**Московский государственный
университет им.М.В.Ломоносова**

Факультет вычислительной математики и кибернетики

Большакова Е.И.

Практикум
на языке программирования Пролог

(Методическое пособие)

1999

УДК 519.6+681.3.06

В данном методическом пособии дается описание заданий практикума на языке программирования Пролог для студентов 4 курса факультета ВМиК МГУ. Задания разработаны в поддержку основных курсов «Математическая логика» и «Искусственный интеллект». Приводятся подробные методические пояснения и рекомендации.

Рецензенты:
доцент Боголюбов Д.П.
научный сотрудник Громыко В.И.

Большакова Е.И. “Практикум на языке программирования Пролог (Методическое пособие)”

Издательский отдел факультета ВМиК МГУ
(лицензия ЛР №040777 от 23.07.96), 1999.-23 с.

Печатается по решению Редакционно-Издательского Совета факультета вычислительной математики и кибернетики МГУ им. М.В.Ломоносова.

ISBN 5-89407-033-3????????

СОДЕРЖАНИЕ

ЗАДАНИЕ 1. ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ПРОЛОГ

Постановка задачи

Запрограммировать на языке Пролог следующие 20 предикатов, которые разбиты на 4 группы. При определении этих предикатов указывается условие их истинности.

I. Предикаты работы со списками

Аргументы L1,L2,L3 обозначают списки, E - некоторый элемент списка (тип элементов списка произволен), N - порядковый номер элемента в списке.

1. append (L1, L2, L3): список L3 является слиянием (конкатенацией) списков L1 и L2;
2. reverse (L1, L2): L2 - перевернутый список L1;
3. delete_first (E, L1, L2): список L2 получен из L1 исключением первого вхождения объекта E;
4. delete_all (E, L1, L2): L2 - это список L1, из которого удалены все вхождения E;
5. delete_one (E, L1, L2): L2 - список L1, в котором исключен один элемент E (исключается какое-то одно вхождение E в список L1);
6. no_doubles (L1, L2): L2 - это список, являющийся результатом удаления из L1 всех повторяющихся элементов;
7. sublist (L1, L2): L1 - любой подсписок списка L2, т.е. непустой отрезок из подряд идущих элементов L2;
8. number (E, N, L): N - порядковый номер элемента E в списке L;
9. sort (L1, L2): L2 - отсортированный по неубыванию список чисел из L1.

Предикаты работы со множествами

Аргументы M1, M2, M3 обозначают множества, которые представляются в виде списков элементов без повторений, порядок элементов в них не существенен, тип элементов - произволен.

10. subset (M1, M2): множество M1 является подмножеством M2;
11. union (M1, M2, M3): множество M3 - объединение множеств M1 и M2; вместо этого предиката может быть взят предикат intersection (M1, M2, M3): M3 - пересечение M1 и M2 или предикат substraction (M1, M2, M3): M3 - разность M1 и M2.

III. Предикаты работы с бинарными деревьями

Аргументы T, T1 и T2 обозначают деревья, представляемые в виде термов, записываемых с помощью тернарного функтора tree (левое поддерево, правое поддерево, метка) и константы nil, например:

tree (tree (nil, nil, f), tree (tree(nil, nil, p), tree (nil, nil, r), k))

представляет дерево, изображенное на рис.1. В вершинах дерева могут находиться объекты произвольного скалярного типа (в приведенном примере - символы).

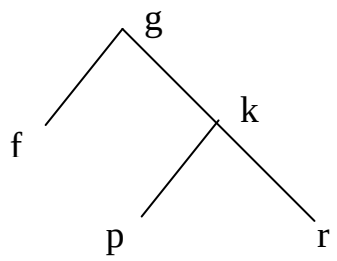


Рис. 1

12. tree_depth (T, N): N - глубина дерева (т.е. количество ребер в самой длинной ветви дерева);
13. sub_tree (T1, T2): дерево T1 является непустым поддеревом дерева T2;
14. flatten_tree (T, L): L - список меток всех узлов дерева T («выровненное» дерево);
15. substitute (T1, V, T, T2): T2 - дерево, полученное путем замены всех вхождений V в дереве T1 на терм T.

IV. Предикаты для работы с графами

Граф может быть представлен либо явно - в виде одной структуры, либо неявно - набором фактов вида edge (P, R, N), устанавливающих наличие ребра (дуги) между вершинами (узлами) P и R и стоимость N (целое неотрицательное число) этого ребра. При явном способе задания графа он представляется термом, включающим в свой состав список ребер (заданных аналогично с помощью тернарного функтора edge) и, возможно, список вершин графа.

