неуспеха восстанавливается то состояние программы, в котором она находилась, когда делала первый выбор в развилке, как будто бы и не было этого выбора. Теперь в развилке выбирается другая альтернатива, после чего программа возобновляет свои вычисления — анализирует новую альтернативу. Если и она не приводит к успеху, то снова происходит возврат к развилке, где выбирается следующая альтернатива, и т. д. Когда в развилке не останется нерассмотренных альтернатив, программа при очередном неуспехе возвращается уже к предыдущей развилке (в программе может быть много развилок), чтобы теперь здесь изменить ранее сделанный выбор.

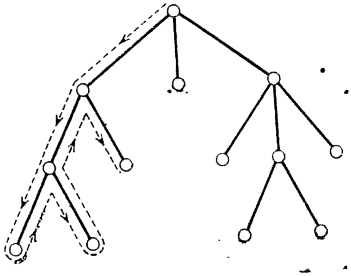


Рис. 1.

Описанный способ выполнения программы, при котором определяются развилки, в них выбираются альтернативы, а в случае неуспеха осуществляются возвраты назад, и есть режим возвратов. Удобство его для программистов заключается в том, что ответственность за запоминание развилок и их альтернатив, за осуществление возвратов к ним с восстановлением прежнего состояния программы берет на себя язык — это делается автоматически. Задача же пользователя — в нужных местах определять нужные развилки и в соответствующие моменты сигнализировать о неуспехе. Можно сказать, что режим возвратов — это «рамка», вставляя в которую подходящие операции, можно легко получить конкретные алгоритмы перебора.

Для иллюстрации режима возвратов рассмотрим работу следующей функции:

```
[DEFINE SUM (LAMBDA (L N)
   [PROG (K (M ()) (S 0))
   A [SET K  [AMONG .L]]
      [SET M (!.M .K)]
      [SET S [+ .S .K]]
      [COND ([EQ .S .N] .M)
            ([LT .S .N] [GO A])
            (T [FAIL])]])]
```

Эта функция находит такие числа из заданного списка L положительных чисел, сумма которых равна заданному положительному числу N (числа в искомом наборе могут повторяться). Найденные числа функция накапливает в M, а их сумму — в S.

Функция SUM вычисляется следующим образом. На каждом шаге цикла она подбирает очередное число для набора. Им может быть любое число из списка L, поэтому выбор его — развилка в программе. Такую развилку определяет специальная функция AMONG, которая в качестве своего значения может выдать любое число из списка L. То число, которое она действительно выдала, запоминается в M и прибавляется к S. Далее S сравнивается с N. Если S и N равны, то нужный набор уже найден (он в M), и в этом случае функция SUM заканчивает свою работу. Если же S меньше N, то чисел в наборе еще недостаточно, поэтому подыскивание чисел продолжается: осуществляется переход на следующий шаг цикла, где определяется новая развилка и т. д.

Но если S оказалось больше N, то продолжать вычисление функции SUM нельзя, так как она зашла в тупик: сумма выбранных чисел оказалась слишком большой. Причиной этого может быть то, что на текущем шаге цикла из списка L было выбрано неподходящее число, поэтому надо изменить прежний выбор и рассмотреть другое число. Для этого с помощью специальной функции FAIL вырабатывается сигнал о неуспехе, по которому программа автоматически возвращается к своей последней развилке — к функции AMONG с текущего шага цикла. При возврате также автоматически восстанавливаются прежние значения переменных M и S — те значения, которые они имели до выбора отвергнутого числа.

Итак, неудачный выбор забыт. Функция AMONG, хотя она ранее и закончила свою работу, вновь «оживает» и теперь в качестве своего значения выдает какое-то другое число из списка L. С этого момента выполнение программы возобновляется: она повторяет текущий шаг цикла, но уже для нового числа. Если и оно не подойдет, тогда программа вновь будет возвращена назад к функции AMONG, которая выберет из списка L третье число. Такие возвраты будут повторяться до тех пор, пока сумма S не окажется меньшей или равной числу N или пока функция AMONG не исчерпает всех своих возможностей. В последнем случае очередной неуспех вернет программу к предпоследней развилке — к AMONG с предыдущего шага цикла, чтобы теперь именно здесь исправить ранее сделанный выбор; при этом будут автоматически восстановлены соответствующие значения переменных M и S.

На рис. 2 схематично показано вычисление выражения [SUM (6 3 2 1) 5]. На этом рисунке пунктирными линиями обозначены неуспешные альтернативы, сплошными — успешный путь вычисления, волнистыми — оставшиеся не рассмотренными альтернативы.

Опираясь на наш пример, уточним некоторые особенности режима возвратов.

Прежде всего отметим, что возвраты по неуспеху не имеют ничего общего с переходами назад. При переходах программа динамически продвигается вперед. Так, в нашем примере после перехода по метке A осуществляется новый вызов функции AMONG, которая начинает новый, независимый от выборов в других активациях этой же функции, выбор элементов из списка L. При неуспехе же действительно происходит возврат к прежней активации функции AMONG, которая «знает», какие альтернативы ею уже рассматривались, а какие — еще нет, и поэтому, возобновив свою работу, она выбирает следующую альтернативу. Кроме того, при переходах текущие значения переменных сохраняются, тогда как при возвратах по неуспеху они уничтожаются и восстанавливаются прежние значения.

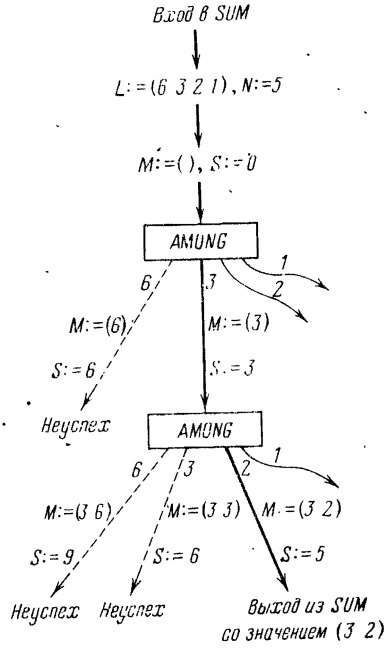


Рис. 2.

Следует отличать неуспехи от ошибок. Ошибка — это нарушение правил языка, при котором выполнение программы прекращается, в то время как появление неуспеха означает, что программа работает правильно, но из нескольких возможных вариантов своего вычисления она рассматривает не тот вариант, который нам нужен.

В любой развилке каждая альтернатива выбирается только раз, и ранее сделанный выбор никогда больше не повторяется. В частности, функция AMONG каждый раз выбирает из списка-аргумента новый элемент. Порядок выбора альтернатив всегда детерминирован. Он, как и сам набор альтернатив в развилке, строго фиксирован: либо указывается пользователем, либо определяется правилами языка и текущим состоянием программы. Например, функция AMONG в качестве своего первого значения всегда выдает первый элемент из списка-аргумента, в качестве второго значения — второй элемент и т. д. Никакого произвола, никакого бросания жребия при выборе альтернатив нет.

При появлении неуспеха программа всегда возвращается к последней (по времени) из существующих развилок. Развилка пе-

рестает существовать лишь тогда, когда в ней не осталось ни одной нерассмотренной альтернативы. В нашем примере развилка, определенная функцией AMONG, уничтожается тогда, когда функция в качестве своего очередного значения выдает последний элемент списка L. С этого момента последней развилкой программы становится развилка, которую определила функция AMONG с предыдущего шага цикла, поэтому именно к ней и произойдет возврат при следующем неуспехе.

В то же время любая развилка, в которой осталась хотя бы одна нерассмотренная альтернатива, продолжает существовать, даже если закончилось вычисление процедур, внутри которых она была определена. Из рис. 2 видно, что по окончании вычисления выражения [SUM (6 3 2 1) 5] внутри функции SUM остались развилки с нерассмотренными альтернативами, поэтому они продолжают существовать и после выхода из функции SUM. Это означает, что если по окончании вычисления функции SUM выработать неуспех, то он вернет программу к последней из этих развилок — к AMONG со второго шага цикла. При этом автоматически возобновляется работа функции SUM, восстанавливается существование всех ее локальных переменных с соответствующими значениями, возобновляется и работа функции AMONG со второго шага цикла. На этот раз она выберет число 1, которое и будет теперь анализироваться функцией SUM (см. рис. 3). Новым значением функции SUM будет список (3 1 1), являющийся другим решением нашей задачи.

Вырабатывая снова и снова неуспехи, можно заставить функцию SUM найти третье, четвертое и другие решения нашей задачи. Например, при вычислении выражения

```
[PROG (X) [SET X [SUM (6 3 2 1) 5]]
    [COND ([NEQ [LENGTH .X] 4] [FAIL])]
    .X]  →  (2 1 1 1)
```

неуспехи, вырабатываемые функцией FAIL из условного оператора, будут возобновлять работу функции SUM до тех пор, пока она не найдет решение из четырех чисел. Этот пример показывает, что неуспех можно использовать не только как сигнал о том, что программа зашла в тупик, но и для того чтобы заставить некоторую процедуру генерировать все или нужное количество вариантов ее успешной работы.

Рассматриваемая нами задача может и не иметь решения, как, например, в случае [SUM (4 2) 7]. Здесь в каждой развилке функции SUM будут рассмотрены все альтернативы, но ни одна из них не приведет к успеху. В конце концов в SUM не останется ни одной развилки, поэтому неуспех, выбранный в очередной раз функцией FAIL, заставит программу вернуться к развилке,

которая была определена до входа в функцию SUM. Тем самым мы стадкиваемся с новой для нас ситуацией. До сих пор, вычисляя какую-нибудь форму, мы обязательно получали некоторое значение. При вычислении же формы [SUM (4 2) 7] ни о каком ее значении не может быть и речи, поскольку вычисление функции

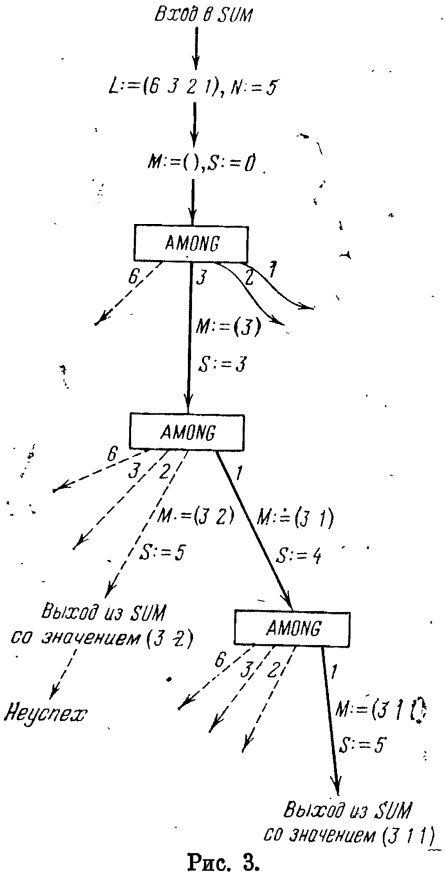


Рис. 3.

SUM не доходит до конца и происходит «выскакивание» из функции по неуспеху.

Таким образом, в плэнере наряду с обычным *успешным* исходом вычисления, когда оно полностью завершается и в результате вырабатывается некоторое значение, возможен и иной исход — *неуспешный*. Вычисление выражения называется неуспешным, если во время этого вычисления был выработан неуспех, по которому программа должна вернуться к развилке, определенной

до начала вычисления данного выражения. Значения такое выражение не имеет.

Следовательно, вычисление выражения [SUM (4 2) 7] неуспешно. Неуспешным является и вычисление ранее рассмотренного условного выражения

[COND ([NEQ [LENGTH .X] 4] [FAIL])]

в том случае, если длина списка X не равна 4. Другие примеры неуспешных вычислений: [SET X (A [FAIL])], [GO [FAIL]].

3.2. Функции для режима возвратов

Рассмотрим встроенные функции плэнера, которые используются для организации вычислений в режиме возвратов.

Функция FAIL *): [FAIL *e*?], SUBR.

Данная функция, как уже известно, вырабатывает неуспех, который прерывает выполнение программы и возвращает ее к последней развилке, восстанавливая при этом прежние состояния объектов программы (переменных, списков свойств и т. д.). Кроме того, функция связывает с этим неуспехом *сообщение*; им является значение аргумента функции, а если его нет — пустой список ( ).

В плэнере с любым неуспехом связывается некоторое сообщение. Сообщения позволяют идентифицировать неуспехи: большинство встроенных функций языка, которые вырабатывают неуспех, в качестве сообщения указывают свое имя, и по таким сообщениям можно установить, какая именно функция выработала неуспех. Сообщения можно использовать и для указания причины появления неуспеха; в этом случае обычно применяется функция FAIL, которая позволяет связать с неуспехом произвольное выражение. Воспользоваться сообщениями можно в развилке после возврата к ней по неуспеху: в зависимости от того, какое сообщение было связано с неуспехом, можно по-разному прореагировать на этот возврат.

Узнать сообщение, связанное с неуспехом, можно с помощью следующей встроенной функции.

Функция MESS: [MESS].

Значением этой функции всегда является сообщение, связанное с последним неуспехом. Пока в программе не будет выработан новый неуспех, значение функции не меняется. При появлении же нового неуспеха с ним связывается иное сообщение, которое и становится новым значением функции MESS.

Функция AMONG: [AMONG *l*], SUBR.

---

*) См. также § 3.7.

Значением аргумента данной функции должен быть список. Если этот список пуст, то функция вырабатывает неуспех, связывая с ним сообщение AMONG. В противном случае функция определяет развилку, i-й альтернативой которой является выбор i-го элемента из списка и выдача его в качестве значения функции. При выборе из списка последнего элемента развилка функции AMONG уничтожается.

Функция ALT: [ALT e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Функция ALT также определяет развилку, но в данном случае i-й альтернативой является вычисление ее i-го аргумента, т. е. простой формы e_i. Таким образом, функция сначала вычисляет форму e_1. Если это вычисление успешно, то функция заканчивает свою работу со значением этой формы. Но если в дальнейшем в программе будет выработан неуспех, по которому произойдет возврат к развилке функции ALT, или если вычисление формы e_1 было неуспешным (в этом случае также произойдет возврат к развилке функции ALT), тогда функция переходит к вычислению своего второго аргумента e_2, и описанные действия повторяются. Перед вычислением последнего аргумента развилка функции ALT уничтожается.

К примеру, вычисление выражения

```
[ALT [GO L]
     [COND ([NEQ [MESS] B] [SET X [MESS]])
           (T [FAIL])]
     [RETURN ()]]
```

происходит так. В первый раз выполняется переход по метке L, и на этом вычисление функции ALT заканчивается. Если затем в программе вырабатывается неуспех, по которому происходит возврат к функции ALT, то она возобновляет свою работу и теперь вычисляет свой второй аргумент. В этом условном выражении анализируется сообщение неуспеха. Если оно отлично от атома B, то оно присваивается переменной X, после чего функция ALT снова завершает свою работу. При сообщении B вычисление условного выражения неуспешно, поэтому функция переходит к вычислению своего последнего аргумента, т. е. осуществляет выход из ближайшего объемлющего блока. Последний аргумент будет вычисляться и в том случае, если после успешного вычисления второго аргумента функции ALT было продолжено выполнение программы, но затем был выработан неуспех, по которому произошел возврат к функции ALT.

При вычислении следующего выражения

```
[DO [ALT ( ) [EXIT T DO]]
    [PRINT [SUM (6 3 2 1) 5]]
    [FAIL]]
```

будут напечатаны все решения задачи из предыдущего параграфа. Первая альтернатива функции ALT — вычисление пустого списка — просто «пропускает» программу к следующим двум операторам, которые, взаимодействуя друг с другом, и напечатают все решения задачи. Когда в функции SUM уже не останется развилок, очередной неуспех вернет программу к развилке функции ALT (она и была поставлена для «ловли» этого неуспеха), и эта функция осуществит, согласно своему второму аргументу, выход из функции DO со значением T. Таким образом, вычисление нашего выражения всегда будет успешным; хотя внутри него неоднократно вырабатывались неуспехи, наружу неуспех не выйдет.

Следующие две функции преобразуют неуспех в «ложь» и наоборот.

Функция GATE: [GATE e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Эта функция преобразует неуспех в «ложь». Действие функции полностью эквивалентно вычислению выражения

[ALT [DO e_1 e_2 ... e_k] ()]

Таким образом, функция GATE последовательно вычисляет свои аргументы. Если при этом успешно завершается вычисление последнего аргумента e_k, то функция заканчивает свою работу со значением этого аргумента. Но если вычисление аргументов оказалось неуспешным, то функция заканчивает свою работу со значением ().

Например, вычисление выражения

[GATE [SUM (5 4 17) 203]]

всегда успешно, причем его значением является либо «ложь», если из чисел 5, 4 и 17 нельзя составить сумму 203, либо «истина», если такая задача имеет решение (более точно, значением выражения в последнем случае будет одно из решений задачи).

Функция UNFALSE: [UNFALSE e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Данная функция преобразует «ложь» в неуспех, с которым связывается сообщение UNFALSE. Действие функции полностью эквивалентно вычислению выражения

```
[COND ([DO e1 e2 ... ek])
      (T [FAIL UNFALSE])]
```

Следовательно, функция UNFALSE по очереди вычисляет свои аргументы. Если это вычисление неуспешно, то неуспешным будет и вычисление функции. В противном случае она анализирует значение своего последнего аргумента: если это значение отлично от (), то оно же объявляется значением функции, а если оно равно (), тогда функция сама вырабатывает неуспех.

Например, при вычислении выражения

```
[PROG (X) [SET X [SUM (5 4 17) 203]]
          [UNFALSE [IS (< > 17 < >) .X]]
          .X]
```

будет найдено решение задачи из предыдущего параграфа, в которое входит число 17. Если такого решения нет или если задача вообще не имеет решений, вычисление данного выражения неуспешно.

3.3. Примеры

Используем функции, описанные в предыдущем параграфе, для решения двух задач методом перебора.

Первая из них — известная задача о восьми ферзях. В этой задаче требуется так расставить на шахматной доске восемь ферзей, чтобы они не били друг друга. Напомним, что ферзь бьет все поля своей горизонтали и своей вертикали, а также двух диагоналей, проходящих через его поле.

Решение данной задачи можно описать в виде следующей функции Ф8, которая использует вспомогательную функцию ДИАГ:

```
[DEFINE Ф8 (LAMBDA ( )
    [PROG (V FV (LV ( )) B E)
        [SET FV (1 2 3 4 5 6 7 8)]
        [FOR H 8 [SET V [AMONG .FV]]
            [ДИАГ .V .LV]
            [SET LV (!.LV .V)]
            [IS (!*B .V !*E)  .FV]
            [SET FV (!.B !.E)]]
        .LV])]
[DEFINE ДИАГ (LAMBDA (V LV)
    [PROG (H (H1 0))
        [SET H [+ [LENGTH .LV] 1]]
        [LOOP V1 .LV [ADD1 H1]
            [UNFALSE [NEQ [ABS [— .H .H1]]
                          [ABS [— .V .V1]]]]]])]
```

Из условий задачи вытекает, что в искомой расстановке ферзей на каждой горизонтали и вертикали шахматной доски должен находиться только один ферзь. В связи с этим функция Ф8 организует цикл, на каждом шаге которого она пытается поставить очередного ферзя на очередную горизонталь H (H = 1, 2, ..., 8). Попутно функция следит за тем, на какие вертикали уже поставлены ферзи, и хранит в списке FV номера вертикалей, которые

пока свободны. Поскольку заранее неизвестно, на какую из свободных вертикалей следует поставить очередного ферзя, функция выбирает любой номер V из списка FV и ставит этого ферзя на вертикаль V. Далее с помощью функции ДИАГ проверяется, не попал ли этот ферзь на одну диагональ с каким-нибудь из предыдущих ферзей. Если попал, то функция ДИАГ вырабатывает неуспех, который заставляет программу выбрать иную вертикаль для ферзя. Причем, если такой выбор невозможен (все поля горизонтали H находятся под ударом ранее расставленных ферзей), программа делает шаг назад и пытается изменить положение ферзя на предыдущей горизонтали. Но если позиция ферзя на горизонтали H выбрана удачно, то номер V его вертикали заносится в список LV, где функция Ф8 запоминает номера вертикалей уже расставленных ферзей, и исключается из списка FV свободных вертикалей.

Вычисление функции Ф8 заканчивается тогда, когда ей удалось поставить ферзя на восьмую горизонталь. Значением функции является список $(v_1\ v_2\ \ldots\ v_8)$, где v_i — номер вертикали, в которую поставлен ферзь на i-й горизонтали. Например, первым значением функции Ф8 будет список (1 5 8 6 3 7 2 4).

Используя функцию Ф8, можно получить различные решения задачи о восьми ферзях. Например, вывести на печать все 92 решения этой задачи можно так:

[GATE [PRINT [Ф8]] [FAIL]].

Другая задача, которую мы рассмотрим, связана с поиском пути в графе. Предположим, что имеется ориентированный граф, на каждую дугу которого навешаны некоторый предикат и некоторый оператор (то и другое — плэнерские формы). Предположим также, что переход по любой дуге графа возможен только тогда, когда ее предикат имеет значение «истина», и что при переходе по дуге должен быть выполнен ее оператор. При этих предположениях требуется определить, существует ли путь от одного заданного узла графа к другому.

Один из таких графов показан на рис. 4. Если считать, что начальным значением переменной L является список из целых чисел, тогда существование пути в этом графе от узла A к узлу C зависит от того, можно ли между числами этого списка так расставить знаки + и —, чтобы в результате получилось выражение с нулевым значением. Например, для списка (5 2 4 7) такой путь существует, так как 5—2+4—7=0, а для списка (6 3) — нет.

Граф будем представлять в виде списка, каждый элемент которого соответствует одному из узлов графа и имеет следующую структуру:

$(N\ (P_1\ S_1\ N_1)\ (P_2\ S_2\ N_2)\ \ldots\ (P_k\ S_k\ N_k))$,

где N — название узла, а $(P_i\ S_i\ N_i)$ обозначает дугу от узла N к узлу N_i, на которую навешены предикат P_i и оператор S_i. Считаем, что в этом элементе списка перечислены все дуги, выходящие из узла N.

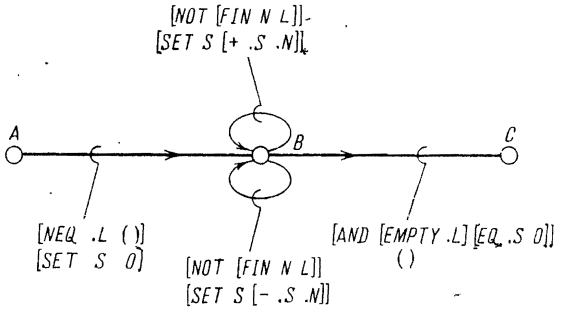


Рис. 4.

Например, граф, изображенный на рис. 4, представляется списком:

```
((A ([NEQ .L ( )] [SET S 0] B))
(B ([NOT [FIN N L]] [SET S [+ .S .N]] B)
   ([AND [EMPTY .L] [EQ .S 0]] ( ) C)
   ([NOT [FIN N L]] [SET S [— .S .N]] B))
(C))
```

Считая, что подобный список является значением константы GRAPH, опишем функцию, которая определяет, существует ли путь от узла BEG⁻ к узлу END⁻:

```
[DEFINE PATH (LAMBDA (BEG⁻ END⁻)
   [PROG (ARCS⁻ PRED⁻ SENT⁻ NODE⁻)
      [COND ([EQ .BEG⁻ .END⁻] [RETURN T])]
        [IS (< > (.BEG⁻ !*ARCS⁻) < >) :GRAPH]
      [IS (*PRED⁻ *SENT⁻ *NODE⁻)
          [AMONG .ARCS⁻]]
      [COND ([EVAL .PRED⁻]
                 [EVAL .SENT⁻]
                 [PATH .NODE⁻ .END⁻])
            (T [FAIL])]]]
```

Мы ввели несколько необычные названия для переменных функции PATH, чтобы они отличались от названий переменных из предикатов и операторов графа (предполагается, что имена переменных из графа не оканчиваются символом надчеркивания). Эти группы переменных действуют одновременно, и они не должны «мешать» друг другу.