Граф, изображенный на рисунке 2, может быть задан неявно фактами edge(a, c, 8), edge(a, b, 3), edge(c, d, 12), edge(b, d, 0), edge(e, d, 9), а в явной форме - например, термом graph([edge(a, c, 8), edge(a, b, 3), edge(c, d, 12), edge(b, d, 0), edge(e, d, 9)], [a, b, c, d, e]), где graph - бинарный фунтор.

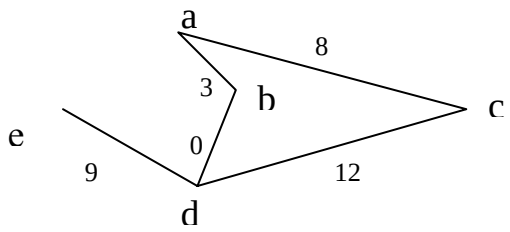


Рис. 2

В нижеследующем описании предикатов используется первый (неявный) способ задания графа, X и У обозначают вершины графа, а L - список вершин.

16. path (X, У, L): L - путь без петель между вершинами X и У, т.е. список вершин между этими вершинами;
17. min_path (X, У, L): L - путь между вершинами X и У, имеющий минимальную стоимость (стоимость пути равна сумме стоимостей входящих в него ребер);
18. short_path (X, У, L): L - самый короткий путь между вершинами X и У (длина пути равна количеству ребер, входящих в него);
19. cyclic : граф является циклическим (т.е. не является деревом);
20. is_connected : граф является связным (т.е. для любых двух его вершин существует связывающий их путь).

Замечания:
1) В предикатах 16-19 граф предполагается связным и может содержать циклы;
2) Все вышеописанные предикаты группы IV для второго способа представления графа должны содержать дополнительный аргумент - сам граф, например: path (G, X, Y, L), где G - структура, представляющая граф.

Требования к программе

Основные требования к программе связаны с особенностями языка Пролог, одна из которых - так называемая инвертируемость прологовских предикатов (процедур). В процедурах и функциях большинства операторных языков программирования (Паскаль, Си и др.) фиксируется, какие параметры (аргументы) являются входными (input), какие выходными (output), т.е. фиксируется направления передачи значений этих аргументов. Прологовский же предикат обычно допускает многообразное использование, т.е. один и тот же аргумент может быть как входным (при одних вычислениях), так и выходным (при других вычислениях), что связано с декларативным пониманием предиката как отношения между объектами. Например, предикат member (E, L), определяемый предложениями

member (E, [E| L]).
member (E, [Z| L]):- member (E, L).

и истинный, если E есть элемент списка L, допускает следующие варианты использования:

?-member (b, [a, b, c]): значения обоих аргументов заданы, результат вычислений - истинность данного отношения;
?-member (X, [a, b, c]): задано значение только второго аргумента, результат вычислений (значение X) - объекты, которые могут быть элементами заданного списка;
?-member (b, У): задан лишь первый аргумент, результат - списки, которые можно составить из объекта-элемента b.

Зафиксированные направления передачи значений аргументов (внутрь/вовне или input/output) предиката часто называют образцом или прототипом передачи (flow pattern), для его записи используются буквы i и o, а также скобки. Для рассмотренного предиката member соответствующие прототипы записываются как: (i, i), (o, i), (i, o). Таким образом, инвертируемый предикат - это предикат, допускающий несколько образцов передачи.

Другая часто встречающаяся особенность предикатов Пролога - недетерминированность. Предикат называется недетерминированным, если его вычисление может дать более одного решения/результата. Точнее говорить о недерминированности (или детерминированности) образцов передачи предиката. Например, для предиката member недетерминированными являются образцы (o, i) и (i, o).

Перечислим теперь требования к пролог-программе:

1. При программировании и отладке следует определить все возможные прототипы для всех 20 предикатов программы и поместить в тексте программы рядом с записью каждого предиката строку комментария, содержащую все обнаруженные прототипы. Прототипы, приводящие к недетерминированным вычислениям, необходимо пометить особо.
2. Для недетерминированных предикатов subset, path, short_path, min_path исключить возможность порождения дважды одного и того же результата (решения). Например, недопустимо возникновение дважды одного и того же пути во множестве решений предиката path.
3. Предикаты flatten_tree, short_path, min_path работы с деревьями и графами должны быть эффективно реализованы. Для flatten_tree потребуется использовать разностный вариант append вместо простого или же технику накапливающего параметра; а для предикатов поиска короткого и минимального пути в графе - выбор подходящего алгоритма перебора вершин графа.
4. Для всех предикатов работы с деревьями и графами необходимо заготовить тестовые данные, демонстрирующие различные результаты их работы для всех возможных прототипов.
5. Для реализации предиката sort допускается любой алгоритм сортировки, кроме пузырьковой: сортировка вставкой (включением), выбором, слиянием, быстрая сортировка и др.
6. Поскольку цель задания - освоение основ логического программирования, то в программе запрещается использовать внелогические предикаты, имеющие побочные эффекты: free, bound, read, assert, retract и другие.