Вычисление функции PATH успешно, если она нашла путь от узла BEG⁻ к узлу END⁻, и неуспешно, если такой путь не существует. Вначале функция проверяет, не совпадают ли начальный и конечный узлы. Если совпадают, то функция сразу заканчивает свою работу со значением T. Иначе в списке GRAPH отыскивается подсписок, относящийся к узлу BEG⁻, после чего в программе определяется развилка, альтернативы которой соответствуют дугам, выходящим из этого узла. В развилке выбирается одна из этих дуг и вычисляется ее предикат. Если он имеет значение «ложь», тогда переход по дуге запрещен, поэтому вырабатывается неуспех, который заставляет программу выбрать иную дугу. Если же предикат имеет значение «истина», тогда переход по дуге возможен, поэтому выполняется ее оператор. Далее происходит рекурсивное обращение к функции PATH, для того чтобы определить, существует ли путь к узлу END⁻ от узла NODE⁻, которым оканчивается выбранная дуга. Если такой путь существует, вычисление рекурсивного обращения успешно, поэтому успешно заканчивается и вычисление текущей активации функции PATH. Но если пути от NODE⁻ к END⁻ нет, вычисление рекурсивного обращения неуспешно. Выработанный при этом неуспех заставляет программу выбрать иную дугу, выходящую из узла BEG⁻, после чего описанные действия повторяются, но уже по отношению к новой дуге.

Подчеркнем, что при возвратах по неуспеху восстанавливаются прежние значения не только переменных функции PATH, но и переменных, используемых в предикатах и операторах графа. Этим уничтожаются все следы работы операторов графа на неудачно выбранном пути, что обеспечивает возможность правильного продолжения поиска. Без режима возвратов было бы крайне трудно восстанавливать значения этих переменных, так как ни их названия, ни сами операторы графа заранее неизвестны.

Отметим также, что метод перебора, использованный в функции PATH, точно такой же, как и в ранее рассмотренных функциях SUM и Ф8. Но есть и отличие: в функции PATH перебор организован не с помощью возвратов по неуспеху и циклов, а с помощью возвратов и рекурсий.

3.4. Управление режимом возвратов

До сих пор мы рассматривали режим возвратов, так сказать, в чистом виде: при появлении в программе неуспеха она всегда возвращалась к последней развилке, где полностью восстанавливалась прежняя обстановка (прежние значения переменных, списков свойств и т. д.). Но это не всегда удобно. Иногда нужен воз-

врат не к последней развилке, а к одной из предыдущих, и нередко необходимо только частичное восстановление обстановки в развилке. Например, в какой-то момент может быть обнаружено, что ответственность за неуспех вычислений несет одна из предыдущих развилок и что никакой выбор в последующих развилках не приведет к успеху. Чтобы не вести бессмысленный перебор в последних развилках, надо сразу вернуться к этой предыдущей развилке и исправить выбор именно здесь. Может быть и так, что при исследовании альтернативы, оказавшейся неуспешной, была получена информация, представляющая интерес и для других альтернатив. Чтобы не уничтожать такую информацию, следует при возврате по неуспеху сохранить у переменных и других объектов программы те значения, которые они получили при исследовании этой альтернативы, а не восстанавливать их прежние значения, существовавшие до выбора альтернативы. Для выполнения подобных действий, т. е. для управления режимом возвратов, в язык введены специальные средства. Но прежде чем рассказать о них, мы уточним, каким образом в языке осуществляются возвраты по неуспеху и как восстанавливаются прежние значения переменных и других объектов программы.

Выполнение программы в режиме возвратов можно сравнить с анализом шахматной позиции, когда делается несколько ходов вперед и затем оценивается получившаяся позиция, после чего восстанавливается исходная позиция и рассматривается другая последовательность ходов. Каким образом можно восстановить исходную позицию, когда было сделано несколько ходов вперед? Можно поступить так: один раз записать расположение фигур в исходной позиции, а затем каждый раз по этой записи восстанавливать данную позицию. Но можно поступить иначе: запоминать сделанные ходы, а затем, выполняя обратные ходы, постепенно восстанавливать исходную позицию.

Точно так же существует и два способа организации возвратов по неуспеху. В первом из них при появлении развилки запоминается копия текущего состояния программы, а при появлении неуспеха состояние программы восстанавливается по последней из копий, после чего вычисления возобновляются от этого состояния. Именно такая трактовка режима возвратов использовалась нами в предыдущих параграфах. Однако для плэнера более точной является иная трактовка — с «обратными ходами».

Считается, что попутно с выполнением любой плэнерской программы запоминается некоторая информация о ее действиях. Так, при появлении в программе развилки запоминается это место программы (будем говорить: ставится *F-точка*) и запоминается, какие альтернативы и в каком порядке должны здесь выбираться. Когда же изменяется значение какого-нибудь объекта програм-

мы, запоминается *обратный оператор,* предназначенный для отмены произведенного действия. Например, если переменная X имела значение A и было выполнено [SET X B], то запоминается обратный оператор [SET X A]; если же переменная Y не имела значения, а переменная Z имела значение C и было выполнено [IS (.Y *Z) (D E)], то запоминается два обратных оператора: [UNASSIGN Y] и [SET Z C]. Вот такие F-точки и обратные операторы и запоминаются в соответствующем порядке одновременно с «прямым» выполнением программы.

При появлении в программе неуспеха ее «нормальное» выполнение прекращается и начинается *распространение неуспеха* — как бы обратное вычисление ее: выполняются обратные операторы в порядке, обратном их запоминанию. Тем самым шаг за шагом ликвидируются ранее произведенные изменения в значениях объектов программы. Распространение неуспеха прекращается, когда при таком «попятном» движении встретится первая F-точка (она соответствует последней развилке программы). F-точка — это «преграда» для неуспеха, она останавливает, «ловит» неуспех *). Следовательно, при распространении неуспеха выполняются только те операторы, которые появились после того, как была поставлена последняя F-точка.

Осуществив таким образом возврат к последней развилке и восстановление прежних значений своих объектов, программа возвращается в «нормальное» состояние, выбирает первую из оставшихся альтернатив развилки и возобновляет от этой точки свои вычисления. Когда в развилке выбирается последняя альтернатива, соответствующая F-точка уничтожается («забывается»), поэтому следующий неуспех будет распространяться уже до предыдущей F-точки.

Именно такой интерпретации выполнения программ в режиме возвратов мы и будем придерживаться в дальнейшем. В частности, воспользуемся ею, чтобы объяснить способы управления этим режимом.

Возможность возвратов к произвольным развилкам программы обеспечивается уничтожением ранее поставленных F-точек. Действительно, при отмене F-точки в соответствующем месте программы снимается «преграда» для неуспеха. Поэтому, если уничтожить несколько последних F-точек программы, а затем выработать обычный неуспех, то он сразу распространится до F-точки, предшествующей уничтоженным.

---

*) Развилку можно рассматривать как совокупность F-точки, которая останавливает распространение неуспеха, и генератора альтернатив, который выдает по одной альтернативе при каждом обращении к нему.

В основе же частичного восстановления обстановки в развилке лежит уничтожение ранее запомненных обратных операторов, а также выполнение действий без одновременного запоминания обратных операторов. В самом деле, отменить при неуспехе побочные эффекты могут только обратные операторы, а в обоих указанных случаях таких операторов не остается. Поэтому при неуспехе соответствующие действия программы не будут отменены, т. е. не будут восстановлены прежние значения каких-то объектов программы и будут сохранены их последние значения.

Встроенные функции языка, позволяющие уничтожать F-точки и обратные операторы и изменять значения объектов программы без запоминания обратных операторов, описываются в следующих параграфах.

3.5. Неотменяемые действия

В этом параграфе рассматриваются функции с побочным эффектом, при вычислении которых обратные операторы не запоминаются. Действия таких функций, следовательно, не отменяются при неуспехе.

Подобные функции не столь уж редки в плэпере. Из числа ранее рассмотренных встроенных функций к ним относятся функции DEFINE, PLIST, READ, PRINT, MPRINT, OPEN, CLOSE, ACTIVE и DIGITS. Таким образом, при возвратах по неуспеху не отменяется: ни определение новой процедуры, ни полная смена списка свойств идентификатора, ни ранее сделанный ввод или вывод, ни открытие или закрытие файла, ни объявление его активным файлом, ни ранее установленное количество печатаемых цифр в дробной части вещественных чисел.

В то же время при вычислении встроенных функций SET, ADD1, SUB1, FIN, UNASSIGN, LOOP, FOR, PUT, CSET и IS обратные операторы запоминаются всегда, поэтому при неуспехе произведенные этими функциями изменения ликвидируются:

обратные операторы функций SET, ADD1, SUB1, FIN и UNASSIGN, изменяющих и уничтожающих значения переменных, восстанавливают при неуспехе прежние значения переменных;

при выполнении функций LOOP и FOR обратные операторы запоминаются всякий раз, когда параметру цикла присваивается новое значение, поэтому при возврате по неуспеху на какой-то из предыдущих шагов цикла у параметра всегда восстанавливается значение, соответствующее именно этому шагу цикла;

обратный оператор функции PUT, которая изменяет значение одного из свойств некоторого идентификатора, восстанавливает при неуспехе прежнее значение этого свойства;

обратный оператор функции CSET либо восстанавливает прежнее значение константы, измененное этой функцией, либо уничтожает константу, если она была введена в программу именно этой функцией CSET (однако обратный оператор не восстанавливает ту процедуру, которая, возможно, была уничтожена при определении константы);

при вычислении функции IS, т. е. при сопоставлении образца с выражением, обратные операторы запоминаются всякий раз, когда изменяется значение какой-нибудь переменной образца, поэтому при распространении неуспеха все присваивания переменным образца будут отменены (см. также § 3.9).

Таким образом, любое изменение в значениях переменных, констант или свойств идентификаторов отменяется при неуспехе. Это не всегда выгодно, поэтому для некоторых из перечисленных функций введены «двойники» — функции с аналогичными действиями, но при вычислении которых обратные операторы не запоминаются и, значит, побочный эффект которых не уничтожается при неуспехе. Названия функций-двойников образуются добавлением буквы P (от слова permanent — постоянный) к именам исходных функций:

[PSET *i e*], SUBR
[PADD1 *i*], SUBR
[PSUB1 *i*], SUBR
[PFIN i_1 i_2], SUBR
[PPUT *i ind v*], SUBR
[PCSET *i e*], SUBR

(У функций UNASSIGN, LOOP, FOR и IS двойников нет.)

Функции-двойники применяются в тех случаях, когда заранее известно, что выполняемое действие не должно отменяться при неуспехе, или когда безразлично, будет отменяться действие или нет, и потому ради экономии памяти лучше использовать, скажем, функцию PSET, а не функцию SET.

Рассмотрим типичный пример использования функции PSET:

```
[PROG (X (Y ())) [ALT () [RETURN .Y]]
      [PSET X [Ф8]]
      [PSET Y (!.Y .X)]
      [FAIL]]
```

При вычислении этого выражения создается список Y всех решений задачи о восьми ферзях. Если бы не было функции PSET, сделать это не удалось бы. В самом деле, если вместо оператора [PSET Y (!.Y .X)] записать оператор [SET Y (!.Y .X)], то при каждом неуспехе уничтожалась бы только что сделанная запись

очередного решения X в список Y, поэтому у переменной Y всегда сохранялось бы начальное значение ( ). Применение же функции PSET «спасает» список Y от таких ненужных восстановлений. Что же касается переменной X, то совершенно безразлично, будет или нет восстанавливаться при неуспехе ее прежнее значение, поэтому, чтобы не запоминать лишний обратный оператор, мы и написали оператор [PSET X [Ф8]] вместо также допустимого оператора [SET X [Ф8]].

Следует отметить, что функция PSET (и ей подобные функции) не запоминает обратный оператор только для себя и не оказывает никакого влияния на другие присваивания. Поэтому значением выражения

```
[PROG ((X A)) [ALT ( ) [RETURN .X]]
      [SET X B]
      [PSET X C]
      [FAIL]]
```

является атом A, но не атом C, как может показаться. Действительно, при вычислении [SET X B] переменной X присваивается значение B и одновременно запоминается обратный оператор [SET X A]. Затем значение переменной X меняется с B на C, и при этом обратный оператор не запоминается. Но это не означает, что переменная X сохранит значение C. Указанный обратный оператор остался, он-то при неуспехе и восстановит у X значение A. Таким образом, если мы хотим, чтобы некоторая переменная не изменяла при неуспехе своего значения, то следует все присваивания ей производить только функцией PSET, а не вперемежку с функцией SET.

Рассмотрим еще один пример:

```
[DEFINE INFINITE (LAMBDA ( )
   [PROG ((N 0))
      A [ALT [PADD1 N] [GO A]]])]
```

Эта функция подобна развилке с бесконечным числом альтернатив, каждая из которых — очередное натуральное число. Действительно, при возврате к этой функции по неуспеху F-точка функции ALT уничтожается, но затем вновь вызывается функция ALT, т. е. тут же ставится новая F-точка. Получается как бы неуничтожаемая F-точка. Кроме того, функция PADD1 не запоминает обратный оператор, поэтому присваиваемое ею значение переменной N сохраняется и после возврата по неуспеху. Тем самым каждый раз значение переменной N увеличивается на 1, что и обеспечивает выдачу всех натуральных чисел.

3.6. Уничтожение развилок и обратных операторов

Теперь мы рассмотрим встроенные функции языка, с помощью которых можно уничтожать ранее запомненные обратные операторы и F-точки.

Если заранее известно, что после вычисления какого-то выражения должны быть уничтожены F-точки и/или обратные операторы, которые появились во время вычисления этого выражения (для краткости будем говорить: «внутри этого выражения»), то применяется одна из функций PERM, STRG или TEMP.

Функция PERM: [PERM *e*], FSUBR.

Эта функция вычисляет свой аргумент *e*. Если его вычисление неуспешно, то неуспешным является и вычисление функции. Иначе функция успешно заканчивает свою работу со значением *E*, но перед выходом она уничтожает все F-точки и обратные операторы внутри *e*.

Например, в результате вычисления выражения

```
[PROG ((X A)) [ALT ( ) [RETURN .X]]
      [PERM [ALT [SET X B] [PRINT C]]]
      [FAIL]]
```

будет получено значение B, причем печататься ничего не будет. Действительно, функция PERM уничтожает как развилку функции ALT, так и обратный оператор функции SET, поэтому неуспех, выработанный функцией FAIL, не восстанавливает прежнее значение A переменной X и будет «пойман» только F-точкой первой функции ALT.

Функция PERM применяется тогда, когда вычисление некоторого выражения не должно возобновляться при неуспехе и когда все, что сделано при вычислении этого выражения, не должно быть отменено при неуспехе. Функция используется и в тех случаях, когда надо спасти память, которую транслятор языка тратит на запоминание обратных операторов и F-точек. Например, если в задаче о восьми ферзях нас интересует только первое решение, то следует вычислять выражение [PERM [Ф8]], чтобы после успешного вычисления функции Ф8 внутри нее не оставалось ни F-точек, ни обратных операторов, которые уже не нужны.

Отметим, что вычисление выражения [PSET X *e*] во многом похоже на вычисление выражения [PERM [SET X *e*]], но если в первом случае обратный оператор просто не запоминается, то во втором случае функция SET запоминает его, а затем функция PERM уничтожает его. Однако есть и более важное различие между этими выражениями: функция PSET не уничтожает F-точ-

ки и обратные операторы внутри своего аргумента *e*, в то время как функция PERM заодно с обратным оператором функции SET уничтожает и все F-точки и обратные операторы внутри *e*. Поэтому вычисление выражения

```
[DO [PSET X [ALT A [EXIT .X DO]]] [FAIL]]
```

успешно и выдает значение A, в то время как вычисление выражения

```
[DO [PERM [SET X [ALT A [EXIT .X DO]]]]
    [FAIL]]
```

неуспешно.

Может случиться так, что при вычислении аргумента функции PERM произошел «насильственный» выход из нее (с помощью функции GO, RETURN или EXIT). Тогда функция PERM не выполняет свои «выходные» действия. Например, после вычисления

```
[PERM [DO [SET X [AMONG .L]]
          [EXIT ( ) PERM]]]
```

сохраняются как F-точка функции AMONG, так и обратный оператор функции SET. Данное замечание справедливо и для функций STRG и TEMP.

Функция STRG: [STRG *e*], FSUBR.

Эта функция действует аналогично функции PERM, но уничтожает только F-точки внутри своего аргумента *e*, сохраняя обратные операторы.

Например, в результате вычисления выражения

```
[PROG ((X A)) [ALT ( ) [RETURN .X]]
    [STGR [ALT [PRINT [SET X B]]
        [PRINT C]]]
    [FAIL]]
```

будет получено значение A и будет напечатано только B.

Функция STRG применяется в тех случаях, когда неуспех не должен возобновлять вычисление выражения *e*, но должен уничтожить все побочные эффекты первого вычисления этого выражения.

Функция TEMP: [TEMP *e*], FSUBR.

Эта функция вычисляет свой аргумент *e*. Если это вычисление неуспешно, то неуспешно и вычисление функции. Иначе функция успешно заканчивает свою работу со значением *E*, а перед выходом выполняет следующие действия: уничтожает все F-точки

внутри e и выполняет все обратные операторы, которые были запомнены при вычислении e. Таким образом, функция TEMP сама уничтожает побочные эффекты, имевшие место при вычислении ее аргумента.

Например, в результате вычисления выражения

```
[PROG ((X A))
    [TEMP [ALT [PRINT [SET X B]]
        [PRINT C]]]
    .X]
```

будет получено значение A и будет напечатано только B.

Функция TEMP применяется в тех случаях, когда надо проверить, может ли быть вычисление некоторого выражения успешным или нет, и когда надо при любом исходе сразу же уничтожить все следы этой проверки. Например, рассмотренная в § 3.3 функция PATH при поиске пути в графе изменяет значения переменных из предикатов и операторов графа, и если она находит путь, то значения этих переменных оказываются отличными от начальных значений. Когда мы хотим определить, существует ли путь в графе, но не хотим, чтобы при этом менялись значения переменных, следует вычислить выражение [TEMP [PATH A C]]. Здесь функция PATH все же изменит значения переменных, но затем функция TEMP восстановит их начальные значения.

Следующие блочные функции представляют собой композиции трех предыдущих функций с функцией PROG:

$[\text{PPROG } (v_1\ v_2 \ldots v_m)\ e_1\ e_2 \ldots e_k]$, FSUBR, $m \geqslant 0,\ k \geqslant 1$
$[\text{SPROG } (v_1\ v_2 \ldots v_m)\ e_1\ e_2 \ldots e_k]$, FSUBR, $m \geqslant 0,\ k \geqslant 1$
$[\text{TPROG } (v_1\ v_2 \ldots v_m)\ e_1\ e_2 \ldots e_k]$, FSUBR, $m \geqslant 0,\ k \geqslant 1$

Любая из этих функций вычисляется так же, как и функция PROG, но перед выходом из нее (по RETURN или после того, как выполнен последний оператор e_k) выполняются те же «выходные» действия, что и в функциях PERM, STRG или TEMP соответственно. Например, при выходе из функции SPROG уничтожаются все F-точки, поставленные при вычислении ее тела, но сохраняются обратные операторы. Отметим, однако, что при выходе из этих функций, осуществляемом с помощью функции GO или EXIT, «выходные» действия не выполняются.

Все предыдущие функции этого параграфа можно использовать только тогда, когда известно заранее, что F-точки и/или обратные операторы должны быть уничтожены. Но бывает и так, что мы не знаем об этом заранее и узнаем об этом позже, когда вычисления уже начаты. В таких ситуациях применяются встроенные функции PERMEX, STRGEX, TEMPEX и FAILEX, которые подобно функции EXIT осуществляют выход из некоторой

объемлющей процедуры, уничтожая при этом F-точки и/или обратные операторы внутри данной процедуры.

Функция PERMEX: [PERMEX *e fn n*?], SUBR.

Эта функция действует аналогично функции EXIT (см. § 1.16 и § 2.6), т. е. осуществляет выход со значением E из $(N+1)$-й объемлющей процедуры с именем *FN*, но перед выходом она уничтожает все F-точки и обратные операторы, появившиеся с начала вычисления данной процедуры (для встроенных процедур класса SUBR и определяемых процедур — с того момента, как началось вычисление их аргументов).

Функция STRGEX: [STRGEX *e fn n*?], SUBR.

Данная функция действует аналогично функции PERMEX, но уничтожает только F-точки.

Функция TEMPEX: [TEMPEX *e fn n*?], SUBR.

Эта функция действует так же, как функция STRGEX, но дополнительно еще выполняет все обратные операторы, запомненные с начала вычисления процедуры, из которой сейчас осуществляется выход.

Функция FAILEX: [FAILEX *e fn n*?], SUBR.

И эта функция осуществляет выход из $(N+1)$-й и объемлющей процедуры с именем *FN*, но выход по неуспеху: функция уничтожает все F-точки, поставленные внутри данной процедуры, и вырабатывает неуспех, связывая с ним сообщение E. Таким образом, по этому неуспеху происходит возврат к развилке, которая была определена до начала выполнения данной процедуры *FN*.

Отметим, что при выходе из функций PERM, STRG, TEMP, PPROG, SPROG и TPROG, осуществляемом функцией PERMEX, STRGEX, TEMPEX или FAILEX, выполняются «выходные» действия последних функций.

Приведем пример использования рассмотренных функций:

```
[PROG (X)
    [ALT ( ) [RETURN T]]
    [SET X [Ф8]]
    [COND
        ([EQ [5 .X] 2] [FAILEX (5 2) PROG])
        ([EQ [— 2 .X] 6] [PERMEX (7 6) PROG])
        (T [FAIL])] ])]
```

Если в задаче о восьми ферзях существует решение, в котором один из ферзей расположен на 5-й горизонтали и 2-й вертикали, тогда данное выражение вырабатывает неуспех с сообщением

(5 2). Если же есть решение, где один из ферзей находится на 7-й горизонтали и 6-й вертикали, тогда все F-точки и обратные операторы внутри этого выражения уничтожаются, а само выражение успешно и имеет значение (7 6). В остальных случаях значением выражения является атом T.

3.7. Именованные развилки

Развилки, которые мы до сих пор рассматривали, не идентифицируются. Если возвраты по неуспеху происходят только к последним развилкам, то различать развилки и не нужно. Но если требуется осуществить возврат к какой-то предыдущей развилке, то возникает проблема: каким образом указать эту развилку? В некоторых случаях здесь помогает рассмотренная в предыдущем параграфе функция FAILEX. С ее помощью можно осуществить возврат к развилке, что была определена до входа в какую-то объемлющую процедуру. Однако другие развилки, например те, что остались в процедурах, уже закончивших свою работу, оказываются недоступными.

В связи с этим в плэнере разрешено давать развилкам имена, чтобы затем можно было ссылаться на эти развилки и осуществлять возврат к ним по неуспеху.

Функция FP: [FP *name* e_1 e_2 ... e_k], FSUBR, $k \geqslant 1$.

Эта функция действует точно так же, как функция ALT, т. е. определяет развилку, i-й альтернативой которой является вычисление формы e_i, но в дополнение к этому функция FP дает имя своей развилке, которое задается аргументом *name*. Данный аргумент должен быть атомом или .-переменной, значением которой является атом; этот атом и есть имя развилки.

Осуществить возврат по неуспеху к развилке, имеющей имя, можно с помощью функции FAIL. В § 3.2 были рассмотрены обращения к этой функции с одним аргументом или без аргументов. Допускается также обращение с двумя аргументами:

[FAIL *e a*], SUBR.

Значением первого аргумента, как и раньше, может быть любое выражение E, а вот значением второго аргумента должен быть какой-либо атом A. Функция FAIL уничтожает все F-точки, поставленные после того, как была определена развилка с именем A, и вырабатывает обычный неуспех, связывая с ним сообщение E. Тем самым этот неуспех вернет программу к развилке с именем A.

Если развилок с именем A несколько, то возврат происходит к последней из них. Если же таких развилок вообще нет, то осуществляется выход по неуспеху на верхний уровень программы —

вычисление текущего выражения программы оказывается неуспешным.

Рассмотрим следующий пример. Предположим, что некоторое дерево задается своим корнем S и функцией ДОЧВЕРШ, которая по названию любой вершины дерева выдает список названий всех ее дочерних вершин. Требуется определить, есть ли в этом дереве хотя бы одна вершина на уровне, глубина которого больше заданного числа N. (Считаем, что корень дерева находится на нулевом уровне, его дочерние вершины — на первом уровне и т. д.)

Решение этой задачи может быть описано в виде следующей функции:

```
[DEFINE УРОВ (LAMBDA (S N) [PROG ((K 0))
    [FP 1 ( ) [RETURN [NEQ [MESS] AMONG]]]
L [SET S [AMONG [ДОЧВЕРШ .S]]]
    [ADD1 K]
    [COND ([LE .K .N] [GO L])]
    [FAIL T 1]])]
```

Данная функция ведет поиск в глубину: выбирает какую-то ветвь дерева и исследует ее до конца или до уровня N + 1. Число K указывает уровень рассматриваемой в данный момент вершины S. Если K не больше N, а S — концевая вершина, то функция ДОЧВЕРШ в качестве своего значения выдает пустой список, поэтому функция AMONG вырабатывает неуспех (с сообщением AMONG), по которому происходит возврат к предыдущей вершине дерева, и далее рассматривается другая ветвь от этой предыдущей вершины. Если глубина дерева не превосходит N, то будут просмотрены все вершины дерева и в конце концов программа вернется к функции FP по неуспеху, с которым связано сообщение AMONG. В данном случае значение функции УРОВ равно ( ). Но если при просмотре дерева была найдена вершина на более глубоком, чем N, уровне, тогда поиск сразу же прекращается и вырабатывается неуспех с сообщением T, который, минуя все развилки, соответствующие точкам ветвления дерева, распространится до функции FP. В данном случае значением функции УРОВ является атом T.

3.8. Функции IF и FIND

В плэнере проверку каких-либо условий можно оформить не только в виде функций-предикатов, принимающих значения «истина» и «ложь», но и в виде функций, вычисление которых успешно, если проверяемое условие выполнено, и неуспешно в противном случае. Таковы, например, рассмотренные выше функции SUM, Ф8 и PATH. Если для работы с предикатами использу-

ется функция COND, то для работы с функциями второго типа в языке используется встроенная функция IF, обращение к которой можно рассматривать как условное выражение, реагирующее на успех/неуспех. Как и у функции COND, аргументы функции IF называются клаузами, а их первые элементы — условиями. Обращение к этой функции имеет следующий вид:

$$\left[\text{IF } \left(e_1\ e_{11} \cdots e_{1m_1}\right) \ldots \left(e_k\ e_{k1} \cdots e_{km_k}\right)\right], \quad \text{FSUBR},$$
$$k \geqslant 1, \quad m_i \geqslant 0.$$

Функция IF прежде всего ставит F-точку, а затем по очереди вычисляет условия e_i. Если вычисление очередного условия неуспешно, то выработанный при этом неуспех будет «пойман» данной F-точкой, после чего функция переходит к вычислению условия из следующей клаузы. Если окажется, что все условия неуспешны, функция уничтожит свою F-точку и успешно закончит работу со значением ( ).

Если же найдено условие e_i, вычисление которого успешно, тогда функция IF уничтожает все F-точки (но не обратные операторы) внутри этого условия и свою F-точку и затем последовательно вычисляет оставшиеся выражения этой i-й клаузы. В случае их успешного вычисления успешно и вычисление всего условного выражения, причем его значением объявляется значение последнего выражения из i-й клаузы. В противном случае вычисление функции IF неуспешно, поскольку ее F-точки уже нет и, следовательно, выработанный при вычислении выражений клаузы неуспех вернет программу к развилке, определенной до начала вычисления функции IF. (Подчеркнем, что если найдено успешное условие, то оставшиеся клаузы никогда не рассматриваются.) Примеры:

```
[IF ([FAIL] A) ([FAIL] B)]  →  ( )
[IF ([FAIL] A) (( ) B)]  →  B
[IF (A)]  →  A
[IF (T [FAIL]) (T B)]  — неуспех
[IF ([ALT T [FAIL 1]] [FAIL 2])]  — неуспех с сообщением 2
```

Еще один пример. При вычислении выражения

```
[IF ([SUM (3 8) 205] [GO A])]
```

будет осуществлен переход по метке A, если из чисел 3 и 8 можно составить сумму 205. В противном случае функция IF вырабатывает значение ( ).

Следующая функция FIND используется в тех случаях, когда требуется найти определенное число вариантов успешного вычис-

ления какого-то выражения или некоторой последовательности выражений. Например, может потребоваться найти три таких решения задачи о восьми ферзях, в которых один из ферзей расположен на 2-й горизонтали и 6-й вертикали. Это, конечно, можно сделать и с помощью ранее рассмотренных функций, но применение функции FIND оказывается более удобным.

Функция FIND относится к классу FSUBR. Обращение к ней выглядит так:

[FIND *mode* (v_1 v_2 ... v_m) p e_1 e_2 ... e_k],

где $m \geqslant 0$, $k \geqslant 1$. FIND — блочная функция: список (v_1 v_2 ... v_m) является описанием ее локальных переменных, а последовательность e_1 e_2 ... e_k образует ее тело. Аргументы *mode* и p должны быть простыми формами. Вычисляется функция следующим образом.

Сначала вычисляется аргумент *mode*. Его значением должен быть список вида (q *min max*), где q, *min*, *max* — неотрицательные целые числа или атом ALL. Если число q равно нулю, тогда функция FIND сразу завершает свою работу со значением ( ). Иначе вводятся локальные переменные v_i (точно так же, как и в функции PROG) и еще две вспомогательные переменные n и *col* с начальными значениями 0 и ( ) соответственно. После этого вычисляется тело блока. Если это вычисление успешно дошло до конца тела (успешно вычислился последний оператор e_k), то n увеличивается на 1, вычисляется аргумент p, и его значение заносится в начало списка *col*. Затем функция FIND вырабатывает неуспех, возобновляя тем самым вычисление своего тела; ищется новый вариант его успешного вычисления. Если такой вариант нашелся (вычисление снова дошло до конца тела), тогда n вновь увеличивается на 1, заново вычисляется агрумент p, и его новое значение добавляется к списку *col*. После этого опять вырабатывается неуспех и т. д.

Функция FIND пытается найти q вариантов успешного вычисления своего тела. Если стольких вариантов нет, количество найденных вариантов (n) должно удовлетворять условию $min \leqslant n \leqslant max$. В любом из этих случаев функция успешно заканчивает свою работу, объявляя своим значением список *col*, в котором собраны значения аргумента p, вычисленные при каждом варианте. Но если q вариантов не найдено или если n меньше *min* или больше *max*, то вычисление функции неуспешно. С выработанным при этом неуспехом связывается как сообщение последнее значение переменной *col*.

После того как функция FIND вычислила аргумент *mode* (считаем, что $q \neq 0$), ее действие полностью эквивалентно вычислению следующего блока:

```
[SPROG (v1 v2 ... vm (n 0) (col ()))
    [ALT ()
        [COND ([GE .n min] [RETURN .col])
              (T [FAIL .col])]]
    e1 e2 ... ek
    [PADD1 n]
    [PSET col (p !.col)]
    [COND ([EQ .n q] [RETURN .col])
        ([GT .n max] [FAILEX .col SPROG])
        (T [FAIL])]]
```

Отметим, что независимо от того, успешно или нет закончилось вычисление функции FIND, все F-точки, появившиеся после вычисления аргумента *mode*, уничтожаются; F-точки же внутри *mode* сохраняются.

Атом ALL для *q*, *min* и *max* играет роль бесконечности. Например, значение (ALL 3 7) аргумента *mode* означает, что надо найти все (сколько есть) варианты успешного вычисления тела функции FIND, и таких вариантов должно быть не меньше трех и не больше семи, иначе вычисление функции будет неуспешным. Значение же (5 2 ALL) означает, что должно быть найдено пять вариантов; если есть еще варианты, то они уже не рассматриваются, а если пяти вариантов нет, тогда требуется, чтобы существовало хотя бы два варианта.

Для наиболее часто встречающихся значений аргумента *mode* — для значений (*q q q*) и (ALL 0 ALL) — допускается сокращенная форма: *q* и ALL соответственно. Эти значения интерпретируются так: «найти ровно *q* вариантов» (не больше и не меньше) и «найти все варианты» (сколько есть). Например, значением выражения

```
[FIND ALL (X) :X
    [SET X [AMONG (1 —2 3 4 —5)]]
    [COND ([LT .X 0] [FAIL])]]
```

является список (4 3 1). Здесь, когда число X отрицательно, вычисление не доходит до конца тела функции FIND, поэтому такие числа не запоминаются. Неотрицательные же числа учитываются все, как того и требует аргумент ALL.

Другой пример:

```
[FIND 3 (X) .X [UNFALSE [EQ [2 [SET X [Ф8]]] 6]]]  →
    ((2 6 8 3 1 4 7 5) (2 6 1 7 4 8 3 5)
    (1 6 8 3 7 4 2 5))
```

Здесь функция FIND находит три решения задачи о восьми ферзях из числа тех решений, в которых один из ферзей расположен на 2-й горизонтали и 6-й вертикали.

3.9. Некоторые уточнения

В связи с использованием в языке режима возвратов требуют уточнения некоторые аспекты вычисления плэнерских программ и сопоставления образцов.

Программа на плэнере, как известно, представляет собой последовательность выражений, которые вычисляются друг за другом. Вычисление любого из них может быть успешным или неуспешным, причем в первом случае внутри выражения могут остаться развилки и обратные операторы. Сохраняются ли эти развилки и обратные операторы? Осуществляется ли возврат по неуспеху к предыдущему выражению программы, если вычисление очередного выражения оказалось неуспешным?

Ответы на оба вопроса отрицательные. Дело в том, что выражения верхнего уровня программы вычисляются, с точки зрения режима возвратов, независимо друг от друга, и вернуться к ранее вычисленному выражению нельзя. А раз так, то нет смысла сохранять развилки и обратные операторы внутри выражения, вычисление которого закончено. Таким образом, если некоторое выражение программы успешно, то по окончании его вычисления все оставшиеся внутри него развилки и обратные операторы уничтожаются, а если выражение неуспешно, то распространение появившегося при этом неуспеха блокируется, после чего вместо результата вычисления печатаются слово =НЕУСПЕХ= и сообщение, связанное с этим неуспехом. В любом случае далее вычисляется следующее выражение программы.

Например, вычисление программы

```
[CSET PI 3.14159]
[AMONG (A B C)]
[FAIL 7]
```

происходит таким образом. Вычисляется первое выражение: вводится константа PI и ей присваивается значение 3.14159; это число печатается как результат вычисления данного выражения; обратный оператор, который запоминался при этом, уничтожается. Затем вычисляется второе выражение, и его значение — атом A — выводится на печать; F-точка функции AMONG уничтожается. При вычислении третьего выражения программы вырабатывается неуспех, с которым связывается сообщение 7; этот неуспех блокируется, т. е. он не вызывает никакого возврата программы назад, и печатаются слово =НЕУСПЕХ= и атом 7.

Теперь рассмотрим, как связаны между собой режим возвратов и механизм сопоставлений.

Прежде всего отметим, что, несмотря на внешнее сходство, выбор подходящих сегментов для сегментных образцов и неудачи

сопоставлений не имеют никакого отношения к развилкам и неуспехам режима возвратов. Сопоставление образцов и режим возвратов лежат как бы в разных плоскостях. Неудачное сопоставление какого-либо элемента образца-списка с фрагментом анализируемого выражения может вызвать возврат только к ранее рассмотренным сегментным элементам образца, но никак не возврат к развилкам. При неуспехах же происходит возврат к развилкам, но не к сегментным элементам образцов.

Отметим также, что если во время сопоставления вычисляется какая-либо форма (образец, являющийся обращением к функции, или аргумент сопоставителя), то по окончании ее вычисления все F-точки внутри нее обязательно уничтожаются, т. е. эта форма как бы окружается функцией STRG. Например, при сопоставлении

```
[IS [AMONG (A B C)] .X]
```

функция AMONG выдает значение A, после чего ее F-точка уничтожается. Поскольку F-точки могут ставить лишь функции, но и их F-точки, как мы видим, уничтожаются, то появление при сопоставлении любого неуспеха ведет к прекращению сопоставления и возврату программы к развилке, которая была определена до начала сопоставления. В этом случае сопоставление не является ни неудачным, ни удачным, его исход неопределен. Например, выражение

```
[IS ([AMONG (A B)] [FAIL]) (A C)]
```

неуспешно, а потому и не имеет значения. По той же причине внутри функции IS никогда не остается F-точек, поэтому никакой неуспех не может заставить ее возобновить свою работу. В связи с этим вычисление выражения

```
[COND ([IS [AMONG (A B)] A] [FAIL]) (T T)]
```

неуспешно.

И, наконец, отметим, что любое изменение значений переменных образца сопровождается запоминанием обратных операторов. Обратные операторы могут появляться и при вычислении функций, обращение к которым имеется в образце. Именно эти обратные операторы и уничтожают побочные эффекты сопоставления как в случае его неудачного исхода, так и при распространении неуспеха.

Например, при начальных значениях A и 6 у переменных X и Y сопоставление

```
[IS (!*X [ADD1 Y] < >) (4 7 2 9)]
```

осуществляется следующим образом. Вначале образцу !*X ставится в соответствие пустой сегмент, при этом переменная X

получает значение ( )и попутно запоминается обратный оператор [SET X A]. Далее вычисляется функция ADD1: значение переменной Y увеличивается на 1 и запоминается обратный оператор [SET Y 6], после чего значение функции — число 7 — сравнивается с первым элементом анализируемого списка. Они не равны, поэтому происходит возврат к образцу !*X и изменяется сопоставляемый с ним сегмент. При этом возврате выполняются все обратные операторы, появившиеся после того, как для данного сегментного образца был выбран пустой сегмент. Следовательно, у переменной Y будет восстановлено значение 6, а у переменной X — значение A. Теперь образцу !*X ставится в соответствие сегмент из одного элемента 4; переменная X получает значение (4), и вновь запоминается обратный оператор [SET X A]. Далее вновь вычисляется функция ADD1: переменной Y присваивается значение 7 и запоминается обратный оператор [SET Y 6]. На этот раз значение функции равно очередному элементу (числу 7) анализируемого списка, поэтому сопоставление списков удачно завершается. Таким образом, функция IS заканчивает свою работу со значением T; появившиеся при ее вычислении обратные операторы [SET X A] и [SET Y 6] сохраняются. Если затем в программе будет выработан неуспех, то при его распространении эти обратные операторы восстановят начальные значения переменных X и Y, т. е. значения A и 6.

Итак, побочные эффекты сопоставления уничтожаются только благодаря выполнению обратных операторов, появившихся во время сопоставления. Поэтому если обратный оператор не запоминался, то не будет и отмены соответствующего побочного эффекта. Например, сопоставление

```
[IS [ET *X [BE [PSET Y .X]] [NUM]] A]
```

неудачно, однако значение A, полученное переменной Y в процессе сопоставления, сохранится.

ГЛАВА 4

БАЗА ДАННЫХ

4.1. Основные понятия

Многие системы искусственного интеллекта, получив на входе описание некоторой задачи, должны найти ее решение. Например, системе может быть предложен набор аксиом и правил вывода, а она должна найти доказательство некоторой теоремы. Или, скажем, системе может быть предложено сформировать план действий робота для достижения поставленной цели в заданной обстановке. В связи с этим в языке программирования, предназначенном для реализации подобных систем, должны быть предусмотрены удобные средства для описания всех компонентов задачи (исходной ситуации, допустимых правил решения, цели) и для реализации процедур поиска решения задач. Язык плэнер учитывает эти требования и предоставляет своим пользователям соответствующие средства. К ним, в частности, относится *база данных*, предназначенная для описания исходной ситуации и тех ситуаций, которые возникают при поиске решения рассматриваемой задачи.

Базой данных в плэнере называется набор *утверждений*. Утверждением может быть любой список в круглых скобках; список называется утверждением, если он включен в базу данных. С каждым утверждением может быть связан список свойств.

Между утверждениями базы данных не устанавливается никаких отношений, в частности не определено понятие места (адреса, индекса) утверждения в базе данных. Доступ к утверждениям осуществляется только по ассоциативному принципу — по описанию признаков искомых утверждений.

В каждой плэнерской программе используется своя база данных, которая в начале выполнения программы пуста (поэтому заполнять ее обязана сама программа), а по окончании выполнения программы уничтожается*).

*) Таким образом, плэнерскую базу данных не следует рассматривать как совокупность данных, хранимых во внешней памяти ЭВМ и доступных из разных программ, что обычно подразумевается в программировании под термином «база данных».

Для работы с базой данных используются следующие операции: запись новых утверждений в базу данных, вычеркивание утверждений из базы данных, поиск утверждений по образцу.

Выполнение операций записи и вычеркивания, изменяющих содержимое базы данных, может сопровождаться вызовом особых процедур — так называемых записывающих и вычеркивающих теорем, назначение которых — контролировать изменения базы данных. Эти процедуры имеют образцы, и вызываются те из них, образцы которых соответствуют записываемому или вычеркиваемому утверждению. Подробности вызова теорем будут рассмотрены в следующей главе.

Операция поиска по образцу используется для анализа содержимого базы данных: задается некоторый образец, и находится то утверждение, которое ему соответствует. При сопоставлении образца с найденным утверждением переменные образца получают соответствующие значения, что дает возможность определить, какое именно утверждение было найдено.

Все эти операции выполняются в режиме возвратов. Если некоторое утверждение было записано в базу данных, но затем в программе возник неуспех, то это утверждение будет удалено из базы данных. Аналогично ранее вычеркнутое утверждение будет при неуспехе восстановлено в базе данных. Поиск же по образцу — это развилка в программе, так как образцу может соответствовать несколько утверждений базы данных. При поиске сначала находится только одно из них, но если в дальнейшем в программе возникнет неуспех, то поиск возобновится и будет найдено следующее подходящее утверждение.

Каждая из этих основных операций имеет несколько модификаций. Например, возможны такие изменения базы данных, которые не отменяются при неуспехе, или возможен такой поиск по образцу, который не возобновляется при неуспехе.

В планерских программах базу данных можно использовать просто как хранилище, в которое можно записывать какую-то информацию, удерживать ее там сколь угодно долго, определять, что там сейчас находится, и т. д. Однако база данных введена в язык с иной целью. Как уже сказано, она предназначена для описания той обстановки, в которой решается задача, предложенная пользователем. Например, программа, планирующая действия робота, должна иметь описание (модель) той среды, в которой действует робот, только тогда она может что-то планировать. Вот такая модель и представляется в виде базы данных.

Как правило, описание любой ситуации можно представить в виде совокупности простых фактов. Например, ситуацию, изображенную на рис. 5, можно описать с помощью следующих фактов: R1 — это комната, R2 — это комната, D — это дверь, D сое-

диняет R1 и R2, робот находится в R1, A — это ящик, A находится в R2. Если удалось так представить ситуацию, то каждый факт описывается в виде одного утверждения. База же данных, как совокупность этих утверждений, будет описывать ситуацию в целом.

Каким образом можно описывать факты в виде утверждений? Никаких ограничений на этот счет в языке нет; единственное требование: утверждения должны быть L-списками. Для языка утверждение — это просто список. Формат же записи фактов, их интерпретацию определяет сам пользователь. Например,

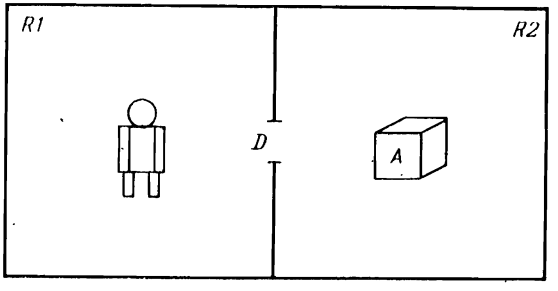


Рис. 5.

факт «B является пирамидой красного цвета» можно описать в виде утверждения (PYRAMID B IS RED) или в виде утверждения (COLOUR OF (PYRAMID B) IS RED), а можно описать и в виде двух утверждений (PYRAMID B) и (COLOUR B RED).

Из всех этих форм записи фактов обычно используется предикатная форма:

$(P\ a_1\ a_2\ \ldots\ a_k)$

где P — имя отношения, описываемого фактом, а a_i — названия объектов, связанных этим отношением. В такой форме одноместные отношения (свойства) описываются как (ROOM R1) или (INTEGER 5), двухместные — как (AT ROBOT R1) или (EQ X 5), трехместные — как (CONNECTS D R1 R2) или (BETWEEN A B C) и т. д. Такого формата мы и будем придерживаться в дальнейшем. Например, используя его, ситуацию, изображенную на рис. 5, можно описать в виде следующего набора утверждений:

(ROOM R1), (ROOM R2), (DOOR D), (BOX A),
(CONNECTS D R1 R2), (AT ROBOT R1), (AT A R2)

Как уже было сказано, с утверждениями базы данных могут быть связаны списки свойств. В этих списках обычно указывается информация о самих утверждениях или о тех фактах, что ими

описываются. Например, можно ввести некоторое свойство и увеличивать его (числовое) значение на единицу каждый раз, когда программа использует утверждение; в конце выполнения программы по этому свойству можно будет узнать, какие утверждения использовались наиболее часто. Или, например, с утверждением (BOX A), означающим, что A является ящиком, можно связать список свойств, указав в нем дополнительную информацию о предмете A: что он, скажем, красного цвета, имеет форму куба и т. п.

Итак, база данных обычно используется как модель того мира, в котором программа должна решать задачи. При такой интерпретации базы данных операции над ней можно трактовать следующим образом. Изменениями базы данных имитируются изменения в этом мире. Например, переход робота из комнаты R1 в комнату R2 имитируется вычеркиванием утверждения (AT ROBOT R1) и записью утверждения (AT ROBOT R2). Поиск же по образцу можно понимать как проверку истинности каких-либо высказываний или как выявление объектов, находящихся в определенных отношениях. Например, поиск по образцу (AT ROBOT R1) — это проверка, действительно ли робот находится в комнате R1, а поиск по образцу (AT *X R2) — это нахождение объекта, расположенного в комнате R2: при сопоставлении данного образца с утверждением (AT A R2) переменная X получает значение A, что и является ответом.

В следующих параграфах описаны встроенные функции языка, предназначенные для работы с базой данных.

4.2. Запись утверждений

Основной функцией, используемой для записи новых утверждений в базу данных, является

Функция ASSERT: [ASSERT *asrt with*? *rec*? *else*?], FSUBR.

Значением аргумента *asrt* должен быть L-список, который как утверждение будет записан в базу данных. Аргумент *with* обязан иметь вид

(WITH ind_1 v_1 ind_2 v_2 ... ind_k v_k), $k \geqslant 1$,

где ind_i и v_i — простые формы; значениями ind_i должны быть идентификаторы IND_i, а v_i могут иметь произвольные значения V_i. Вычисление этого аргумента дает список свойств (IND_i — названия свойств, V_i — их значения), который будет связан с записываемым утверждением. Если при обращении к функции ASSERT аргумент *with* не задан, то утверждение не будет иметь

списка свойств, или, другими словами, с ним будет связан пустой список свойств.

Аргумент *rec* называется рекомендацией и указывает, какие записывающие теоремы должны быть вызваны. Использование рекомендаций будет объяснено в главе 5.

Аргумент *else* должен иметь вид (ELSE *e*?), где *e* — простая форма. Он определяет действия функции ASSERT в том случае, когда ей не удается записать утверждение в базу данных. Отсутствие аргумента *else* эквивалентно заданию аргумента (ELSE ()), а аргумент (ELSE) рассматривается как сокращение от (ELSE [FAIL]).

Функция ASSERT действует следующим образом. Сначала она вычисляет аргумент *asrt* и, если он есть, аргумент *with*, а затем просматривает базу данных, чтобы определить, не записано ли там такое же утверждение. Если записано (при любом списке свойств), тогда функция ничего не добавляет в базу данных и вычисляет лишь форму *e* из аргумента *else*. Но если утверждения в базе данных еще не было, то оно записывается туда и с ним связывается список свойств, полученный из аргумента *with*. Одновременно запоминается обратный оператор, назначение которого — отменить при неуспехе эту запись в базу данных. При отсутствии рекомендации *rec* функция ASSERT на этом и заканчивает свою работу; ее значение — записанное утверждение. Если же аргумент *rec* задан, то осуществляется вызов записывающих теорем: вычисляется выражение [DRAW *a rec*], где *a* — только что записанное утверждение (см. § 5.3).