Методические указания

1. При выборе способа представления графа для программирования предикатов группы IV следует учесть следующее. При простейшем, неявном способе зада-

ния графа легче программировать предикаты 16-18, однако он не применим для задания несвязных графов и работы с ними. Задание графа в виде терма - менее экономный способ, поскольку информация в нем частично дублируется, но он более универсален.

Кроме того, возможны и другие, еще более неэкономные и сложные способы представления (задания) графа, облегчающие в то же время его обработку; например, задание графа как списка пар вида

pair(<вершина>, <список вершин, смежных с ней>),

где pair - бинарный функтор. Для примера графа на рис. 2 таким списком будет

[pair (a, [b, c]), pair(b, [a, d]), pair(c, [a, d]), pair(d, [b, c, e]), pair(e, [d])].

При программировании некоторых предикатов может оказаться полезным перевод графа из одной формы представления в другую.

При любом способе задания графа ключевым моментом является то, что необходимо заранее решить, является ли обрабатываемый граф ориентированным или нет.

2. Несмотря на внешнюю похожесть определений предикатов min_path и short_path их эффективная реализация на Прологе требует применения разных алгоритмов поиска путей, т.е. перебора вершин в графе.

Нахождение минимального по стоимости пути предполагает полный просмотр графа и путей в нем. Такой просмотр целесообразно программировать на основе алгоритма поиска вглубь (depth_first_search), поскольку он по сути встроен в пролог-интерпретатор. При поиске вглубь всегда для продолжения просмотра из еще не рассмотренных вершин графа выбирается вершина, наиболее удаленная от начальной вершины (т.е. от которой был начат поиск) [Братко, с.330-335; Стерлинг, с.224-232].

Алгоритм поиска вглубь просто программируется и эффективно реализуется на Прологе, поскольку сам пролог-интерпретатор при доказательстве целей просматривает и обрабатывает альтернативы именно стратегией в глубину.

В отличие от min_path нахождение самого короткого пути (предикат short_path) в общем случае не требует полного просмотра графа; наиболее быстрое обнаружение такого пути гарантируется другим алгоритмом перебора вершин графа - алгоритмом поиска вширь (breadth_first_search). Для этого алгоритма характерно то, что всегда для продолжения поиска выбирается одна из вершин, наиболее близких к начальной.

Как и алгоритм поиска вглубь, поиск вширь является полным алгоритмом, т.е. при необходимости просматривает граф полностью, но порядок просмотра существенно иной.

Программирование на Прологе алгоритма поиска вширь существенно сложнее, так как для этого требуется сохранять и модифицировать в процессе перебора список всех путей, ведущих от начальной вершины к вершинам, от которых можно продолжать поиск [Братко, с. 336-344].

При программировании поиска вширь может оказаться полезным стандартный (встроенный) пролог-предикат второго порядка

findall (Var, Goal, Vlist),

где Var - переменная, Goal - предикат, имеющий в качестве одного из своих аргументов переменную Var, Vlist - выходная переменная - список возможных решений Goal. Сам findall детерминированный, его выполнение означает доказательство Goal в режиме бектрекинга: после каждого успешного окончания до-

казательства найденное значение Var добавляется в список Vlist и автоматически вырабатывается неуспех - до тех пор, пока не будут рассмотрены все варианты доказательства Goal. По окончании этого процесса предикат findall считается доказанным, а значением единственного выходного аргумента findall будет список из найденных значений переменной var.

Например, в результате доказательства

findall (X, append (X, _, [a, b, c]), Xlist)

Xlist получит значение ([a], [a, b], [a, b, c]).

3. Основой пролог-интерпретатора является встроенные механизмы унификации (сопоставления) и бектрекинга (возвратов). Для написания более эффективного варианта программы необходимо применять разностные структуры (списки, очереди и др.), позволяющие избегать ненужного копирования и просмотра структур (термов) в памяти за счет использования механизма сопоставления [Стерлинг, с.190-197].