Рассмотрим несколько примеров. Предположим, что переменная X имеет значение B. Тогда в результате вычисления

```
[ASSERT (SUBSET A .X)]
```

в базу данных будет записано утверждение (SUBSET A B), с которым связывается пустой список свойств. Значение функции — список (SUBSET A B). Однако, если такое утверждение уже имелось в базе данных, то повторно оно не записывается; в данном случае функция ASSERT просто вырабатывает значение ().

Пусть переменная AS имеет значение (BOX A), а переменная Y — значение RED. Тогда при вычислении

```
[ASSERT .AS (WITH COLOUR .Y FORM CUBE)
      (ELSE)]
```

в базу данных будет записано утверждение (BOX A), с которым будет связано два свойства: одно — с названием COLOUR и значением RED, а другое — с названием FORM и значением CUBE.

Если же такое утверждение уже имелось в базе данных, то запись не производится, а вырабатывается неуспех (согласно аргументу (ELSE)).

Действие функции ASSERT отменяется при неуспехе. Если это нежелательно, то следует использовать «двойник» этой функции, которым является

Функция PASSERT: [PASSERT *asrt with*? *rec*? *else*?], FSUBR.

Данная функция вычисляется аналогично функции ASSERT за одним исключением: обратный оператор здесь не запоминается.

В языке есть еще две функции, которые записывают утверждения в базу данных. Они записывают сразу по несколько утверждений, что удобно делать, например, при начальном заполнении базы данных.

Функция DATA. Обращение к ней имеет следующий вид:

[DATA $asrt_1$ $asrt_2$ ... $asrt_k$], FSUBR, $k \geqslant 1$.

Действие этой функции эквивалентно вычислению выражения

[DO [PASSERT $asrt_1$] ... [PASSERT $asrt_k$]]

Таким образом, функция DATA по очереди вычисляет свои аргументы и записывает их значения (которые должны быть L-списками) в базу данных как утверждения, связывая с ними пустые списки свойств. При этом обратные операторы не запоминаются, уже записанные утверждения повторно не записываются, никакие теоремы не вызываются.

Например, базу данных, описывающую ситуацию, которая изображена на рис. 5, можно заполнить так:

```
[DATA (ROOM R1) (ROOM R2) (DOOR D) (BOX A)
    (CONNECTS D R1 R2) (AT ROBOT R1) (AT A R2)]
```

Если предположить, что в базе данных в настоящий момент записано только одно утверждение (B), тогда вычисление

```
[DATA (A) (B) [FAIL] (C)]
```

будет происходить следующим образом: (A) записывается в базу данных; (B) записано не будет, так как оно там уже есть; при вычислении третьего аргумента вырабатывается неуспех, поэтому до аргумента (C) дело не дойдет и он не будет записан. Отметим, что возникший неуспех не отменяет запись утверждения (A).

Функция DATA используется в тех случаях, когда с записываемыми утверждениями связываются пустые списки свойств. Если же утверждения должны иметь непустые списки свойств, тогда используется

Функция DB. К ней следует обращаться так:

[DB $asrt_1$ pl_1 $asrt_2$ pl_2 ... $asrt_k$ pl_k],
FSUBR, $k \geqslant 1$.

Эта функция действует аналогично функции DATA, но с записываемыми утверждениями она связывает списки свойств, которые определяют аргументы pl_i. Значением каждого аргумента pl_i должен быть список вида (IND_1 V_1 IND_2 V_2 ... IND_m V_m), $m \geqslant 0$, где IND_j — названия свойств, а V_j — их значения.

Если, например, переменная X имеет значение (COLOUR RED), то в результате вычисления

[DB (BOX A) .X (BOX B) ( ) (BOX C) (I.X SIZE 5)]

в базу данных будут записаны утверждение (BOX A), с которым связывается свойство COLOUR, имеющее значение RED, утверждение (BOX B) с пустым списком свойств и утверждение (BOX C), с которым связываются свойство COLOUR со значением RED и свойство SIZE со значением 5. Значением функции объявляется значение ее последнего аргумента.

4.3. Вычеркивание утверждений

Функция ERASE: [ERASE *asrt test*? *rec*? *else*?],
FSUBR.

Аргументы *asrt*, *rec* и *else* здесь такие же, как и у функции ASSERT (см. § 4.2). Аргумент *test* должен иметь вид

(TEST ind_1 pat_1 ind_2 pat_2 ... ind_k pat_k), $k \geqslant 1$,

где ind_i — простые формы, значениями которых обязаны быть идентификаторы IND_i, а pat_i — простые образцы. Этот аргумент используется для проверки списка свойств утверждения, которое предполагается вычеркнуть. Данная проверка осуществляется так же, как и в сопоставителе HAS (см. § 2.6): значения свойств с названиями IND_i сопоставляются с образцами pat_i, и если все эти сопоставления удачны, то считается, что список свойств удовлетворяет TEST-аргументу. При отсутствии данного аргумента список свойств не проверяется.

Функция ERASE вычисляет аргумент *asrt* и отыскивает полученное утверждение в базе данных. Если его там нет или если оно есть, но его список свойств не удовлетворяет TEST-аргументу, тогда база данных не изменяется и единственное, что делает функция,— это вычисляет форму из аргумента *else* (см. описание функции ASSERT). Если же утверждение было в базе данных

и его список свойств удовлетворяет TEST-аргументу, тогда данное утверждение вычеркивается из базы данных и попутно запоминается обратный оператор, который при неуспехе восстановит это утверждение в базе данных. При отсутствии аргумента *rec* функция ERASE на этом и заканчивает свою работу; ее значение — вычеркнутое утверждение. Если же рекомендация *rec* задана, то осуществляется вызов вычеркивающих теорем: вычисляется выражение [CHANGE *a rec*], где *a* — только что вычеркнутое утверждение (см. § 5.3).

Функция PERASE: [PERASE *asrt test*? *rec*? *else*?], FSUBR.

Это — «двойник» функции ERASE, при вычислении которого обратный оператор не запоминается.

Рассмотрим примеры. При вычислении

[ERASE (AT ROBOT R1)]

из базы данных будет вычеркнуто утверждение (AT ROBOT R1). Если же такого утверждения в базе данных нет, функция проработает как пустой оператор.

Если переменная X имеет значение (BOX C), то при вычислении

```
[ERASE .X (TEST COLOUR [NON BLUE])
    (ELSE [GO L])]
```

из базы данных удаляется утверждение (BOX C), но только при условии, что значение свойства COLOUR у этого утверждения отлично от BLUE. Если это условие не выполнено или если такого утверждения вообще нет в базе данных, то ничего не вычеркивается и осуществляется переход по метке L.

4.4. Поиск по образцу

Доступ к утверждениям базы данных осуществляет

Функция SEARCH: [SEARCH *pat test*?], FSUBR.

Образец *pat* должен быть обязательно L-списком. Первый аргумент функции SEARCH может быть также .-переменной, значением которой является образец-список, тогда этот образец и используется при поиске. Аргумент *test* — такой же, как и у функции ERASE (см. § 4.3).

Функция SEARCH прежде всего ставит F-точку, а затем осуществляет поиск утверждения, соответствующего образцу *pat*. Если такое утверждение найдено, то проверяется, удовлетворяет ли его список свойств TEST-аргументу (при отсутствии этого

аргумента список свойств не проверяется). Если не удовлетворяет, тогда функция уничтожает побочные эффекты сопоставления образца *pat* с этим утверждением и отыскивает в базе данных другое утверждение, соответствующее образцу *pat*. Когда будет найдено подходящее утверждение с подходящим списком свойств, функция успешно закончит свою работу, выдавая это утверждение в качестве своего значения. Если же в базе данных вообще нет утверждений, соответствующих образцу *pat*, или если такие утверждения есть, но их списки свойств не удовлетворяют TEST-аргументу, то функция уничтожает свою F-точку и вырабатывает неуспех с сообщением SEARCH.

После успешного завершения работы функции SEARCH в программе может возникнуть неуспех, по которому произойдет возврат к этой функции. В таком случае функция возобновляет свою работу: она просматривает оставшуюся часть базы данных и пытается найти другое подходящее утверждение. Если такое утверждение найдено, функция вновь успешно завершает свою работу. Если затем опять произойдет возврат по неуспеху, то работа функции возобновится вновь, и т. д.

Рассмотрим примеры. Предположим, что база данных была заполнена в результате вычисления

```
[DB (СТУДЕНТ ИВАНОВ)
    (ФАКУЛЬТЕТ МЕХМАТ КУРС 5)
    (СТУДЕНТ ПЕТРОВ)
    (ФАКУЛЬТЕТ ВМК КУРС 4)
    (АСПИРАНТ СЕМЕНОВ) (ФАКУЛЬТЕТ ВМК)]
```

Тогда вычисление выражения

```
[SEARCH (СТУДЕНТ *X)
    (TEST КУРС *Y ФАКУЛЬТЕТ [NON ВМК])]
```

может происходить следующим образом. Пусть функция SEARCH сначала находит утверждение (СТУДЕНТ ПЕТРОВ), тогда переменной X присваивается значение ПЕТРОВ. Далее проверяется список свойств этого утверждения: осуществляется сопоставление значения 4 свойства КУРС с образцом *Y, которое удачно, поэтому переменная Y получает значение 4, а затем значение ВМК свойства ФАКУЛЬТЕТ сопоставляется с образцом [NON ВМК], но это сопоставление неудачно. Следовательно, список свойств найденного утверждения не удовлетворяет TEST-аргументу, поэтому данное утверждение «отвергается», присваивания переменным X и Y отменяются, и функция SEARCH продолжает поиск в базе данных. Теперь она находит утверждение (СТУДЕНТ ИВАНОВ) и убеждается, что его список свойств удовлетворяет TEST-аргументу. Поэтому функция успешно заканчивает свою работу со

значением (СТУДЕНТ ИВАНОВ); переменные X и Y имеют соответственно значения ИВАНОВ и 5.

Вычисление [SEARCH (СТУДЕНТ *X)] также успешно, однако здесь нельзя заранее предсказать, какое именно значение — ИВАНОВ или ПЕТРОВ — получит переменная X. Все зависит от того, какое из утверждений (СТУДЕНТ ИВАНОВ) или (СТУДЕНТ ПЕТРОВ) будет найдено первым. В языке не фиксируется порядок просмотра утверждений базы данных, но предполагается, что он определяется транслятором языка и детерминирован (при повторном выполнении программы утверждения базы данных просматриваются в том же порядке, что и при первом выполнении программы). Отметим также, что дважды одно и то же утверждение не рассматривается функцией SEARCH и что утверждения, записанные в базу данных после того, как уже произошло обращение к функции SEARCH, ею при возобновлении поиска не учитываются.

Другие примеры:

```
[FIND ALL (X) .X [SEARCH (СТУДЕНТ *X)]]  →
    (ИВАНОВ ПЕТРОВ)
[SEARCH (*Y СЕМЕНОВ)]  →  (АСПИРАНТ СЕМЕНОВ),
    Y:= АСПИРАНТ
[SEARCH (СТУДЕНТ [ ]) (TEST КУРС 3)] — неуспех
[FIND ALL (X) .X
    [SEARCH ([ ] *X) (TEST ФАКУЛЬТЕТ ВМК)]]  →
    (ПЕТРОВ СЕМЕНОВ)
```

Вычисление первой из этих функций можно трактовать как поиск всех лиц, о которых в базе данных сказано, что они студенты. Во втором примере узнается, кем является Семенов. Третий пример — это проверка, есть ли в базе данных информация о студентах третьего курса. В четвертом примере находятся все лица, о которых известно, что они учатся на факультете ВМК.

Функция SEARCH1: [SEARCH1 *pat test?*], FSUBR.

Эта функция действует аналогично функции SEARCH, но не ставит F-точку, поэтому при неуспехе функция SEARCH1 не возобновляет свою работу. Если эта функция один раз нашла подходящее утверждение, то никакие другие утверждения ее не интересуют. В том случае, когда функция не нашла ни одного подходящего утверждения, она вырабатывает неуспех с сообщением SEARCH1.

Предположим, к примеру, что в базе данных записаны утверждения вида (ГОРОД *x*), означающее, что *x* — город, утверждения вида (ПОСЕЛОК *x*), означающее, что *x* — поселок, и утверждения вида (СТРАНА *x y*), означающее, что *x* находится в

стране *y*. Тогда найти все города страны Утопия можно с помощью выражения

```
[FIND ALL (X) .X [SEARCH (ГОРОД *X)]
      [SEARCH1 (СТРАНА .X УТОПИЯ)]]
```

Первый оператор здесь находит название какого-либо города, а второй оператор проверяет, расположен ли этот город в стране Утопия. Если не расположен, тогда функция SEARCH1 вырабатывает неуспех, поэтому название этого города не запоминается функцией FIND, в противном же случае — запоминается, после чего FIND вырабатывает неуспех. Этот неуспех не должен возобновлять работу второго оператора, так как заранее известно, что в базе данных для каждого населенного пункта *x* имеется только одно утверждение вида (СТРАНА *x* УТОПИЯ), и искать другое такое же утверждение бессмысленно. Поэтому-то во втором операторе указана функция SEARCH1, а не SEARCH. В первом же операторе использована функция SEARCH, поскольку при неуспехе этот оператор должен возобновить свою работу и найти название другого города.

Еще одной функцией, осуществляющей поиск утверждений, является

Функция CANDIDATES: [CANDIDATES *pat type*?], FSUBR.

На первый аргумент этой функции накладываются те же ограничения, что и в функции SEARCH, а аргумент *type*, который вычисляется, должен быть простой формой. Если аргумента *type* нет или если его значением является атом ASSERT*), то функция в качестве своего значения выдает список всех утверждений базы данных, соответствующих образцу *pat*, вместе с их списками свойств. Возможный пример:

```
[CANDIDATES (BOX .X)]  →  ((BOX A) (SIZE 3)
    (BOX B) () (BOX C) (SIZE 5 COLOUR RED))
```

Следует отметить, что функция CANDIDATES не выполняет полного сопоставления образца *pat* с утверждениями. Дело в том, что такое сопоставление может иметь побочные эффекты, поэтому сопоставление образца *pat* с разными утверждениями происходило бы в разных условиях. Например, если переменная X не имела значения, то после сопоставления образца (BOX .X) с первым подходящим утверждением, скажем с (BOX A), переменная X получила бы значение A, после чего образец не соответствовал бы уже ни утверждению (BOX B), ни утверждению (BOX C). Чтобы не бы-

*) О действиях функции при других значениях аргумента *type* см. § 5.5.

ло таких ненужных эффектов, данная функция осуществляет лишь частичное сопоставление, при котором не вычисляются никакие функции и сопоставители из образца *pat*, не учитываются значения никаких переменных образца и никаким переменным ничего не присваивается. Проверяется лишь соответствие атомов и длин L-списков с верхнего уровня образца *pat*. Например, при таком частичном сопоставлении образец (ON [BLOCK] .X) соответствует утверждению (ON BLOCK TABLE) или утверждению (ON PRISM CUBE), но не соответствует утверждениям (ABOVE BLOCK TABLE) и (ON PRISM), а образец (NOT (IS MAN *X)) соответствует (NOT (AT A R1)) и не соответствует (NOT WOMAN). Утверждения, которые соответствуют образцу *paf* при этом частичном сопоставлении, являются кандидатами на удачное полное сопоставление с данным образцом, и именно список таких утверждений и выдается в качестве значения функции CANDIDATES.

Данная функция используется в тех случаях, когда требуется определить операцию поиска по образцу, отличную от той, что реализуется функцией SEARCH. Например, можно следующим образом определить функцию SEEK, которая действует аналогично функции SEARCH (без TEST-аргумента), но возобновляет свою работу только при неуспехе, с которым связано сообщение REPEAT:

```
[DEFINE SEEK (LAMBDA (*P-) [PROG (LA- A- PL-)
    [PSET LA- [CANDIDATES .P-]]
A [ALT ( ) [COND ([EQ [MESS] REPEAT] [GO A])
                 (T [FAIL [MESS]])]]
B [COND ([EMPTY .LA-] [FAIL SEEK])]
    [PERM [IS (*A- *PL- !*LA-) .LA-]]
    [COND ([IS [PAT .P-] .A-] .A-)
          (T [GO B])] ])]
```

4.5. Другие операции

Следующие три встроенные функции осуществляют операции над списками свойств утверждений.

Функция GETA: [GETA *asrt ind*?], SUBR.

Эта функция действует аналогично функции GET (см. § 1.17), но по отношению к списку свойств утверждения, совпадающего со значением аргумента *asrt*. Таким образом, функция GETA в качестве своего значения выдает весь список свойств этого утверждения, если при обращении к ней не задан второй аргумент, либо выдает значение только одного свойства — того, чье название указано аргументом *ind*. Если в базе данных нет нужного утверждения, функция вырабатывает неуспех с сообщением GETA.

Например, если значением переменной X является список (АРКА С А В), то после выполнения

```
[ASSERT .X
    (WITH СВЕРХУ С СЛЕВА А СПРАВА В)]
```

имеем

```
[GETA .X СЛЕВА]  →  A
[GETA (АРКА С А В) СНИЗУ]  →  ( )
[GETA .X]  →  (СВЕРХУ С СЛЕВА А СПРАВА В)
```

Функция PUTA: [PUTA *asrt ind v*], SUBR.

Данная функция, аналогичная функции PUT (см. § 1.17), заменяет в списке свойств утверждения, заданного аргументом *asrt*, прежнее значение свойства с названием *IND* на новое значение *V*. Попутно запоминается обратный оператор, который при распространении неуспеха восстановит прежнее значение данного свойства. Если нужного утверждения в базе данных нет, то функция ничего не меняет и вырабатывает неуспех с сообщением PUTA. Пример:

```
[DO [PUTA .X СВЕРХУ D] [GETA .X]]  →
    (СВЕРХУ D СЛЕВА А СПРАВА В)
```

Функция PPUTA: [PPUTA *asrt ind v*], SUBR.

Это — «двойник» функции PUTA, при выполнении которого обратный оператор не запоминается.

Следующая функция используется в тех случаях, когда нужно распечатать содержимое всей базы данных.

Функция DUMP: [DUMP].

Значением функции является целое число — количество утверждений, имеющихся в настоящий момент в базе данных. Кроме того, функция выводит на печать в порядке возрастания длин все утверждения базы данных вместе с их списками свойств. Возможный вид печати:

```
(БЛОК А)  (ЦВЕТ СИНИЙ ФОРМА КУБ)
(БЛОК В)  (ФОРМА КУБ)
(БЛОК С)  (ФОРМА ПРИЗМА ЦВЕТ БЕЛЫЙ)
(ОПОРА В С)  ( )
(ОПОРА А С)  ( )
(АРКА С А В)  (СВЕРХУ С СЛЕВА А СПРАВА В)
```

Если говорить более точно, то функция DUMP записывает утверждения и их списки свойств в активный файл вывода (см. § 1.15). Поэтому, используя эту функцию, можно записать текущую базу данных в некоторый файл, расположенный во внешней

памяти ЭВМ, чтобы затем воспользоваться этой базой данных в какой-то другой плэнерской программе. Осуществить такую запись в файл FILE можно, например, с помощью следующей функции:

```
[DEFINE STORE (LAMBDA (FILE) [PROG (AF)
    [SET AF [ACTIVE PUT]]
    [OPEN .FILE PUT] [ACTIVE .FILE PUT]
    [DUMP] [PRINT END]
    [CLOSE .FILE PUT] [ACTIVE .AF PUT]])]
```

Считывание базы данных из файла FILE можно описать в виде следующей функции:

```
[DEFINE RESTORE (LAMBDA (FILE) [PROG (AF A PL)
    [SET AF [ACTIVE GET]]
    [OPEN .FILE GET] [ACTIVE .FILE GET]
    [WHILE [NEQ [SET A [READ]] END]
        [SET PL [READ]] [DB .A .PL]]
    [CLOSE .FILE GET] [ACTIVE .AF  GET] ])]
```

ГЛАВА 5

ТЕОРЕМЫ

5.1. Основные понятия

В языках программирования процедуры обычно вызываются по имени: в тексте программы явно указывается имя процедуры, которая должна быть выполнена на очередном шаге вычислений. В плэнере так вызываются функции и сопоставители. Но в плэнере возможен и иной способ вызова — *вызов по образцу*, при котором не указывается, какая конкретно процедура должна быть выполнена, а указывается лишь свойство, которым должна обладать вызываемая процедура. Так в языке вызываются процедуры, называемые *теоремами*. Каждая теорема кроме тела имеет еще и образец, используемый при ее вызове. Вызов происходит так: в программе задается некоторый образец (будем называть его вызывающим) и среди существующих теорем выбирается та, чей образец соответствует этому образцу; данная теорема и вызывается — вычисляется ее тело.

Такой «безымянный» вызов является более гибким, чем вызов по имени. При вызове по образцу выбор теорем осуществляется во время вычисления программы, а не при написании ее. Поэтому теоремы могут быть добавлены к программе позже, могут быть даже построены в процессе вычисления самой программы, и при этом текст остальной части программы не должен меняться.

Вызов теорем по образцу сочетается с режимом возвратов, так как вызывающему образцу могут соответствовать образцы нескольких теорем. Из всех этих теорем выбирается какая-нибудь одна, и она вычисляется. Если ее вычисление успешно, то остальные теоремы не рассматриваются, и вызов по образцу успешно завершается. Но если вычисление теоремы оказалось неуспешным, то она «отвергается», и вызывается другая теорема с подходящим образцом. Если и она неуспешна, вызывается третья

теорема и т. д. В конце концов либо будет найдена теорема, вычисление которой успешно, и тогда вызов по образцу успешен, либо будет установлено, что все теоремы, образцы которых соответствуют вызывающему образцу, вырабатывают неуспех. В последнем случае, как и в случае, когда теорем с подходящим образцом вообще нет, вызов по образцу считается неуспешным.

Использовать теоремы и описанный способ их вызова можно по-разному, но в язык они введены ради следующей цели.

Как уже было сказано, плэнер предоставляет средства для описания задач, решение которых должны находить системы искусственного интеллекта, и средства для реализации процедур поиска решения этих задач. Кроме рассмотренной в предыдущей главе базы данных, к таким средствам относятся и теоремы, которые используются для описания «правил игры» в решаемой задаче, т. е. тех операций, которые разрешено применять при решении задачи: правил логического вывода, элементарных действий робота, ходов в игровых задачах, правил нахождения ответов на простейшие вопросы и т. п. Каждое такое правило описывается в виде теоремы, образец которой указывает, что́ можно получить в результате применения данного правила, а ее тело определяет действия, которые следует выполнить для получения этого результата. Таким образом, образец теоремы описывает цель, а ее тело — способ достижения этой цели. Если среди используемых операций есть такие, применение которых приводит к одному и тому же результату, то они описываются теоремами с одинаковыми образцами. Эти теоремы предлагают разные способы достижения одной и той же цели.

При таком использовании теорем цель решаемой задачи может быть описана в виде вызывающего образца, и тогда решение задачи будет найдено автоматически. В самом деле, при вызове теорем по этому образцу-цели транслятор языка подыскивает теорему, которая своим образцом «обещает» получить нужный результат. Тело такой теоремы описывает алгоритм достижения цели, поэтому, выполняя его, транслятор и находит решение задачи. Может, правда, оказаться, что в текущей обстановке предложенный теоремой путь достижения цели не приводит к успеху, тогда транслятор отказывается от этой теоремы и отыскивает другую подходящую теорему. Так, опираясь на имеющиеся теоремы и сочетая вызов по образцу с режимом возвратов, транслятор языка и осуществляет поиск решения задачи, описанной пользователем.

Не следует, однако, думать, что поиск решения задачи заключается лишь в выборе одной подходящей теоремы. Как правило, теоремы в свою очередь ставят цели, поэтому при дости-

жении одной цели обычно возникает целая иерархия подцелей, для достижения каждой из которых транслятор подыскивает свою подходящую теорему. Такой механизм достижения цели путем сведения ее к подцелям принято называть *дедуктивным механизмом.*

Проиллюстрируем сказанное на простом примере. Предположим, что на столе построена башня из кубиков. Если факт «x находится на y» представляется утверждением (ON x y), то ситуация, изображенная на рис. 6, описывается базой данных из трех утверждений: (ON A B), (ON B C) и (ON C TABLE).

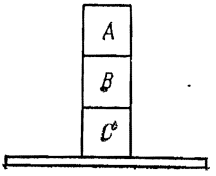


Рис. 6.

Предположим также, что понятие «выше» определяется через понятие «находиться на» следующим образом:

$on(x, z) \supset above(x, z)$

$on(x, y) \wedge above(y, z) \supset above(x, z)$

Первое из этих предложений может быть описано в виде плэнерской теоремы (назовем ее TH1) с образцом:

```
(ABOVE *X *Z)
```

и телом из одного оператора:

```
[SEARCH (ON .X .Z)]
```

Образец этой теоремы указывает, что с ее помощью можно показать, что некоторый кубик X расположен выше некоторого кубика Z, а ее тело говорит, что для этого достаточно найти в базе данных утверждение о том, что X находится на Z.

Второе предложение можно описать в виде теоремы (обозначим ее TH2), которая имеет аналогичный образец:

```
(ABOVE *X *Z)
```

но другое тело:

```
[SEARCH (ON .X *Y)]
[ACHIEVE (ABOVE .Y .Z)]
```

Это тело предлагает иной способ доказательства того, что X выше Z: сначала надо найти тот кубик Y, на котором находится X, а затем надо доказать, что Y выше Z. ACHIEVE — это встроенная функция плэнера, которая осуществляет вызов теорем по образцу, являющемуся ее аргументом. Таким образом, для доказательства того, что Y выше Z, предлагается вызвать любую теорему, которая своим образцом «обещает» это сделать (такой теоремой может быть и сама теорема TH2).

Теперь предположим, что мы хотим доказать, что кубик А расположен выше кубика С. Такая цель задается выражением

[ACHIEVE (ABOVE A C)]

На этом описание нашей задачи заканчивается. Решение же ее будет найдено автоматически в результате вычисления указанного обращения к функции ACHIEVE, которое осуществляется следующим образом.

Функция ACHIEVE вызывает любую из теорем, образцы которых соответствуют образцу (ABOVE A C). Таких теорем две: TH1 и TH2. Предположим, что первой вызвана теорема TH1. При сопоставлении ее образца с вызывающим образцом переменная X получает значение A, а переменная Z — значение C. Далее вычисляется тело теоремы, т. е. обращение к функции SEARCH, которая пытается найти в базе данных утверждение (ON A C). Такого утверждения там нет, поэтому вычисление этой функции и, следовательно, тела теоремы неуспешно. Это означает, что хотя теорема TH1 и «обещала» доказать, что А выше С, но сделать это она не смогла. Поэтому функция ACHIEVE отказывается от теоремы TH1 и теперь вызывает теорему TH2. Здесь также при сопоставлении образцов переменная X получает значение A, а переменная Z — значение C. Далее вычисляется тело теоремы TH2. Сначала функция SEARCH находит в базе данных утверждение (ON A B), соответствующее ее образцу (ON .X *Y), и при этом переменной Y присваивается значение B. Затем функция ACHIEVE ставит цель: доказать, что B выше C. Для этого вызывается теорема, образец которой соответствует образцу (ABOVE B C). Предположим, что вызвана теорема TH1. Тогда ее переменные X и Z получают соответственно значения B и C. На этот раз вычисление тела теоремы TH1 успешно, так как в базе данных есть утверждение (ON B C). Поэтому успешна и функция ACHIEVE, вызвавшая ее. Поскольку обращение к функции ACHIEVE — последний оператор в теле теоремы TH2, то на этом успешно завершается вычисление теоремы TH2 и вычисление исходного выражения [ACHIEVE (ABOVE A C)]. То, что вычисление этого выражения закончилось успехом, и означает, что поставленная цель — доказать, что А выше С,— достигнута. Если бы вычисление этого выражения было неуспешным, то это значило бы, что высказывание «А выше С» ложно или, если говорить более осторожно, недоказуемо при имеющихся фактах (базе данных) и имеющихся правилах вывода (теоремах TH1 и TH2).

Рассмотренный пример показывает, что программировать на плэнере можно не только описывая алгоритм решения задачи, но и описывая лишь постановку задачи. В последнем случае

ответственность за поиск ее решения возлагается на транслятор языка. И если эта задача может быть решена, то транслятор в конце концов найдет решение. В этом смысле плэнер можно рассматривать не только как язык программирования, но и как систему искусственного интеллекта. Однако не следует увлекаться автоматическим поиском решения задач, поскольку он осуществляется методом полного перебора и потому поиск решения сложных задач занимает очень много времени. Более разумным является способ программирования, при котором пользователь управляет работой дедуктивного механизма.

Пользователь обычно знает, какие теоремы являются наиболее подходящими для достижения определенных целей, а какие теоремы, напротив, в данных конкретных условиях не подойдут. Для того чтобы он мог передавать такие знания дедуктивному механизму, язык предоставляет пользователю возможность давать *рекомендации* относительно того, какие теоремы следует вызывать, а какие нет, в каком порядке следует их вызывать. Например, при следующем вызове теорем по образцу

[ACHIEVE (SUBSET A B) (USE TH5 [NON TH7])]

указана рекомендация (список, начинающийся с USE), которая требует, чтобы в первую очередь была вызвана теорема TH5, а если она не подойдет, тогда могут быть вызваны любые другие теоремы, образцы которых соответствуют вызывающему образцу (SUBSET A B), но только не теорема TH7. Иногда рекомендации, которые пользователь может дать дедуктивному механизму, зависят от того, какая конкретная обстановка складывается при достижении цели, поэтому в плэнере предусмотрена возможность определения рекомендаций в процессе вычисления программы, разрешено и изменение ранее данных рекомендаций. В языке также предусмотрены способы борьбы с зацикливаниями при достижении целей. С помощью всех этих средств пользователь может направлять работу дедуктивного механизма в нужное русло, повышать его эффективность.

Кроме рассмотренных выше теорем, которые применяются при достижении целей и потому называются *целевыми*, в плэнере имеются также *записывающие* и *вычеркивающие* теоремы. Эти теоремы определяются, вызываются и вычисляются так же, как и целевые теоремы; различаются лишь ситуации, в которых они вызываются. Записывающие теоремы вызываются тогда, когда в базу данных записываются утверждения, соответствующие их образцам, а вычеркивающие теоремы — при вычеркивании из базы данных утверждений, соответствующих их образцам. Назначение этих теорем — контролировать изменения базы данных. Они

могут проверять «законность» (с точки зрения условий решаемой задачи) производимых записей и вычеркиваний или выполнять сопутствующие им изменения базы данных. Например, записывающая теорема с образцом

(*X ПРОДАЛ *Y)

и телом

[ERASE (.X ИМЕЕТ .Y)]

вызывается тогда, когда в базу данных записывается утверждение о том, что некий X продал нечто Y. Действие теоремы заключается в том, что она попутно с этой записью вычеркивает из базы данных ставшее теперь ложным утверждение о том, что этот X имеет Y.

Как и другие процедуры языка, теоремы имеют имена, и любую теорему можно вызвать по имени (при этом вызывающий образец играет роль фактического параметра). Однако в таких случаях более уместным будет использование функций, а не теорем. Кроме того, теоремы, как и утверждения, могут иметь списки свойств, где указываются дополнительные сведения о теоремах. Имена и списки свойств теорем нужны в основном для того, чтобы в рекомендациях можно было ссылаться на конкретные теоремы и указывать характеристики, которые должны иметь вызываемые теоремы.

5.2. Определение теорем

В языке нет встроенных теорем. Любая теорема, которая будет использоваться в программе, должна быть определена. Для этого нужно, как и при определении функций и сопоставителей, обратиться к встроенной функции DEFINE, которая в данном случае может иметь три аргумента:

[DEFINE *th de pl*?].

Здесь *th* обозначает идентификатор, указывающий имя определяемой теоремы. Аргумент *pl* задает список свойств для данной теоремы *); этот аргумент не вычисляется и должен иметь вид

$(IND_1\ V_1\ IND_2\ V_2\ \ldots\ IND_n V_n)$, $n \geqslant 1$

где IND_i — название (идентификатор), а V_i — значение (любое

*) Этот список свойств не имеет никакого отношения к списку свойств идентификатора *th*; это разные списки свойств.

выражение) i-го свойства. Если аргумент *pl* не задан, то с теоремой связывается пустой список свойств.

Аргумент *de* называется *определяющим выражением* теоремы и должен иметь вид

$$(type\ (v_1\ v_2\ \ldots\ v_m)\ pat\ e_1\ e_2\ \ldots\ e_k),\ m \geqslant 0,\ k \geqslant 1.$$

Список $(v_1\ v_2\ \ldots\ v_m)$ — это *описание локальных переменных* теоремы, а последовательность простых форм e_i — *тело* теоремы. Описание локальных переменных и тело теоремы — точно такие же, как и у функции PROG (см. § 1.9). Любая теорема вычисляется так же, как функция PROG, но вызывается не по имени, а по образцу.

Элемент *pat* — это образец теоремы. Он обязательно должен быть L-списком, не содержащим обращений к функциям и сопоставителям, а также никаких сегментных образцов. Таким образом, элементами этого спискового образца могут быть только атомы, простые обращения к переменным и константам и L-списки с теми же ограничениями. Подобные образцы мы будем называть *T-образцами.*

Элемент *type* определяет тип описываемой теоремы и может быть одним из следующих атомов: CONSEQ (от слова consequent), ANTEC (от antecedent) или ERASING. При CONSEQ определяется целевая теорема, при ANTEC — записывающая теорема, при ERASING — вычеркивающая теорема.

Рассмотрим несколько примеров:

```
[DEFINE TH5
  (CONSEQ (X) (ANIMAL *X)
      [SEARCH (MONKEY .X)])
   (DIFFICULTY EASY)]
```

Здесь определена целевая теорема с именем TH5, которая имеет одну локальную переменную X и образец (ANIMAL *X). Тело этой теоремы состоит из одного оператора — обращения к функции SEARCH. С теоремой связывается свойство с названием DIFFICULTY и значением EASY. Содержательно эту теорему можно трактовать так: если надо доказать, что X — животное, то для этого достаточно показать, что X — обезьяна. Свойство же теоремы может означать, что сделать это несложно.

```
[DEFINE ПОЕЗДКА (ANTEC (X Y Z)
          (*X УЕХАЛ В *Y)
    [SEARCH1 (.X НАХОДИТСЯ В *Z)]
    [ERASE (.X НАХОДИТСЯ В .Z)]
    [ASSERT (.X НАХОДИТСЯ В .Y)])]
```

У этой записывающей теоремы с именем ПОЕЗДКА три локальных переменных X, Y и Z. Ее образец — (*X УЕХАЛ В *Y), а тело ее состоит из трех операторов. С теоремой связывается пустой список свойств. Как уже было сказано, записывающие теоремы вызываются в тех случаях, когда в базу данных записываются утверждения, соответствующие их образцам. Теорема ПОЕЗДКА вызывается при занесении в базу данных информации о том, что некий X уехал в некоторое место Y. Теорема требует, чтобы попутно из базы данных было вычеркнуто утверждение о прежнем местонахождении этого X и было записано утверждение о том, что X находится теперь в Y.

```
[DEFINE БАНКРОТ (ERASING (X) (БОГАТ *X)
    [ASSERT (БЕДЕН .X)])]
```

Данная вычеркивающая теорема предназначена для вызова в тех случаях, когда из базы данных вычеркивается утверждение о том, что некто X богат. Теорема требует, чтобы одновременно с этим в базу данных было записано утверждение о том, что этот X беден.

5.3. Вызов теорем

Теоремы вызываются при вычислении встроенных функций ACHIEVE, DRAW и CHANGE. Обращения к этим функциям и их действия аналогичны, но каждая из них вызывает теоремы только «своего» типа: функция ACHIEVE вызывает целевые теоремы, функция DRAW — записывающие теоремы, а функция CHANGE — вычеркивающие теоремы. Поэтому механизм вызова теорем мы рассмотрим на примере функции ACHIEVE.

Функция ACHIEVE: [ACHIEVE *pat rec*?], FSUBR.

Эта функция вызывает целевые теоремы по образцу *pat* согласно рекомендации *rec*.

Аргумент *pat* должен быть T-образцом или .-переменной, значением которой является T-образец. Рекомендация же *rec* должна быть одной из следующих ($k \geqslant 0$):

(USE r_1 r_2 ... r_k)
(USE1 r_1 r_2 ... r_k)
(TRY r_1 r_2 ... r_k)
(FILTER l)

Элементы рекомендации — r_i — должны быть простыми образцами, а l — простой формой. Отсутствие рекомендации эквивалентно заданию рекомендации (USE); рекомендации (USE), (USE1) и (TRY) рассматриваются как сокращения от (USE [ ]), (USE1 [ ]) и (TRY [ ]) соответственно.

Для краткости рекомендацию, начинающуюся с USE, будем называть USE-рекомендацией, рекомендацию, начинающуюся с USE1, — USE1-рекомендацией и т. п.

Если при обращении к функции ACHIEVE указана FILTER-рекомендация, то функция начинает свою работу с того, что вычисляет форму l из этой рекомендации. Значением данной формы должна быть рекомендация, начинающаяся с USE, USE1 или TRY, либо пустой список ( ). Последний вариант означает, что теоремы не должны вызываться, поэтому функция сразу заканчивает свою работу со значением ( ). Другие значения этой формы определяют рекомендации, согласно которым и будут вызываться теоремы. FILTER-рекомендация предоставляет возможность строить рекомендации в процессе выполнения программы, если они не были известны заранее, при написании программы.

Просмотр теорем. После вычисления FILTER-рекомендации, а если ее нет, то сразу, функция ACHIEVE ставит F-точку и затем составляет так называемый *список возможностей,* в который (в порядке, определяемом реализацией языка) заносит имена всех целевых теорем, являющихся кандидатами на вызов по образцу *pat.*

Теорема является кандидатом, если ее образец соответствует вызывающему образцу в предположении, что любым переменным из обоих этих образцов всегда есть соответствие и что подспискам с верхнего уровня образцов соответствуют любые подсписки той же длины; при таком частичном сопоставлении никакие присваивания переменным не осуществляются. В этом смысле образец (AT .X (*Y 1)) соответствует образцу (AT BOX *Z) или образцу (.P *W (2 4)), но не соответствует образцам (ON BOX *Z) и (AT *W R1).

Список возможностей определяет, какие теоремы в принципе могут быть вызваны по образцу *pat.* Но вот какие из них действительно будут вызываться и в каком порядке, определяет рекомендация *rec.* Для этого элементы рекомендации (образцы r_i) просматриваются последовательно слева направо, и для каждого из них выполняются следующие действия.

Образец r_i по очереди сопоставляется с именами из списка возможностей. Подчеркнем, что сопоставление производится именно с идентификаторами-именами. Единственное исключение — встроенный сопоставитель TEST:

[TEST ind_1 pat_1 ... ind_k pat_k], FSUBR, $k \geqslant 1$,

который действует аналогично сопоставителю HAS (см. § 2.6), но проверяет не список свойств идентификатора-имени, а список свойств теоремы с этим именем.

Если нашлось имя, соответствующее образцу r_i, то оно удаляется из списка возможностей, а теорема с этим именем вызывается. Если она не подошла (при полном сопоставлении ее образец не соответствует вызывающему образцу или оказалось неуспешным вычисление ее тела), тогда побочные эффекты сопоставления ее имени с образцом r_i уничтожаются, теорема отвергается и в списке возможностей отыскивается другое имя, соответствующее образцу r_i. Когда этот список будет просмотрен полностью, элемент r_i удаляется из рекомендации *rec* и теперь все описанные действия повторяются по отношению к следующему элементу рекомендации. Если в рекомендации больше нет элементов, вызов теорем прекращается.

Поскольку имя вызванной теоремы удаляется из списка возможностей, повторно эта теорема уже не будет вызываться. Однако это не всегда удобно, поэтому из описанного выше правила сделано исключение: если r_i — идентификатор, т. е. он явно указывает имя теоремы, которую рекомендуется вызвать, то эта теорема (при условии, конечно, что она целевая) вызывается в любом случае, есть ее имя в списке возможностей или нет.

Проиллюстрируем правило просмотра теорем на примере:

```
(USE TH6
    [ET [NON TH1] [TEST CLASS A] *X]
    TH9).
```

Пусть исходный список возможностей имеет вид (TH1 TH2 TH5 TH6 TH9). Выбирается первый элемент рекомендации — TH6. Этот атом удаляется из списка возможностей, и вызывается теорема с именем TH6. Предположим, что она не подошла. Тогда рассматривается следующий элемент рекомендации — обращение к сопоставителю ET. Он по очереди сопоставляется с оставшимися идентификаторами из списка возможностей. Ему соответствует имя любой, кроме TH1, теоремы, у которой свойство с названием CLASS имеет значение A; кроме того, при сопоставлении имя теоремы присваивается переменной X, чем можно будет воспользоваться в дальнейшем. Пусть первым из таких имен был идентификатор TH2. Тогда этот идентификатор удаляется из списка возможностей, и затем вызывается теорема TH2. Если она не подошла, то определяется другое имя, соответствующее сопоставителю ET. Если это, например, TH9, то вызывается теорема с этим именем, а само имя удаляется из списка возможностей, который теперь принимает вид (TH1 TH5). Предположим, что и теорема TH9 не подошла. Поскольку для данного элемента рекомендации список возможностей просмотрен полностью, далее рассматривается последний элемент рекомендации — TH9. Это — явно

указанное имя теоремы, поэтому, хотя такого имени уже нет в списке возможностей, теорема TH9 все равно вызывается. Если она вновь не подошла, вызов теорем завершается. Теоремы же TH1 и TH5 так и не будут вызваны.

Отметим важный случай вызова теорем. Если при обращении к функции ACHIEVE не была указана рекомендация или если была дана рекомендация без элементов, типа (USE), то вызываться будут все теоремы-кандидаты в том порядке, в котором расположены их имена в списке возможностей, т. е. в порядке, определяемом транслятором языка.

Вызов теоремы. Рассмотрим теперь, как осуществляется вызов выбранной теоремы.

Прежде всего вводятся локальные переменные этой теоремы. Некоторые из них, согласно описанию, получают начальные значения, а другие остаются без значения. Далее осуществляется сопоставление вызывающего образца с образцом теоремы. Детали этого сопоставления будут рассмотрены в § 5.4, а пока лишь отметим, что в вызывающем образце используются переменные, существовавшие до появления локальных переменных теоремы, а в образце теоремы используются ее локальные переменные (допускается также и применение внешних по отношению к теореме переменных). Если данное сопоставление неудачно, то на этом вызов теоремы завершается — теорема не подошла; ее локальные переменные уничтожаются. Но если сопоставление удачно, тогда вычисляется тело теоремы. Это вычисление начинается при тех значениях и ограничениях на значения (см. § 5.4) переменных, которые они получили при сопоставлении. В данном случае успех/неуспех вызова теоремы зависит от того, успешно или нет вычисление ее тела. Если оно неуспешно, то при распространении неуспеха будут отменены побочные эффекты как операторов тела теоремы, так и сопоставления ее образца с вызывающим образцом. При любом исходе по окончании вычисления тела теоремы ее локальные переменные уничтожаются.

Рассмотрим следующий пример. Пусть при вычислении

```
[PROG ((Y 5) E)
   [ACHIEVE (БЛИЗКО .Y 4.95 *E)]]
```

для вызова выбрана теорема с таким определяющим выражением:

```
(CONSEQ (X Y (EPS 0.1))
        (БЛИЗКО *X *Y .EPS)
   [UNFALSE [LT [ABS [— .X .Y]] .EPS]])
```

При вызове этой теоремы прежде всего будут введены ее локальные переменные: X и Y без значений, а EPS — со значением 0.1. (Отметим, что переменная Y из теоремы не имеет никакого от-

ношения к переменной Y из функции PROG.) Затем выполняется сопоставление образцов, причем переменные Y и E вызывающего образца берутся из функции PROG, а под переменными X, Y и EPS образца теоремы понимаются ее локальные переменные. Сопоставление удачно, при этом переменная E получает значение 0.1, а локальные переменные X и Y теоремы — значения 5 и 4.95 соответственно. С этими значениями переменных и начинается вычисление тела теоремы. Вычисление успешно, поэтому вызов теоремы завершается успехом. Ее локальные переменные уничтожаются, а за переменной E функции PROG сохраняется значение 0.1.

Если бы значением переменной EPS было число 0.01, то сопоставление образцов также было бы удачным (только переменная E получила бы значение 0.01), но вот вычисление тела теоремы оказалось бы неуспешным. Распространение появившегося при этом неуспеха (до F-точки функции ACHIEVE) привело бы к отмене побочных эффектов сопоставления, поэтому переменная E снова оказалось бы без значения.

Типы рекомендаций. Выбранная для вызова теорема, как уже сказано, может не подойти по двум причинам: либо ее образец не соответствует вызывающему образцу, либо соответствие между образцами есть, но при вычислении тела теоремы вырабатывается неуспех. В первом случае функция ACHIEVE при любой рекомендации переходит к выбору следующей подходящей теоремы. Но во втором случае, а также в случае успешного вычисления тела теоремы дальнейшие действия функции ACHIEVE зависят от типа данной ей рекомендации.

Если дана USE-рекомендация и вызов очередной теоремы закончился неуспехом, то этот неуспех будет «пойман» F-точкой, поставленной функцией в начале своей работы, после чего функция переходит к выбору следующей теоремы. Если ни одна из рекомендованных теорем не подошла, тогда вызов по образцу неуспешен: функция ACHIEVE уничтожает свою F-точку и вырабатывает неуспех с сообщением ACHIEVE. Но если вычисление выбраной теоремы оказалось успешным, тогда при USE-рекомендации остальные теоремы не рассматриваются, и функция с успехом заканчивает свою работу. Ее значением объявляется значение «успешной» теоремы.

Однако F-точка, поставленная функцией ACHIEVE, сохраняется. Поэтому если затем в программе возникнет неуспех, по которому произойдет возврат к этой F-точке, то функция возобновит свою работу: будет выбрана следующая рекомендованная теорема и т. д.— до первой «успешной» теоремы или до исчерпания рекомендации.

USE1-рекомендация аналогична USE-рекомендации за одним исключением: если найдена «успешная» теорема, то по окончании ее вычисления уничтожаются все F-точки внутри этой теоремы, а также F-точка самой функции ACHIEVE. Поэтому по неуспеху вызов теорем в данном случае возобновляться не будет.

Обе эти рекомендации (как и отсутствие рекомендаций) целесообразны при вызове целевых теорем, так как для достижения цели вполне достаточно найти лишь одну подходящую теорему. Может, правда, оказаться, что найденный способ достижения цели не удовлетворителен по каким-либо причинам (например, из-за него не удается достичь цели более высокого уровня). Тогда в зависимости от того, надо искать новый способ достижения цели или нет, применяется либо USE-, либо USE1-рекомендация.

Однако при вызове записывающих или вычеркивающих теорем такие рекомендации не очень удобны. Дело в том, что эти теоремы контролируют изменения в базе данных, причем каждое изменение может проверяться несколькими теоремами. Поэтому вполне естественным является требование, чтобы каждое изменение базы данных проходило все эти проверки и чтобы все они были успешными. Этому требованию удовлетворяет TRY-рекомендация, согласно которой должны быть вызваны все рекомендованные теоремы с подходящими образцами и вычисление каждой из них должно быть успешным. Только при этих условиях вызов теорем будет считаться успешным.

Более точно, функция, вызывающая теоремы, действует при TRY-рекомендации следующим образом. Если очередная выбранная теорема не подошла из-за того, что ее образец не соответствует вызывающему образцу, то функция выбирает следующую рекомендованную теорему. Если же сопоставление образцов удачно, но вычисление тела теоремы неуспешно, тогда выработанный при этом неуспех не «ловится» функцией (при TRY-рекомендации функция не ставит F-точку), остальные теоремы не рассматриваются и работа функции оканчивается неуспехом. В случае успешного вычисления вызванной теоремы все F-точки внутри нее уничтожаются, после чего вызывается следующая рекомендованная теорема. При TRY-рекомендации вызов теорем считается успешным, только если функция «добралась» до конца рекомендации. Значением функции в таком случае является атом T.

Примеры интерпретации рекомендаций:

(USE) — вызывать любую теорему с подходящим образцом до первой «успешной»;

(TRY) — вызывать все теоремы с подходящими образцами, и каждая из них должна быть успешной;

(USE TH5 [ ]) — вызвать в первую очередь теорему TH5, а если она не подойдет, вызывать любые другие теоремы в произвольном порядке;

(TRY [TEST CLASS 5]) — все теоремы 5-го класса с подходящими образцами должны быть успешными.

Функция DRAW: [DRAW *pat rec*?], FSUBR.

Эта функция вызывает записывающие теоремы по образцу *pat* согласно рекомендации *rec*. Функция действует аналогично функций ACHIEVE.

Функция CHANGE: [CHANGE *pat rec*?], FSUBR.

Данная функция, действуя аналогично функции ACHIEVE, вызывает вычеркивающие теоремы по образцу *pat* согласно рекомендации *rec*.

5.4. Сопоставление образца с образцом

При вызове теорем по образцу осуществляется сопоставление двух образцов. Например, образец (AT *X *Y) может сопоставляться с образцом (AT *Z (5 *W)). Такое сопоставление во многом отличается от сопоставления образца с выражением, и его правила будут объяснены в этом параграфе.

Сопоставление двух произвольных образцов является алгоритмически неразрешимой проблемой (хотя бы потому, что определение соответствия между сопоставителями — это проблема эквивалентности алгоритмов), поэтому в плэнере на образцы, которые сопоставляются друг с другом, накладываются ограничения: они не должны содержать сегментных элементов и обращений к функциям и сопоставителям. Именно по этой причине образцами теорем и вызывающими образцами могут быть не любые образцы, а только образцы с указанными ограничениями; их мы назвали Т-образцами.

В некоторых случаях сопоставление таких образцов сводится к знакомому нам сопоставлению образца с выражением. Например, при сопоставлении образцов (ON *X TABLE) и (ON A .Y) выполняются сопоставления ON с ON, *X с A и TABLE с .Y. Однако возможен и ряд новых случаев, когда переменные сопоставляются с переменными или со списками, содержащими переменные. Прежде чем сформулировать точные правила таких сопоставлений, проиллюстрируем эти новые случаи на конкретных примерах.

Если при удачном сопоставлении образца с выражением все переменные образца обязательно получают значения, то при сопоставлении двух образцов некоторые их переменные могут остаться без значений. Однако на будущие значения таких переменных накладываются определенные ограничения.

Например, сопоставление образца *X с образцом *Y считается удачным, так как *-переменные соответствуют чему угодно, но ни одна из переменных X и Y не получает значения, так как неизвестно, какое значение можно им присвоить. В то же время вполне разумно потребовать, чтобы будущие значения этих переменных совпадали, ибо только при таком условии можно считать, что образцы *X и *Y совпадают. Поэтому в результате сопоставления *X с *Y устанавливается *связь между переменными* X и Y. Это означает, что если в дальнейшем одна из переменных получит какое-либо значение, то автоматически такое же значение получит и другая переменная.

Предположим, что при вычислении выражения

```
[ACHIEVE (ПТИЦА *Y)]
```

которое можно трактовать как требование найти некий объект, являющийся птицей, вызывается следующая теорема:

```
[DEFINE TH1 (CONSEQ (X) (ПТИЦА *X)
    [SEARCH (ПОПУГАЙ .X)])]
```

которая определяет, что любой попугай является птицей. При сопоставлении вызывающего образца (ПТИЦА *Y) с образцом теоремы (ПТИЦА *X) переменным X и Y не присваиваются никакие значения, но между ними устанавливается связь. Далее вычисляется тело теоремы — обращение к функции SEARCH, которая находит в базе данных, скажем, утверждение (ПОПУГАЙ КЕША). При этом переменная X получает значение КЕША, которое одновременно становится и значением переменной Y. Вычисление теоремы на этом заканчивается, ее локальная переменная X ликвидируется, поэтому связь X с Y уничтожается. Однако значение КЕША у переменной Y сохраняется, оно-то и дает ответ на запрос «найти птицу».

Связи, устанавливаемые между переменными, транзитивны. Если, к примеру, установлена связь между переменными Z и Y, а затем устанавливается связь между Y и X, то переменная Z будет связана и с переменной X. Предположим, что имеется еще одна теорема:

```
[DEFINE TH2 (CONSEQ (Y)
          (ЖИВОЕ-СУЩЕСТВО *Y)
    [ACHIEVE (ПТИЦА .Y)])]
```

и поставлена цель «найти Z, являющееся живым существом»:

[ACHIEVE (ЖИВОЕ-СУЩЕСТВО *Z)].

Тогда сначала вызывается теорема TH2, переменная Y которой связывается с переменной Z, а затем вызывается теорема TH1, переменная X которой связывается с переменной Y, а тем самым и с переменной Z. Поэтому когда X получит значение КЕША, то такое же значение автоматически получат переменные Y и Z.

Несколько иная ситуация возникает при сопоставлении *-переменной со списком, содержащим хотя бы одну *-переменную. Предположим, например, что теорема TH1 вызывается по образцу (ПТИЦА (ВЕСЕЛЫЙ *Y)), тогда при сопоставлении этого образца с образцом теоремы (ПТИЦА *X) выполняется сопоставление *X с (ВЕСЕЛЫЙ *Y). Такое сопоставление также считается удачным, и переменные X и Y также не получают никаких значений. Однако здесь накладывается *ограничение на будущее значение переменной* X: оно должно быть списком из двух элементов, первый из которых — атом ВЕСЕЛЫЙ, а второй — произволен, но при этом обязан совпадать с будущим значением переменной Y (между Y и вторым элементом устанавливается связь). Поэтому если в дальнейшем при некотором сопоставлении переменной X будет присваиваться какое-то значение, то прежде всего будет проверено, удовлетворяет ли оно данному ограничению. Если нет, то присваивание не выполняется, а сопоставление заканчивается неудачей. Например, при сопоставлении образца (ПОПУГАЙ .X) функции SEARCH из теоремы TH1 с утверждением (ПОПУГАЙ КЕША) будет предпринята попытка присвоить переменной X значение КЕША, однако это значение не удовлетворяет ограничению на X, и потому сопоставление .X с КЕША неудачно. Неудачно и сопоставление образца с утверждением в целом, поэтому переменная X по-прежнему остается без значения. Но если значение, присваиваемое переменной X, удовлетворяет ограничению на эту переменную, тогда присваивание выполняется, и при этом второй элемент данного значения становится значением переменной Y. Например, сопоставление образца функции SEARCH с утверждением (ПОПУГАЙ (ВЕСЕЛЫЙ ЖЕКА)) удачно; переменная X получает значение (ВЕСЕЛЫЙ ЖЕКА), а переменная Y — значение ЖЕКА.

На будущее значение переменной может накладываться несколько ограничений. В таком случае требуется, чтобы новое ограничение не противоречило старому. Если это не так, то возникает неудача в том сопоставлении, где делается попытка наложить новое ограничение. Например, если вначале было выполнено сопоставление *X с (ВЕСЕЛЫЙ *Y), а затем образец .X сопоставляется с образцом (ГРУСТНЫЙ *Z), то второе сопоставление не-

удачно. Если же новое ограничение совместимо со старым, то берется «пересечение» обоих ограничений, и оно становится текущим ограничением на будущее значение переменной. Например, после сопоставлений *X с (ВЕСЕЛЫЙ *Y) и .X с (ВЕСЕЛЫЙ (*Z ПЕТЯ)) ограничение на переменную X будет таким: ее будущее значение должно быть списком из двух элементов, первый из которых — атом ВЕСЕЛЫЙ, а второй элемент, как и будущее значение переменной Y, обязан быть списком из любого элемента, совпадающего с будущим значением переменной Z, и атома ПЕТЯ.

Иногда «пересечение» ограничений может однозначно определить будущее значение переменной. В таком случае переменной автоматически присваивается это значение. Например, после сопоставлений *X с (ВЕСЕЛЫЙ *Y) и .X с (*Z ЖЕКА) переменная X получает значение (ВЕСЕЛЫЙ ЖЕКА); попутно переменной Y присваивается значение ЖЕКА, а переменной Z — значение ВЕСЕЛЫЙ.

Теперь рассмотрим взаимоотношения между ограничениями, накладываемыми на будущие значения переменных, и связями, устанавливаемыми между переменными. Если несколько переменных связаны между собой, а затем на одну из них накладывается ограничение, то это ограничение автоматически переносится и на остальные переменные. Если же, напротив, на несколько переменных были наложены ограничения, а затем между этими переменными устанавливается связь, тогда берется, если возможно, «пересечение» всех ограничений, и оно становится общим ограничением на все эти переменные. Если ограничения несовместимы, то связь установить не удастся.

Предположим, к примеру, что при вычислении выражения

[ACHIEVE (ЖИВОЕ-СУЩЕСТВО (ГРУСТНЫЙ *Z))]

вызывается теорема TH2. При сопоставлении ее образца (ЖИВОЕ-СУЩЕСТВО *Y) с вызывающим образцом на переменную Y накладывается такое ограничение: ее будущим значением может быть только список из атома ГРУСТНЫЙ и произвольного элемента, равного будущему значению переменной Z. При вычислении тела теоремы TH2 осуществляется вызов по образцу (ПТИЦА .Y) теоремы TH1, имеющей образец (ПТИЦА *X). При сопоставлении этих образцов между переменными Y и X устанавливается связь, поэтому на переменную X переносится ограничение, наложенное на переменную Y. Следовательно, образцу (ПОПУГАЙ .X) функции SEARCH не может соответствовать ни утверждение (ПОПУГАЙ КЕША), ни утверждение (ПОПУГАЙ (ВЕСЕЛЫЙ ЖЕКА)), но может соответствовать утверждение (ПОПУГАЙ

(ГРУСТНЫЙ ТОМ)). И если последнее утверждение действительно есть в базе данных, тогда переменная X получает значение (ГРУСТНЫЙ ТОМ); это же значение автоматически получает переменная Y, а значением переменной Z становится атом ТОМ.

И, наконец, отметим, что ограничения и связи каждой переменной сохраняют свою силу только до получения ею первого значения. После этого она может получать без всяких оговорок любые другие значения, и это никак не будет сказываться на остальных переменных. Дело в том, что связи и ограничения переменных, как и их значения, сохраняет только префикс «.»: если переменная входит в образец с этим префиксом, то при сопоставлении используется ее значение, а если его нет, принимаются во внимание ее связи и ограничение. Если же переменная входит в образец с префиксом «*», то ее прежнее состояние (значение, связи или ограничение) игнорируется. Такое правило действует при любых сопоставлениях, в том числе и при сопоставлении образца с выражением *). И поскольку изменить значение переменной, уже имеющей значение, можно, только обратившись к ней с префиксом «*», дальнейшая «судьба» такой переменной никак не связана с остальными переменными или какими-либо ограничениями.

В этой связи отметим, что теорему

```
[DEFINE TH3 (CONSEQ (X) (ПТИЦА *X)
   [SEARCH (ПОПУГАЙ *X)])]
```

нельзя рассматривать как описывающую высказывание «любой попугай есть птица». Действительно, если теорема вызывается по образцу (ПТИЦА КЕША), т. е. требуется доказать, что Кеша есть птица, то при сопоставлении этого образца с образцом теоремы переменная X получит значение КЕША. Но в образце функции SEARCH обращение к переменной X задано с префиксом «*», поэтому при сопоставлении этого образца с утверждениями данное значение переменной X игнорируется, и такому образцу соответствует любое утверждение из двух элементов, первый из которых — атом ПОПУГАЙ. Следовательно, теорема не проверяет, является ли Кеша попугаем, а проверяет лишь то, есть ли в базе данных вообще какое-нибудь утверждение о попугаях. Эта ошибка обусловлена тем, что в образце функции SEARCH вместо обращения к переменной X с префиксом «.», который сохранил бы

*) Присваивание [SET X *e*] считается эквивалентным присваиванию [IS *X *e*], поэтому функция SET всегда игнорирует прежнее состояние переменной X. То же справедливо и для всех других функций присваивания.

значение КЕША, указано обращение с префиксом «*», при котором данное значение не принимается во внимание.

Этот пример показывает, что если должно учитываться «прошлое» переменной, то обращаться к ней следует только с префиксом «.». Если же «прошлое» переменной нас не интересует и мы хотим дать ей новое значение, новые связи или ограничения, тогда обращаться к переменной надо с префиксом «*».

Теперь мы сформулируем точные правила сопоставления двух Т-образцов.

1. Если во время сопоставления меняется состояние (значение, связи или ограничение) какой-либо переменной, то обязательно запоминается обратный оператор, назначение которого — восстановить при неудаче прежнее состояние этой переменной. Поскольку в Т-образцах нет сегментных элементов, первая же неудача при сопоставлении образцов ведет к неудачному исходу всего сопоставления и выполнению всех обратных операторов, появившихся с начала сопоставления (т. е. к отмене всех побочных эффектов).

2. Оба сопоставляемых образца (напомним, что это L-списки) должны иметь одинаковую длину, иначе сопоставление неудачно. Эти списки сопоставляются поэлементно: сначала выполняется сопоставление их первых элементов, затем — вторых элементов и т. д. Только если все эти сопоставления удачны, удачным будет и сопоставление образцов в целом.

Далее перечислены все возможные случаи сопоставления элементов Т-образцов.

3. Если один из сопоставляемых элементов не включает в себя переменных либо содержит .-переменные, которые имеют значения, то сопоставление этих элементов сводится к сопоставлению образца с выражением. Например, при значении С у переменной Y сопоставление (A *X) с (A (B .Y)) эквивалентно сопоставлению [IS (A *X) (A (B C))].

4. Сопоставление двух *-переменных всегда удачно. Как побочный эффект обе переменные «теряют» свои прежние состояния и остаются без значений, после чего между этими переменными устанавливается связь.

5. Сопоставление *X с .Y также удачно. Как побочный эффект прежнее состояние переменной X «забывается»; если Y имеет значение, то оно становится и значением переменной X*), иначе между X и Y устанавливается связь, поэтому на переменную X переносятся ограничение и связи переменной Y.

6. При сопоставлении .X с .Y возможны следующие случаи:

— если обе переменные имеют значения, то эти значения сравниваются: они равны — удача, не равны — неудача;

*) Аналогичное правило действует и при сопоставлении образца с выражением.

— если значение имеет только одна переменная, скажем Y, то проверяется, удовлетворяет ли это значение ограничению на будущее значение переменной X. Если не удовлетворяет, то сопоставление .X с .Y неудачно, иначе, как и в случае отсутствия ограничений на X, сопоставление удачно, и переменная X и все связанные с нею переменные получают значение переменной Y *);

— если переменные X и Y не имеют значений, то проверяется, не противоречат ли друг другу ограничения на эти переменные (если, конечно, таковые есть). Если противоречат, то сопоставление неудачно. В противном случае сопоставление удачно, и при этом между переменными X и Y устанавливается связь, а «пересечение» их прежних ограничений **) становится новым ограничением на каждую из этих переменных и, следовательно, на все связанные с ними переменные.

7. Сопоставление *X со списком (pat_1 pat_2 ... pat_k) удачно, если переменная X не входит в этот список ***). Как побочный эффект прежнее состояние переменной X «забывается», и она остается без значения, но на ее будущее значение накладывается следующее ограничение: оно должно быть L-списком из k элементов, на каждый из которых накладывается ограничение так, как если бы этот элемент, рассматриваемый как *-переменная, сопоставлялся с соответствующим образцом pat_i.

Например, если переменная Z имеет значение B, то после сопоставления *X с (A .Z *Y .Y) на переменную X накладывается такое ограничение: ее будущим значением должен быть L-список из четырех элементов, первый из которых — атом A, второй — атом B, третий — произвольный, но совпадающий с будущим значением переменной Y (она «теряет» свое прежнее значение, так как входит в образец с префиксом «*»), а четвертый элемент должен совпадать с третьим элементом (связь между этими элементами устанавливается через переменную Y, которая второй раз входит с префиксом «.», сохраняющим ее связь с третьим элементом).

8. Сопоставление .X со списком (pat_1 pat_2 ... pat_k) в том случае, когда переменная X не имеет ни значения, ни ограничения, ни связей, аналогично предыдущему сопоставлению. Если же у переменной X есть значение, то мы имеем сопоставление выражения (значения X) с образцом. В остальных случаях (при ус-

---

*) Аналогичное правило действует и при сопоставлении образца с выражением.

**) Взятие «пересечения» можно рассматривать как выполнение сопоставления образцов, по которым были установлены ограничения на переменные.

***) Например, сопоставление *X с (A (B *X)) неудачно.

ловии, что X не входит в список) делается попытка наложить на X новое ограничение, которое образуется из сопоставляемого с переменной X образца-списка так, как описано в п. 7. Если это не удается сделать, сопоставление считается неудачным. В противном случае берется «пересечение» этого и прежнего ограничений и оно становится новым ограничением на будущие значения переменной X и всех связанных с нею переменных.

5.5. Дополнительные возможности

Функция GOAL. Как уже было сказано, для достижения целей используются целевые теоремы. Однако нередко цель может быть достигнута более простым способом — просмотром базы данных и нахождением утверждения, соответствующего образцу-цели. Поэтому при достижении цели сначала, как правило, осуществляется поиск в базе данных и лишь затем, если этот поиск неуспешен, вызываются теоремы. Именно так действует встроенная функция GOAL:

[GOAL *pat test*? *rec*?], FSUBR.

Здесь *pat* должен быть T-образцом или .-переменной, значением которой является T-образец. Аргумент *test* — такой же, как и у функции SEARCH (см. § 4.4), а аргумент *rec* — это рекомендация. Действие функции GOAL эквивалентно вычислению

[ALT [SEARCH *pat tets*?] [ACHIEVE *pat rec*?]]

(если дана USE1-рекомендация, то вместо функции SEARCH подразумевается функция SEARCH1).

Предположим, к примеру, что в базе данных записаны следующие утверждения:

(РЕБЕНОК МАША)
(ЧЕЛОВЕК ПЕТР)

и что имеется теорема, соответствующая высказыванию «любой ребенок является человеком»:

(CONSEQ (X) (ЧЕЛОВЕК *X)
[SEARCH (РЕБЕНОК .X)])

Тогда, как видно, истинность факта «некто есть человек» можно доказать двумя способами: либо найти в базе данных такое утверждение, либо воспользоваться этой теоремой. И если заранее неизвестно, какой способ подойдет, следует применить функцию GOAL. Например, вычисление [GOAL (ЧЕЛОВЕК ПЕТР)] успешно, так как в базе данных есть утверждение (ЧЕЛОВЕК

ПЕТР). Успешно и вычисление [GOAL (ЧЕЛОВЕК МАША)]: здесь также сначала просматривается база данных, однако утверждения (ЧЕЛОВЕК МАША) там нет, поэтому вызывается теорема, которая и докажет, что Маша является человеком.

Функция UNIQUE. При вызове теорем по образцу можно попасть в бесконечную рекурсию. Например, по некоторому образцу P может быть вызвана теорема TH, которая непосредственно сама или через вызванные ею теоремы осуществляет вызов теорем по образцу P, и при этом снова вызывается теорема TH.

Для борьбы с такими зацикливаниями в язык встроена функция UNIQUE, обращение к которой имеет следующий вид:

[UNIQUE]

Эта функция вырабатывает неуспех (с сообщением UNIQUE), если последний из активных вызывающих образцов не является «уникальным». Если же этот образец «уникален», то функция успешно заканчивает свою работу со значением T.

Вызывающий образец является активным, если по нему вызвана теорема, вычисление которой в настоящее время еще не окончено. Образец не является «уникальным», если среди других активных образцов есть такой, что оба этих образца используются для вызова теорем одного и того же типа (целевых, записывающих или вычеркивающих) и оба образца равны как списки при условии, что в них вместо всех .-переменных, имеющих значения, и :-переменных подставлены их значения (*-переменные, а также .-переменные, не имеющие значений, не заменяются). Например, если переменная X имеет значение A, а переменная Y не имеет значения, то образцы (ON .X .Y) и (ON A .Y) совпадут при указанном условии, а образцы (ON .X .Y) и (ON B .Y) или образцы (ON *X .Y) и (ON *Z .Y) не совпадут.

Рассмотрим следующий пример. Предположим, что имеются точки A, B и C, причем точка A соединена с точкой B, а точка B соединена с точкой C, и требуется определить, можно ли попасть из A в C.

Если утверждение вида (PATH x y) означает, что точка x соединена с точкой y, а утверждение вида (REACHABLE x) говорит, что в точку x можно попасть из точки A, тогда исходную ситуацию нашей задачи можно описать в виде следующей базы данных:

```
(PATH A B)   (PATH B A)
(PATH B C)   (PATH C B)
(REACHABLE A)
```

Высказывание «из A можно попасть в x» доказывается так: надо сначала найти некоторую точку y, соединенную с x, а за-

тем доказать «из A можно попасть в *y*». Этому соответствует такая теорема:

```
[DEFINE TH (CONSEQ (X Y) (REACHABLE *X)
   [UNIQUE]
   [SEARCH (PATH .X *Y)]
   [GOAL (REACHABLE .Y)])]
```

Цель нашей задачи задается в виде выражения

```
[GOAL (REACHABLE C)]
```

Рассмотрим протокол вычисления этого выражения:

```
цель 1: (REACHABLE C)
    теорема TH: X: = C
        [UNIQUE] — успех
        [SEARCH (PATH C *Y)] — успех, Y: = B
        цель 2: (REACHABLE .Y)
            теорема TH: X: = B
                [UNIQUE] — успех
(*)             [SEARCH (PATH B *Y)] — успех, Y: = C
                цель 3: (REACHABLE .Y)
                    теорема TH: X: = C
                        [UNIQUE] — неуспех
```

Остановимся в этот момент вычислений и отметим следующее. Первое обращение к функции UNIQUE успешно потому, что в этот момент имеется лишь один активный вызывающий образец. Во второй раз функция также успешна, поскольку последний вызывающий образец (REACHABLE .Y) при значении B у переменной Y не совпадает с первым образцом-целью (REACHABLE C). Но в третий раз функция UNIQUE вырабатывает неуспех, так как третья цель — (REACHABLE .Y) — при значении C переменной Y совпадает с первой целью. Этот неуспех — сигнал о том, что такая цель уже ставилась. Неуспех возвращает программу к функции SEARCH, отмеченной (*), которая найдет теперь для переменной Y другое значение — A, и дальнейшее вычисление будет успешным:

```
(*)             [SEARCH (PATH B *Y)] — успех, Y: = A
                [GOAL (REACHABLE A)] — успех
            выход из TH
    выход из TH
успех
```

В этом примере функция UNIQUE спасла нас от бесконечных перемещений между точками B и C.

Следует, однако, отметить, что функция UNIQUE может выработать неуспех и тогда, когда зацикливания в действительности нет. Например, последняя цель может совпадать с одной из предыдущих, но поставлены эти цели могут быть в разных условиях —

при разных базах данных. Поэтому совпадение целей еще не означает зацикливания.

Функции CANDIDATES и APPLY. Функция CANDIDATES уже рассматривалась (см. § 4.4): она применяется для поиска утверждений-кандидатов. Эту же функцию можно использовать и для нахождения теорем-кандидатов. В таком случае при обращении к ней:

[CANDIDATES *pat type*], FSUBR

в качестве аргумента *pat* должен быть задан Т-образец или .-переменная, значением которой является Т-образец, а значением второго аргумента обязан быть один из следующих атомов: CONSEQ, ANTEC или ERASING. Этот атом указывает, теоремы какого типа (целевые, записывающие или вычеркивающие) следует просматривать данной функции. Как уже было сказано в § 5.3, теорема является кандидатом на вызов по образцу *pat*, если ее образец соответствует образцу *pat* в предположении, что переменным обоих образцов соответствуют любые элементы в другом образце и что списковым элементам (с верхнего уровня) соответствуют любые списки той же длины.

Значением функции CANDIDATES является L-список, составленный из имен теорем-кандидатов требуемого типа и списков свойств этих теорем. Значение функции может быть, например, таким:

```
(TH2 (CLASS 5) TH7 ( )
     TH1 (DIFFICULTY EASY CLASS 2))
```

С помощью следующей встроенной функции можно вызывать теоремы по имени:

[APPLY *pat th-name*], FSUBR

Первый аргумент этой функции должен быть Т-образцом или .-переменной, значением которой является Т-образец, а значением второго аргумента обязан быть идентификатор — имя одной из теорем. Функция осуществляет вызов этой теоремы по образцу *pat*. Если вызов теоремы неуспешен или если указанной теоремы вообще нет, тогда вычисление функции APPLY неуспешно (она не ставит F-точку). Если же образец теоремы соответствует образцу *pat* и ее тело успешно, то вычисление функции успешно и ее значением объявляется значение, выработанное теоремой.

Функции CANDIDATES и APPLY предоставляют пользователю возможность по-своему организовать вызов теорем, если его не удовлетворяет тот, что встроен в язык.

Редактирование рекомендаций. Рекомендации, которые пользователь дает функциям, вызывающим теоремы по образцу, могут

быть определены как заранее (при написании программы), так и непосредственно перед вызовом теорем (с помощью FILTER-рекомендации). Но после того как рекомендации определены, они не меняются. В то же время в некоторых случаях возникает потребность в изменении рекомендаций, после того как вызов теорем уже начался. Например, может потребоваться изменить тип рекомендации (скажем, с USE на USE1) или порядок просмотра теорем, запретить вызов ранее рекомендованной теоремы или, наоборот, рекомендовать новую теорему. Так, теорема, описывающая действие «поднять предмет правой рукой» и не сумевшая достичь поставленной цели, может исключить из числа рекомендованных теорему, описывающую действие «поднять предмет левой рукой».

Для редактирования уже действующих рекомендаций в язык встроены следующие функции: GETPOSS, которая позволяет узнать, какие теоремы еще не вызывались, но могут быть вызваны; GETREC, с помощью которой можно определить, какие элементы рекомендации предстоит еще рассмотреть; PUTREC, которая дает возможность заменить прежнюю рекомендацию на новую. Все эти функции работают с рекомендацией ближайшей объемлющей функции, вызывающей теоремы по образцу.

Функция GETPOSS: [GETPOSS].

Данная фукция находит ближайшую объемлющую функцию, которая осуществляет вызов теорем, и в качестве своего значения выдает ее текущий список возможностей, т. е. список имен теорем-кандидатов, которые пока еще не вызывались (в этом списке нет имени теоремы, вычисляющейся в данный момент). Если такой ближайшей функцией является функция APPLY, то выдается пустой список ( ). Если среди объемлющих функций нет ни одной, что вызывает теоремы, то вырабатывается неуспех с сообщением GETPOSS.

Функция GETREC: [GETREC].

Эта функция также находит ближайшую объемлющую функцию, вызывающую теоремы, и выдает в качестве своего значения ее текущую рекомендацию (в ней нет тех элементов исходной рекомендации, которые уже были рассмотрены, но есть элемент, рассматриваемый в настоящий момент). Если такой ближайшей функцией является функция APPLY, то GETREC в качестве своего значения выдает идентификатор — имя вызванной теоремы. Если среди объемлющих функций нет таких, которые вызывают теоремы, то вырабатывается неуспех с сообщением GETREC.

Функция PUTREC: [PUTREC *l*], SUBR.

Значением L аргумента этой функции должен быть список, представляющий собой рекомендацию, начинающуюся с USE, USE1 или TRY. Функция PUTREC находит ближайшую объемлющую функцию, вызывающую теоремы, и заменяет ее текущую рекомендацию на новую — на L. Как только закончится вычисление теоремы, в которой произошло обращение к PUTREC, так сразу же вступит в силу эта новая рекомендация. Обратный оператор функции PUTREC не запоминает. Ее значение — список L.

Если ближайшей функцией, вызывающей теоремы, является функция APPLY или если среди объемлющих функций нет таких, которые вызывают теоремы, тогда функция PUTREC вырабатывает неуспех, связывая с ним сообщение PUTREC.

Операции над списками свойств теорем. В заключение рассмотрим встроенные функции, с помощью которых можно анализировать и изменять списки свойств теорем.

Функция GETH: [GETH *th-name ind*?], SUBR.

Эта функция действует аналогично функции GET (см. § 1.17), но по отношению к списку свойств теоремы, имя которой задается аргументом *th-name*. Таким образом, функция GETH выдает в качестве своего значения весь список свойств этой теоремы, если аргумент *ind* не задан, либо значение только одного свойства — того, название которого определяет аргумент *ind*. Если нужной теоремы нет, вырабатывается неуспех с сообщением GETH.

Функция PUTH: [PUTH *th-name ind v*], SUBR.

Данная функция, аналогичная функции PUT (см. § 1.17), заменяет в списке свойств теоремы, имя которой указано аргументом *th-name*, прежнее значение свойства с названием *IND* на новое значение *V*. Одновременно запоминается обратный оператор, назначение которого — восстановить при неуспехе прежнее значение этого свойства. Если указанной теоремы нет, вырабатывается неуспех с сообщением PUTH.

Функция PPUTH: [PPUTH *th-name ind v*], SUBR.

Эта функция действует аналогично функции PUTH, но обратный оператор не запоминает.

5.6. Примеры использования целевых теорем

Плэнер, как и любой другой язык программирования, предназначен для записи разработанных человеком алгоритмов решения задач. В то же время на плэнере можно программировать

путем описания того, что имеется и что надо получить, без явного указания того, как это можно получить. Ответственность за поиск решения описанной задачи возлагается при этом на дедуктивный механизм языка. В предыдущих параграфах уже встречались отдельные примеры работы этого механизма и использования целевых теорем при описании задач. В данном параграфе эти же аспекты будут рассмотрены более детально.

Как уже было сказано, основное назначение целевых теорем — описывать элементарные средства достижения целей, из которых (средств) и должно быть составлено решение задачи. Образец каждой такой теоремы обычно описывает ту цель, которой можно достичь с помощью теоремы, а ее тело — действия, которые следует предпринять, для того чтобы можно было считать эту цель достигнутой. Действия могут быть самыми разнообразными, но обычно они следующие: либо показывается, что цель является следствием из фактов, описанных в текущей базе данных, либо осуществляется (допустимым образом) такое изменение базы данных, чтобы в новой базе данных цель оказалась достигнутой. В первом случае теоремы отвечают на вопросы типа «верно ли, что данный предмет находится в данном месте», а во втором случае теоремы удовлетворяют требования типа «сделать так, чтобы данный предмет оказался в данном месте». Первые теоремы применяются для описания логических закономерностей (правил вывода, свойств используемых понятий, связей между ними и т. п.) той задачи, решение которой должна найти плэнерская программа, а вторые теоремы используются для описания операций и правил преобразования (например, элементарных действий робота), которые разрешено применять в этой задаче.

Рассмотрим примеры определения и использования целевых теорем обоих указанных типов.

Логические правила обычно записываются в виде «если A, то B». Это можно интерпретировать следующим образом: если надо показать истинность B, то следует показать истинность A. Подобная императивная трактовка правил и лежит в основе описания их в виде плэнерских теорем: следствие B выносится в образец теоремы, а предпосылка (или предпосылки) A оформляются как тело теоремы.

Например, свойство транзитивности отношения «меньше» в логической нотации записывается так:

$$less(x, y) \wedge less(y, z) \supset less(x, z).$$

Нужная для нас интерпретация: если надо доказать, что x меньше z, то для этого следует найти какой-нибудь y, о котором известно, что x меньше его, и затем следует доказать, что y меньше z.

Этой интерпретации соответствует следующая целевая теорема:

```
[DEFINE TRANS (CONSEQ (X Y Z) (LESS *X *Z)
    [SEARCH (LESS .X *Y)]
    [GOAL (LESS .Y .Z)])]
```

Эту теорему можно использовать для доказательства фактов об отношении «меньше», которые явно не указаны в базе данных, но которые логически следуют из записанных там фактов.

Пусть к примеру, база данных была заполнена в результате вычисления

```
[DATA (LESS A B) (LESS A 20) (LESS B C)
    (LESS C 53)]
```

Если мы хотим проверить, «верно ли, что A меньше 53», тогда достаточно вычислить выражение

```
[GOAL (LESS A 53)]
```

Ответ на поставленный вопрос будет найден автоматически («да», если вычисление этого выражения успешно, и «нет», если оно неуспешно). В нашем случае поиск ответа осуществляется следующим образом (напомним, что функция GOAL — это комбинация функций SEARCH и ACHIEVE):

```
[GOAL (LESS A 53)]
    [SEARCH (LESS A 53)] — неуспех
    [ACHIEVE (LESS A 53)]
    теорема TRANS: X: = A, Z: = 53
        [SEARCH (LESS A *Y)] — успех, Y: = B
        [GOAL (LESS B 53)]
            [SEARCH (LESS B 53)] — неуспех
            [ACHIEVE (LESS B 53)]
            теорема TRANS: X: = B, Z: = 53
                [SEARCH (LESS B *Y)] — успех,
                    Y: = C
                [GOAL (LESS C 53)]
                    [SEARCH (LESS C 53)] — успех
            выход из TRANS
    выход из TRANS
успех
```

Здесь для доказательства «A меньше 53» было найдено B, которое больше A, а затем была поставлена цель доказать, что B меньше 53. Для этого же было найдено C, которое больше B, а затем была поставлена цель доказать, что «C меньше 53». Последнее высказывание доказывается непосредственно — нахож-

дением соответствующего утверждения в базе данных. Таким образом, была найдена цепочка A < B < C < 53, которая и доказывает исходное высказывание.

Теорему TRANS можно использовать также и для поиска объектов, связанных отношением «меньше», причем таких объектов, о которых в базе данных не сказано явно, что один меньше другого. Например, при вычислении

```
[ACHIEVE (LESS *P 53)]
```

эта теорема может присвоить переменной P значение B, т. е. на запрос «найти объект, меньший 53» она даст ответ B:

```
[ACHIEVE (LESS *P 53)]
теорема TRANS: X связывается с P, Z: = 53
(*) [SEARCH (LESS .X *Y)] — успех,
        X: = C (P: = C), Y: = 53
    [GOAL (LESS 53 53)]
        [SEARCH (LESS 53 53)] — неуспех
        [ACHIEVE (LESS 53 53)]
        теорема TRANS: X: = 53, Z: = 53
            [SEARCH (LESS 53 *Y)] — неуспех
        возврат по неуспеху к (*)
    [SEARCH (LESS .X *Y)] — успех,
        X: = B (P: = B), Y: = C
    [GOAL (LESS C 53)]
        [SEARCH (LESS C 53)] — успех
выход из TRANS
успех
```

(Здесь между пунктирными линиями показана неуспешная ветвь вычисления.)

Теорема TRANS — пример описания «чисто» логического высказывания. Но в плэнерских теоремах можно применять и нелогические действия и проверки, можно использовать не только приказы типа «найти что-то» или «доказать что-то», но и любые другие процедуры языка. Например, для доказательства «x меньше z» можно воспользоваться и арифметическими функциями, если x и z — числа:

```
[DEFINE LESS (CONSEQ (X Z) (LESS *X *Z)
    [UNFALSE [AND [NUM .X] [NUM .Z]
        [LT .X .Z]]])]
```

Каждая из теорем TRANS и LESS предлагает свой способ достижения одной и той же цели и применима при своих условиях. Так, теорема LESS не сможет доказать (20 меньше C», но

легко доказывает «20 меньше 53», тогда как теорема TRANS не сумеет доказать последнее высказывание при имеющейся базе данных.

Следует отметить, что эти теоремы не вполне равноценны. Если с помощью теоремы TRANS можно находить объекты, связанные отношением «меньше», то теорема LESS для этого не приспособлена. Она определена так, что при вызове, скажем, по образцу (LESS *P 53) она даст ошибку, поскольку функция NUM невыполнима при неопределенном значении своего аргумента. И если возможны вызовы теорем по таким образцам, то следует либо изменить определение теоремы LESS, либо запретить с помощью рекомендаций ее вызов, например, так:

[ACHIEVE (LESS *P 53) (USE [NON LESS])]

Теперь рассмотрим целевые теоремы, которые для достижения цели изменяют базу данных. Такие теоремы, как уже отмечалось, используются для описания тех операций, которые можно применять в задаче, решаемой плэнерской программой. Эти операции обычно представляются следующим образом: если удовлетворены предпосылки A, тогда можно выполнить действие D, получая в результате R. Как и раньше, нас интересует императивная трактовка: если требуется получить R, тогда надо сделать так, чтобы были удовлетворены предпосылки A, а затем уже нужно выполнить действие D. Именно эта трактовка и лежит в основе представления операций в виде плэнерских теорем: результат R выносится в образец теоремы, а приказы «удовлетворить предпосылки A» и «выполнить действие D» оформляются как тело теоремы.

Но что здесь понимается под словами «выполнить действие»? Действие — это некоторое изменение реальной обстановки. Поскольку в плэнерских программах обстановка описывается в виде базы данных, то изменение обстановки имитируется изменением базы данных. Тем самым под «выполнением действия» подразумевается соответствующее изменение базы данных, а именно: вычеркивание утверждений, описывающих факты, которые после выполнения действия стали ложными, и запись новых утверждений, которые описывают факты, ставшие истинными.

Рассмотрим, например, действия робота «перенести некоторый предмет из одного места в другое» и «перейти из одного места в другое».

Будем предполагать, что робот может переносить только ящики и что он может беспрепятственно переходить и переносить любые ящики из любой точки в любую другую. Будем также считать, что в базе данных имеются следующие утверждения:

(ATR x) — робот находится на полу в точке x;
(AT th x) — предмет th находится на полу в точке x;
(BOX b) — b является ящиком.

При этих предположениях «перенос предмета TH из точки X в точку Y» можно описать в виде такой целевой теоремы:

```
[DEFINE НЕСТИ (CONSEQ (X Y TH) (AT *TH *Y)
    [SEARCH (BOX .TH)]
    [SEARCH1 (AT .TH *X)]
    [GOAL (ATR .X) (USE1)]
    [ERASE (AT .TH .X)] [ERASE (ATR .X)]
    [ASSERT (AT .TH .Y)] [ASSERT (ATR .Y)]
    [PLANNING (ПЕРЕНЕСТИ .TH ИЗ .X В .Y)])]
```

Образец этой теоремы описывает основной результат действия «перенести» — предмет TH окажется в точке Y (есть и побочный результат: в этой же точке окажется и робот).

Данное действие имеет две предпосылки. Первую из них — TH должен быть ящиком — проверяет функция SEARCH, с обращения к которой начинается тело теоремы. Вторая предпосылка — робот должен оказаться в той же точке, что и ящик (только тогда робот может начать перенос). Теорема определяет, в какой точке X сейчас находится ящик, и ставит цель сделать так, чтобы стало истинным утверждение о том, что робот находится в точке X*). Достичь этой цели можно с помощью какой-либо теоремы, описывающей передвижение робота. Какая именно из них будет использована, мы не знаем, да это и не надо нам знать. Однако нам известно заранее, что достаточно найти лишь один способ подхода работа к ящику и что в случае последующего неуспеха функция GOAL не должна возобновлять свою работу. Именно это мы и указали с помощью USE1-рекомендации.

Если удалось удовлетворить обе предпосылки, то далее теорема «выполняет» само действие переноса ящика: она вычеркивает из базы данных утверждения о том, что ящик и робот находятся в точке X, и записывает утверждения о том, что они теперь расположены в точке Y.

В конце теоремы записано обращение к функции PLANNING, которая будет рассмотрена чуть позже. Пока же можно считать, что она действует как пустой оператор.

---

*) Отметим, что в теоремах рассматриваемого типа следует не столько проверять, выполнены ли предпосылки, сколько требовать, чтобы они стали выполненными. Если, например, в теореме НЕСТИ заменить обращение к функции GOAL на обращение к функции SEARCH, т. е. на проверку, находится ли робот в точке X, и если окажется, что робота нет в этой точке, то теорема никогда не сможет достичь своей цели.

Действие «перемещение робота из точки X в точку Y» описывается следующей теоремой:

```
[DEFINE ИДТИ (CONSEQ (X Y) (ATR *Y)
    [GOAL (ATR *X) (USE1 [NON ИДТИ])]
    [ERASE (ATR .X)]
    [ASSERT (ATR .Y)]
    [PLANNING (ИДТИ ИЗ .X В .Y)])]
```

Образец теоремы указывает, что в результате выполнения теоремы робот окажется в точке Y. Предпосылкой описываемого действия является то, что робот находится в какой-либо точке X на полу. Если он не на полу (а, например, на ящике), тогда функция GOAL требует сделать так, чтобы он оказался на полу; при этом может быть применена любая подходящая теорема, но не сама теорема ИДТИ. Собственно же действие перехода имитируется вычеркиванием утверждения о том, что робот в X, и записью утверждения о том, что робот в Y.

Рассмотрим, как будут использоваться обе эти теоремы в задаче, в которой требуется поместить ящики BOX1 и BOX2 в какую-нибудь одну точку, если в исходной ситуации робот находился на полу в точке P, ящик BOX1 — в точке Q, а ящик BOX2 — в точке R.

База данных, описывающая исходную ситуацию, задается так:

```
[DATA (ATR P) (BOX BOX1) (AT BOX1 Q)
      (BOX BOX2) (AT BOX2 R)]
```

а цель задачи задается выражением

```
[PROG (H) [GOAL (AT BOX1 *H)]
          [GOAL (AT BOX2 .H)]]
```

Вычисление этого выражения осуществляется следующим образом:

```
[GOAL (AT BOX1 *H)]
    [SEARCH (AT BOX1 *H)] — успех, H:= Q
успех
[GOAL (AT BOX2 Q)]
    [SEARCH (AT BOX2 Q)] — неуспех
    [ACHIEVE (AT BOX2 Q)]
    теорема НЕСТИ: = TH: = BOX2, Y: = Q
        [SEARCH (BOX BOX2)] — успех
        [SEARCH1 (AT BOX2 *X)] — успех, X: = R
        [GOAL (ATR R)]
            [SEARCH (ATR R)] — неуспех
            [ACHIEVE (ATR R)]
            теорема ИДТИ: Y:= R
```

[GOAL (ATR *X)]
[SEARCH (ATR *X)] — успех, X: = P
[ERASE (ATR P)]
[ASSERT (ATR R)] } «перейти из P в R»
выход из ИДТИ
[ERASE (AT BOX2 R)]
[ERASE (ATR R)]
[ASSERT (AT BOX2 Q)]
[ASSERT (ATR Q)] } «перенести BOX2 из R в Q»
выход из НЕСТИ
успех

Таким образом, была выбрана точка Q, где уже находился ящик BOX1, после чего робот подошел к ящику BOX2 и перенес его в точку Q. В этот момент база данных стала такой:

(ATR Q) (BOX BOX2) (AT BOX2 Q)
(BOX BOX1) (AT BOX1 Q),

т. е. она описывает ситуацию, в которой цель задачи достигнута.

В задачах о роботах часто требуется не выполнение, а планирование действий робота: необходимо выдать список действий, выполнение которых приведет к поставленной цели, но сами эти действия не должны быть выполнены, и потому должно быть сохранено исходное состояние базы данных. Добиться этого можно с помощью тех же самых теорем, что используются для выполнения действий, но при этом необходимы некоторые изменения.

Во-первых, для того чтобы запоминать, какие теоремы, при каких параметрах и в каком порядке вызывались, можно ввести константу, скажем, PLAN с начальным значением ( ), в которую каждая проработавшая теорема будет заносить информацию о себе. Именно такое изменение значения константы и осуществляет функция PLANNING:

```
[DEFINE PLANNING (LAMBDA (INF)
   [CSET PLAN (!:PLAN .INF)])]
```

В конце концов значением этой константы окажется перечень действий робота, которые были выполнены ради достижения поставленной цели. В нашем примере с переносом ящиков значением константы PLAN будет следующий список:

((ИДТИ ИЗ P В R) (ПЕРЕНЕСТИ BOX2 ИЗ R В Q))

Во-вторых, после того как цель достигнута и план действий робота найден, необходимо уничтожить все произведенные изменения в базе данных. Для этого можно воспользоваться функцией TEMP или TPROG (см. § 3.6), которая перед завершением своей

работы отменяет все побочные эффекты «внутри» себя. Таким образом, если бы в нашей задаче требовалось спланировать, а не выполнить действия, то следовало бы несколько по-иному задать цель задачи:

```
[TPROG (H) [CSET PLAN ( )]
    [GOAL (AT BOX1 *H)] [GOAL (AT BOX2 .H)]
    :PLAN]
```

Тогда значением этого выражения будет указанный выше список-план, а исходное состояние базы данных не изменится.

Опишем еще несколько действий робота и решим известную задачу «обезьяна и бананы», правда, в несколько измененной формулировке, взятой из [17].

В этой задаче предполагается, что в некоторой комнате, где действует робот, на полу расположены высокие и низкие ящики, а на стене есть выключатель, который расположен на недосягаемой для робота высоте. Предполагается, что робот может повернуть выключатель (зажечь свет в комнате), если он будет находиться на высоком ящике, помещенном под выключателем. Требуется составить план действий робота, в результате выполнения которых он сможет зажечь свет.

В дополнение к рассмотренным выше утверждениям базы данных будем также использовать следующие утверждения:

(ONR b) — робот находится на ящике b (в каждый момент в базе данных может быть либо это утверждение, либо утверждение (ATR x), означающее, что робот на полу);

(AT LIGHTSWITCH x) — выключатель расположен над точкой x;

(LIGHTSWITCH s) — если $s =$ ON, то свет горит, а если $s =$ = OFF, то не горит.

Кроме того, будем считать, что с каждым утверждением (BOX b) связано свойство HEIGHT, значение которого указывает высоту ящика b в каких-то единицах длины. Ящик считается высоким, если его высота не менее 5 единиц.

При этих соглашениях действие робота «залезть на предмет TH» можно описать в виде такой теоремы:

```
[DEFINE ЗАЛЕЗТЬ (CONSEQ (TH X) (ONR *TH)
    [SEARCH (BOX .TH)]
    [SEARCH1 (AT .TH *X)]
    [GOAL (ATR .X) (USE1)]
    [ERASE (ATR .X)]
    [ASSERT (ONR .TH)]
    [PLANNING (ЗАЛЕЗТЬ НА .TH)])]
```

Образец этой теоремы описывает результат «робот находится на ТН», который можно получить при ее выполнении. В теле теоремы сначала проверяется, является ли ТН ящиком (высота ящика не проверяется — предполагается, что робот может залезть на любой ящик), а затем ставится цель: сделать так, чтобы робот оказался на полу в той же точке X, что и ящик. Если эти предпосылки удовлетворены, тогда «выполняется» само действие залезания на ящик: вычеркивается, что робот на полу в точке X, и записывается, что он на ящике ТН.

Действие «слезть с ящика» может быть выполнено только при условии, что робот действительно находится на ящике. Результатом же этого действия будет то, что робот окажется на полу в той же точке, где расположен ящик:

```
[DEFINE СЛЕЗТЬ (CONSEQ (TH X) (ATR *X)
    [SEARCH1 (ONR *TH)]
    [SEARCH1 (AT .TH .X)]
    [ERASE (ONR .TH)]
    [ASSERT (ATR .X)]
    [PLANNING (СЛЕЗТЬ С .TH)])]
```

Действие «зажечь свет» описывается следующей теоремой:

```
[DEFINE ЗАЖЕЧЬ (CONSEQ (X TH)
          (LIGHTSWITCH ON)
    [SEARCH1 (AT LIGHTSWITCH *X)]
    [SEARCH (BOX *TH) (TEST HEIGHT [GE 5])]
    [GOAL (AT .TH .X)]
    [GOAL (ONR .TH)]
    [ERASE (LIGHTSWITCH OFF)]
    [ASSERT (LIGHTSWITCH ON)]
    [PLANNING (ЗАЖЕЧЬ СВЕТ)])]
```

Данная теорема сначала определяет, над какой точкой X расположен выключатель, затем подыскивает ящик, высота которого не меньше 5 единиц, и ставит две подцели: этот ящик должен сказаться в точке X и робот должен находиться на ящике. Собственно действие сводится к вычеркиванию утверждения, что свет не горит, и к записи утверждения, что он зажжен.

Можно было бы описать и другие действия робота («ставить ящик на ящик», «выключить свет» и т. д.), но мы ограничимся только рассмотренными выше и опишем теперь исходную ситуацию и цель нашей задачи.

Исходную ситуацию задачи будем предполагать следующей: робот находится на ящике BOX1, ящик BOX1 высотой 2 расположен в точке P1, ящик BOX2 высотой 6 — в точке P2, выключа-

тель не повернут и находится над точкой P3. База данных, описывающая эту ситуацию, задается так:

```
[DB (BOX BOX1) (HEIGHT 2)
    (BOX BOX2) (HEIGHT 6)]
[DATA (ONR BOX1) (AT BOX1 P1) (AT BOX2 P2)
    (AT LIGHTSWITCH P3) (LIGHTSWITCH OFF)]
```

Цель задачи — составить план действий, в результате исполнения которого в комнате будет гореть свет,— описывается выражением

```
[TPROG ( ) [CSET PLAN ( )]
           [GOAL (LIGHTSWITCH ON)] :PLAN]
```

При вычислении этого выражения будет составлен следующий список-план:

```
((СЛЕЗТЬ С BOX1)
(ИДТИ ИЗ P1 В P2)
(ПЕРЕНЕСТИ BOX2 ИЗ P2 В P3)
(ЗАЛЕЗТЬ НА BOX2)
(ЗАЖЕЧЬ СВЕТ))
```

5.7. Использование записывающих и вычеркивающих теорем

Записывающие теоремы можно вызвать, обратившись к функции DRAW, но обычно они вызываются при вычислении функции ASSERT или PASSERT (см. § 4.2). Если в обращении к такой функции указана рекомендация, то после того как функция запишет утверждение в базу данных, начинается вызов записывающих теорем согласно этой рекомендации. Образцом для вызова здесь служит только что записанное утверждение *).

Например, при вычислении выражения

```
[ASSERT (AT BOX PLACE) (TRY)]
```

в базу данных будет записано, если, конечно, его там не было, утверждение (AT BOX PLACE), после чего будет выполнено

```
[DRAW (AT BOX PLACE) (TRY)]
```

т. е. будут по очереди вызваны все записывающие теоремы, образцы которых соответствуют списку (AT BOX PLACE). Если

*) При сопоставлении с образцами теорем утверждение трактуется как выражение, а не как образец. Поэтому в утверждении (A [B]) элемент [B] рассматривается просто как список, а не как обращение к процедуре B.

все эти теоремы успешны, тогда вычисление обращения к функции ASSERT успешно завершается со значением T. Но если хотя бы одна из этих теорем окажется неуспешной, то возникший неуспех не будет остановлен функцией ASSERT, и он, в частности, отменит запись утверждения (AT BOX PLACE) в базу данных.

Аналогична ситуация и с вычеркивающими теоремами: они будут вызываться при вычислении функции ERASE (или PERASE), если в обращении к ней указана какая-нибудь рекомендация. Образцом для вызова служит только что вычеркнутое утверждение.

Записывающие и вычеркивающие теоремы введены в язык, для того чтобы можно было контролировать изменения базы данных. С помощью этих теорем обычно проверяется, допустимо ли произведенное изменение, или выполняются дополнительные изменения базы данных.

Например, если в решаемой задаче не допускается, чтобы два предмета находились в одной и той же точке, то это можно проконтролировать с помощью такой записывающей теоремы:

```
[DEFINE A1 (ANTEC (X Y) (AT *X *Y)
    [IF ([SEARCH (AT [NON .X] .Y)] [FAIL])])]
```

Если выполняется [ASSERT (AT TH P1) (TRY)] и вызвана эта теорема, то она проверит, не было ли в базе данных утверждения о том, что в точке P1 находится предмет, отличный от TH. Если было, то теорема выработает неуспех, который и отменит как незаконную запись утверждения (AT TH P1). В противном случае теорема успешна, и запись не будет отменена.

В то же время записывающая теорема

```
[DEFINE A2 (ANTEC (X Y) (AT *X *Y)
    [ERASE (CLEAR .Y)])]
```

предназначена для устранения конфликтов в базе данных. Она требует вычеркнуть утверждение о том, что точка Y свободна, если в базу данных было записано утверждение о том, что некоторый предмет помещен в эту точку.

Еще один пример:

```
[DEFINE LOVE (ANTEC (X Y) (LOVES *X *Y)
    [ASSERT (HUMAN .X)]
    [ASSERT (HUMAN .Y)])]
```

Эта теорема осуществляет сопутствующие изменения базы данных: если записывается, что X любит Y, то как следствие записывается, что X и Y являются людьми.

Аналогично применяются и вычеркивающие теоремы.

В заключение рассмотрим использование теорем разного типа на примере системы, отвечающей на вопросы о родственных отношениях.

Предположим, что пользователь ведет диалог с этой системой, сообщая ей факты вида

(x — МУЖ y), (x — ЖЕНА y),
(x — ОТЕЦ y), (x — МАТЬ y),
(x — СЫН y), (x — ДОЧЬ y)

и задавая вопросы вида

(В КАКОМ РОДСТВЕ x И z ?),
(КТО p y ?).

(Здесь x, y и z — имена людей, причем x и z записываются в именительном падеже, а y — в родительном; p — любое из названий родства: МУЖ, ЖЕНА, ОТЕЦ, МАТЬ, СЫН, ДОЧЬ, БРАТ, СЕСТРА, ДЕД, ДЯДЯ, ШУРИН, НЕВЕСТКА и т. д.) Система должна понимать сообщаемые ей факты и отвечать на вопросы пользователя.

Возможный пример диалога с системой (перед предложениями пользователя поставлены звездочки):

* (МАРИНА — ДОЧЬ ОЛЕГА)
ПОНЯТНО
* (ГАЛЯ — МАТЬ МАРИНЫ)
ПОНЯТНО
* (В КАКОМ РОДСТВЕ ОЛЕГ И ГАЛЯ ?)
(ОЛЕГ — МУЖ ГАЛИ)
* (ОЛЕГ — ОТЕЦ ТАРАСА)
ПОНЯТНО
* (ГАЛЯ — МАТЬ ТАРАСА)
Я ЗНАЮ
* (КТО СЫН ГАЛИ ?)
НЕ ЗНАЮ
* (ЛЕНА — ЖЕНА ТАРАСА)
ПОНЯТНО
* (КТО СЫН ГАЛИ ?)
ТАРАС
* (В КАКОМ РОДСТВЕ ЛЕНА И ГАЛЯ ?)
(ЛЕНА — НЕВЕСТКА ГАЛИ)
* КОНЕЦ

Прокомментируем этот диалог, объясняя принципы работы системы.

Если системе сообщается новая информация, то она записывает ее в свою базу данных и отвечает «понятно». При этом

в базу данных заносится не сам сообщенный факт, а два утверждения: о поле первого из лиц, указанных в факте, и о родственном отношении между обоими лицами, нейтральном по отношению к полу. Например, при поступлении факта «Марина — дочь Олега» в базу данных записываются утверждения «Марина — женщина» и «Олег — родитель Марины», а при поступлении факта «Галя — мать Марины» записываются утверждения «Галя — женщина» и «Галя — родитель Марины». Одновременно с записью таких фактов система выводит все возможные следствия из них. К примеру, система знает (так уж она устроена), что родители одного и того же ребенка являются супругами и что супруги — это лица разного пола. Поэтому после поступления второго факта система записывает в свою базу данных, что Олег и Галя — супруги и что Олег — мужчина. Это и позволяет системе ответить на вопрос о родственном отношении между Олегом и Галей.

Далее системе сообщается, что Олег — отец Тараса. Система записывает, что Олег — родитель Тараса (пол Олега она уже знает), и выводит отсюда, что и супруга Олега (Галя) также является родителем Тараса. Поэтому на следующий сообщенный ей факт «Галя — мать Тараса» система отвечает, что она уже знает об этом. (Система, получив любой факт, прежде всего пытается определить, не выводится ли он из уже известных данных; если выводится, то система ничего не записывает в свою базу данных и отвечает «я знаю».)

Следующее предложение пользователя — это вопрос «Кто сын Гали?». В данный момент система знает, что у Гали двое детей, Марина и Тарас, и что Марина — ее дочь. Пол же Тараса пока ей неизвестен, поэтому она и отвечает «не знаю». И лишь затем, когда ей сообщили факт «Лена — жена Тараса», из которого она узнала, что Тарас — мужчина, система смогла ответить на вопрос о сыне Гали.

После этого был задан вопрос о том, кем Лена приходится Гале. Пытаясь найти ответ, система перебирает все известные ей отношения родства и устанавливает, что Лена — жена сына Гали, т. е. невестка, о чем и сообщает пользователю. При поступлении слова КОНЕЦ система прекращает свою работу.

Прежде чем перейти к описанию нашей вопросно-ответной системы, сформулируем те предположения, на основе которых она построена. Во-первых, считается, что все лица, о которых сообщает пользователь, имеют разные имена. Во-вторых, подразумевается, что родители одного и того же ребенка обязательно являются супругами. (Очевидные предположения типа «супруги — это лица разного пола» мы не указываем.) В-третьих, предполагается, что пользователь указывает только допустимые факты и вопросы (он, например, не должен сообщать факты типа «Тарас —

брат Марины») и не сообщает противоречивых фактов (типа «Валя — дочь Нины» и «Маша — жена Вали»). Кроме того, чтобы соблюсти правила русского языка и в то же время не касаться сложной проблемы изменения имен по падежам, в систему заранее встраивается список имен (с указанием их именительного и родительного падежей), которые и может указывать пользователь. Данный список является значением константы ИМЕНА; ее определение может быть, например, таким:

```
[CSET ИМЕНА ((ОЛЕГ ОЛЕГА) (ГАЛЯ ГАЛИ)
   (ТАРАС ТАРАСА) (МАРИНА МАРИНЫ) (ЛЕНА ЛЕНЫ))]
```

В программе будет необходимо получать именительный падеж какого-либо имени по его родительному падежу и наоборот. Оба этих преобразования реализует одна функция:

```
[DEFINE ПАДЕЖ (LAMBDA (И1) [PROG (И2)
   [IS (< > [AUT (.И1 *И2) (*И2 .И1)] < >)
      :ИМЕНА]
   .И2])]
```

Данная функция используется, например, при обработке фактов, поступивших от пользователя. Если, скажем, системе сообщено (ОЛЕГ — МУЖ ГАЛИ), то она заменяет (с помощью функции ПАДЕЖ) слово ГАЛИ на слово ГАЛЯ, а затем преобразует этот факт к префиксному виду: (МУЖ ОЛЕГ ГАЛЯ). Списки такого вида служат образцами для вызова записывающих теорем, которые добавляют утверждения в базу данных и выводят следствия из них.

Отметим, что сами факты типа (МУЖ ОЛЕГ ГАЛЯ) не записываются в базу данных. Для системы базовыми являются понятия «мужчина», «женщина», «супруг» и «родитель», к которым она и сводит все остальные понятия родства. Например, вместо факта (МУЖ ОЛЕГ ГАЛЯ) система записывает утверждения (МУЖЧИНА ОЛЕГ) и (СУПРУГ ОЛЕГ ГАЛЯ), а вместо (ДОЧЬ МАРИНА ОЛЕГ) — утверждения (ЖЕНЩИНА МАРИНА) и (РОДИТЕЛЬ ОЛЕГ МАРИНА). Такие преобразования и записи выполняют следующие записывающие теоремы:

```
[DEFINE МУЖ (ANTEC (X Y) (МУЖ *X *Y)
   [PASSERT (МУЖЧИНА .X)]
   [PASSERT (СУПРУГ .X .Y) (TRY)])]
[DEFINE ЖЕНА (ANTEC (X Y) (ЖЕНА *X *Y)
   [PASSERT (ЖЕНЩИНА .X)]
   [PASSERT (СУПРУГ .X .Y) (TRY)])]
[DEFINE ОТЕЦ (ANTEC (X Y) (ОТЕЦ *X *Y)
```

```
    [PASSERT (МУЖЧИНА .X) (TRY)]
    [PASSERT (РОДИТЕЛЬ .X .Y) (TRY)])]
[DEFINE МАТЬ (ANTEC (X Y) (МАТЬ *X *Y)
    [PASSERT (ЖЕНЩИНА .X) (TRY)]
    [PASSERT (РОДИТЕЛЬ .X .Y) (TRY)])]
[DEFINE СЫН (ANTEC (X Y) (СЫН *X *Y)
    [PASSERT (МУЖЧИНА .X) (TRY)]
    [PASSERT (РОДИТЕЛЬ .Y .X) (TRY)])]
[DEFINE ДОЧЬ (ANTEC (X Y) (ДОЧЬ *X *Y)
    [PASSERT (ЖЕНЩИНА .X) (TRY)]
    [PASSERT (РОДИТЕЛЬ .Y .X) (TRY)])]
```

Итак, эти теоремы сводят сообщенные системе факты к более простым утверждениям. Однако они не занимаются выводом следствий из них. Для этого они вызывают (согласно рекомендации TRY в их функциях PASSERT) другие записывающие теоремы, которые уже и выводят все возможные следствия (в терминах базовых понятий) из накопленных фактов и записывают их в базу данных.

Например, если в базу данных было записано, что x — супруг y-а, то записывается также, что y — супруг x-а и что y — женщина (если известно, что x — мужчина) или y — мужчина (если x — женщина). Такой вывод реализует следующая теорема:

```
[DEFINE СУПРУГ1 (ANTEC (X Y) (СУПРУГ *X *Y)
    [PASSERT (СУПРУГ .Y .X)]
    [IF ([SEARCH (МУЖЧИНА .X)]
            [PASSERT (ЖЕНЩИНА .Y)])
        ([SEARCH (ЖЕНЩИНА .X)]
            [PASSERT (МУЖЧИНА .Y)])])]
```

Отметим, что если утверждение (ЖЕНЩИНА .Y) уже есть в базе данных, то эта теорема не будет записывать его вновь.

Из факта «x — супруг y-а» можно вывести и другое следствие: каждый ребенок x-а является и ребенком y-а, и наоборот. Этот вывод осуществляет такая теорема:

```
[DEFINE СУПРУГ2 (ANTEC (X Y) (СУПРУГ *X *Y)
    [FIND ALL (Z) .Z
            [SEARCH (РОДИТЕЛЬ .X *Z)]
            [PASSERT (РОДИТЕЛЬ .Y .Z)]]
    [FIND ALL (Z) .Z
            [SEARCH (РОДИТЕЛЬ .Y *Z)]
            [PASSERT (РОДИТЕЛЬ .X .Z)]])]
```

Из факта «x — родитель y-а» можно сделать следующие выводы: если у x есть супруг z, то z также является родителем

y-а, а если известен и другой родитель *y*-а, то можно записать, что оба родителя являются супругами. Такие следствия выводит теорема

```
[DEFINE РОДИТЕЛЬ (ANTEC (X Y Z)
        (РОДИТЕЛЬ *X *Y)
    [IF ([SEARCH (СУПРУГ .X *Z)]
        [PASSERT (РОДИТЕЛЬ .Z .Y)])
      ([SEARCH (РОДИТЕЛЬ [ET [NON .X] *Z] .Y)]
        [PASSERT (СУПРУГ .X .Z) (TRY)])])]
```

И, наконец, необходимы еще две теоремы, которые выводят следствия из фактов «*x* — мужчина» и «*x* — женщина»:

```
[DEFINE МУЖЧИНА (ANTEC (X Y) (МУЖЧИНА *X)
    [IF ([SEARCH (СУПРУГ .X *Y)]
        [PASSERT (ЖЕНЩИНА .Y)])])]
[DEFINE ЖЕНЩИНА (ANTEC (X Y) (ЖЕНЩИНА *X)
    [IF ([SEARCH (СУПРУГ .X *Y)]
        [PASSERT (МУЖЧИНА .Y)])])]
```

Первая из этих теорем вызывается, когда в базу данных записывается, что *x* — мужчина. Если в базе данных также есть утверждение о том, что *x* имеет супругу *y*, то эта теорема записывает, что *y* — женщина. Аналогично действует и вторая теорема.

На этом мы закончили описание той части программы, которая обеспечивает заполнение базы данных на основе фактов, сообщаемых пользователем. Теперь рассмотрим другую часть, которая используется при поиске ответов на вопросы пользователя. В нее входят целевые теоремы, которые сводят отношение родства, указанное в вопросе, к базовым отношениям и проверяют, выполняются ли они.

Например, для ответа на вопрос «*x* — муж *y*?» в базе данных отыскиваются утверждения о том, что *x* — супруг *y* и что *x* — мужчина. Такой поиск осуществляется следующей целевой теоремой (вопросы, как и факты, также преобразуются системой к префиксному виду):

```
[DEFINE МУЖ? (CONSEQ (X Y) (МУЖ *X *Y)
    [SEARCH (СУПРУГ .X .Y)]
    [SEARCH (МУЖЧИНА .X)])]
```

Аналогично определяются целевые теоремы для понятий «жена», «отец», «мать», «сын» и «дочь»:

```
[DEFINE ЖЕНА? (CONSEQ (X Y) (ЖЕНА *X *Y)
    [SEARCH (СУПРУГ .X .Y)]
    [SEARCH (ЖЕНЩИНА .X)])]
```

```
[DEFINE ОТЕЦ? (CONSEQ (X Y) (ОТЕЦ *X *Y)
    [SEARCH (РОДИТЕЛЬ .X .Y)]
    [SEARCH (МУЖЧИНА .X)])]
[DEFINE МАТЬ? (CONSEQ (X Y) (МАТЬ *X *Y)
    [SEARCH (РОДИТЕЛЬ .X .Y)]
    [SEARCH (ЖЕНЩИНА .X)])]
[DEFINE СЫН? (CONSEQ (X Y) (СЫН *X *Y)
    [SEARCH (РОДИТЕЛЬ .Y .X)]
    [SEARCH (МУЖЧИНА .X)])]
[DEFINE ДОЧЬ? (CONSEQ (X Y) (ДОЧЬ *X *Y)
    [SEARCH (РОДИТЕЛЬ .Y .X)]
    [SEARCH (ЖЕНЩИНА .X)])]
```

Примерно так же описываются и теоремы, отвечающие на вопросы о других родственных отношениях. Приведем лишь определение теорем, относящихся к понятиям «брат», «дядя» и «невестка»:

```
[DEFINE БРАТ? (CONSEQ (X Y Z)
            (БРАТ *X *Y)
    [SEARCH (РОДИТЕЛЬ *Z .Y)]
    [SEARCH (РОДИТЕЛЬ .Z .X)]
    [SEARCH (МУЖЧИНА .X)])]
[DEFINE ДЯДЯ? (CONSEQ (X Y Z)
            (ДЯДЯ *X *Y)
    [SEARCH (РОДИТЕЛЬ *Z .Y)]
    [ACHIEVE (БРАТ .X .Z)])]
[DEFINE НЕВЕСТКА1? (CONSEQ (X Y Z)
            (НЕВЕСТКА *X *Y)
    [ACHIEVE (ЖЕНА .X *Z)]
    [ACHIEVE (СЫН .Z .Y)]
    [SEARCH (ЖЕНЩИНА .Y)])]
[DEFINE НЕВЕСТКА2? (CONSEQ (X Y Z)
            (НЕВЕСТКА *X *Y)
    [ACHIEVE (ЖЕНА .X *Z)]
    [ACHIEVE (БРАТ .Z .Y)])]
[DEFINE НЕВЕСТКА3? (CONSEQ (X Y Z W)
            (НЕВЕСТКА *X *Y)
    [ACHIEVE (ЖЕНА .X *Z)]
    [ACHIEVE (БРАТ .Z *W)]
    [ACHIEVE (МУЖ .W .Y)])]
```

Для отношения «x — невестка y-а» мы определили три теоремы — по одной на каждый возможный вариант данного отношения: невестка — это жена сына (если y — женщина), либо жена брата, либо жена брата мужа.

Теперь приведем определение основной функции нашей вопросно-ответной системы, которая обеспечивает прием предложений

пользователя, их анализ и преобразование к префиксному виду, организует вызов записывающих и целевых теорем и выдает ответы пользователю в виде, удобном для восприятия (считаем, что функции ввода-вывода работают с терминалом):

```
[DEFINE РОДНЯ (LAMBDA ( ) [PROG (T X Y P L)
 A
 [SET T [READ]]
 [PERM [COND
 ([EQ .T КОНЕЦ] [RETURN ( )])
 ([IS (*X — *P *Y) .T]
    [SET Y [ПАДЕЖ .Y]]
    [IF ([GOAL (.P .X .Y)]
           [MPRINT Я ЗНАЮ])
        (T [DRAW (.P .X .Y)  (TRY)]
           [PRINT ПОНЯТНО])])
 ([IS (В КАКОМ РОДСТВЕ *X И *Y ?) .T]
    [IF ([ACHIEVE (*P .X .Y)]
           [PRINT (.X — .P [ПАДЕЖ .Y])])
        ([ACHIEVE (*P .Y .X)]
           [PRINT (.Y — .P [ПАДЕЖ .X])])
        (T [MPRINT НЕ ЗНАЮ])])
  ([IS (КТО *P *Y ?) .T]
    [SET Y [ПАДЕЖ .Y]]
    [SET L [FIND ALL (X) .X
              [GOAL (.P *X .Y)]]]
    [COND (.L [MPRINT !.L])
          (T [MPRINT НЕ ЗНАЮ])])
  (T [PRINT НЕПОНЯТНО])]]
 [GO A]])]
```

ЛИТЕРАТУРА

1. Лавров С. С., Силагадзе Г. С. Автоматическая обработка данных. Язык лисп и его реализация.— М.: Наука, 1978.
2. Маурер У. Введение в программирование на языке лисп.— М.: Мир, 1976.
3. Hewitt C. PLANNER: A Language for Manipulating Models and Proving Theorems in a Robot. Memo 68.— AI Lab., MIT, Cambridge, Mass., 1971.
4. Hewitt C. Description and Theoretical Analysis (Using Schemata) of PLANNER: A Language for Proving Theorems and Manipulating Models in a Robot. TR — 258.— AI Lab., MIT, Cambridge, Mass., 1972.
5. Sussman G., McDermott D. Why Conniving is Better than Planning. Memo 255.— AI Lab., MIT, Cambridge, Mass., 1972.
6. Derksen J. A. C. QA-4: A Language for Artificial Intelligence.— Stanford Research Institute, Menlo Park, Calif., 1973.
7. Wilber B. M. The QLISP Reference Manual. Tech. Note 118.— AI Center, SRI, Menlo Park, Calif., 1976.
8. Bobrow D. G., Winograd T. An Overview of KRL, a Knowledge Representation Language.— Cognitive Science, v. 1, № 1, 1977, p. 3—46.
9. Roberts R. B., Goldstein I. P. The FRL Primer. Memo 408.— AI Lab., MIT, Cambridge, Mass., 1977.
10. Брябрин В. М. Ф-язык — формализм для представления знаний в интеллектуальной диалоговой системе.— В кн.: Прикладная информатика, вып. 1. М.: Финансы и статистика, 1981, с. 73—103.
11. Warren D. H. D., Pereira L. M. PROLOG: The Language and its Implementation Compared with LISP.— SIGPLAN Notices, v. 12, № 8, 1977, p. 109—115.
12. Sussman G., Winograd T., Charniak E. MICRO-PLANNER Reference Manual. Memo 203.— AI Lab., MIT, Cambridge, Mass., 1971.
13. Виноград Т. Программа, понимающая естественный язык.— М.: Мир, 1976.
14. Davies D. J. M. POPLER 1.5 Reference Manual.— University of Edinburg, Scotland, 1973.
15. Пильщиков В. Н. Язык программирования ПЛЭНЕРБЭСМ.— М.: Изд-во Моск. ун-та, 1978.
16. Пильщиков В. Н. Система программирования ПЛЭНЕРБЭСМ.— М.: Изд-во Моск. ун-та, 1982.
17. Файкс Р., Нильсон Н. Система STRIPS — новый подход к применению методов доказательства теорем при решении задач.— В кн.: Интегральные роботы. М.: Мир, 1973, с. 382—403.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

УКАЗАТЕЛЬ ВСТРОЕННЫХ ПРОЦЕДУР ПЛЭНЕРА

Ниже перечислены в алфавитном порядке названия всех встроенных функций и сопоставителей языка плэнер с указанием страниц книги, где дается описание этих процедур.

Встроенные функции:

Встроенные сопоставители:

Владимир Николаевич Пильщиков

ЯЗЫК ПЛЭНЕР

(Серия: «Библиотечка программиста»)

Редактор *М. Г. Мальковский*
Техн. редактор *В. Н. Кондакова*
Корректоры *С. Н. Макарова, Е. В. Сидоркина*

ИБ № 12065

Сдано в набор 04.04.83. Подписано к печати 28.09.83. Т-19217. Формат 84×108$^{1}/_{32}$. Бумага тип. № 3. Обыкновенная гарнитура. Высокая печать. Условн. печ. л. 10,92. Уч.-изд л. 12,59. Тираж 17 000 экз. Заказ № 570. Цена 75 коп.

Издательство «Наука»
Главная редакция физико-математической литературы
117071, Москва, В-71, Ленинский проспект, 15

4-я типография издательства «Наука»
630077, Новосибирск, 77, Станиславского, 25

75 коп.