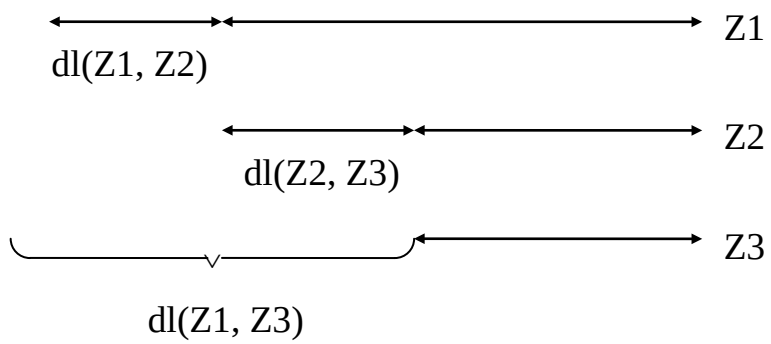
Наиболее часто используются разностные списки, идея которых связана с тем, что каждый список можно представить как разность двух других списков, причем не одним способом. Например, для списка [1,2,3] возможны следующие варианты:

dl ([1,2,3,4,5], [4,5])
dl ([1,2,3| У], У)
dl ([1,2,3], [])

Здесь dl - бинарный функтор для представления разностного списка, т.е. разности двух списков: первый его аргумент - уменьшаемое (укорачиваемое), второй - вычитаемое.

Таким образом, разностный список - другое представление обычного списка, при котором вычитаемый список (второй аргумент функтора dl) фиксирует конец обычного списка, и он становится сразу доступен в процессе сопоставления, без просмотра обычного списка от начала до конца. Поэтому если два обычных списка соединяются в один список предикатом append за время, линейное от длины первого аргумента, то два неполных разностных списка могут быть соединены в разностный список за константное время. Причем реализуется это преобразование - разностный append - одним предложением
append_dl (dl (Z1, Z2), dl (Z2, Z3), dl (Z1, Z3)).

Обоснованием этого предложения служит следующий схематический рисунок разностных списков:



Подчеркнем, что соединение разностных списков происходит неявно, в процессе унификации, поэтому операционное поведение программ с разност-

ными списками труднее для понимания и при отладке часто имеет смысл использовать трассировку.

Как правило, Пролог-программы с явными обращениями к предикату append могут быть переработаны в более эффективные за счет исключения простого append и использования разностных списков и разностного предиката append, что в ряде случаев по сути равносильно использованию переменных-накопителей.

Заметим, что имя функтора dl выбрано произвольно и может быть заменено на любой другой бинарный фунтор, и даже опущено - тогда первый и второй аргументы разностного списка становятся двумя отдельными аргументами предиката, использующего разностный список. При этом рассмотренный выше разностный append от трех аргументов превратится в

append_dl (Z1, Z2, Z2, Z3, Z1, Z2).

4. При программировании ряда предикатов могут потребоваться средства управления механизмом бектрекинга пролог-интерпретатора. В языке Пролог для этого используются два основных стандартных предиката без аргументов - предикаты fail и cut (обычно обозначаемый как !).

Предикат fail инициирует бектрекинг - при его выполнении происходит возврат к последней по времени точке бектрекинга (при этом автоматически восстанавливается вся операционная обстановка этой точки, включая значения переменных) и возобновляется доказательство текущей цели от этой точки. Fail по сути является предикатом, который никогда не выполняется, т.к. отсутствуют определяющие его предложения, поэтому можно использовать вместо fail любой неопределенный предикат с другим именем.

Заметим, что по смыслу предикат fail должен стоять последним в списке целей правой части любого пролог-правила (предложения).

Стандартный прологовский предикат ! предотвращает бектрекинг, в этом смысле его действие противоположно действию fail. Предикат ! употребляется для сокращения дерева доказательства цели путем отсечения некоторых его ветвей, поэтому он называется отсечением.

В общем случае выполнение отсечения ! в предложении S общего вида:
$P:- R_1, ..., R_k, !, R_{k+1}, ..., R_n.$

где $0 \le k \le n$,

относящемуся к пролог-процедуре P и используемого при доказательстве цели G, означает уничтожение всех последних по времени точек бектрекинга, возникающих с момента входа в процедуру P (т.е. начиная с поиска предложения этой процедуры, заголовок или левая часть которого унифицируема с текущей доказываемой целью G). Напомним, что под пролог-процедурой P понимается набор из всех предложений, в левой части которых стоит предикат P.

При выполнении отсечения, во-первых, отбрасываются все точки бектрекинга, возникающие при доказательстве целей R1, ..., Rk, расположенных левее предшествующих отсечению (то есть отбрасываются все альтернативные решения конъюнкции целей R1, ..., Rk), а во-вторых, уничтожается также точка бектрекинга, связанная с возможными альтернативами доказательства цели G, поэтому другие предложения процедуры P, заголовок которых унифицируем с G, будут при доказательстве отброшены.

В то же время выполненное отсечение не влияет на цели R_{k+1}, ..., R_n, расположенные правее его и возникающие при их доказательстве точки бектрекинга, таким образом, эти цели могут порождать более одного решения.

Однако, если при доказательстве конъюнкции R_{k+1}, ..., R_n возник неуспех (fail), и процесс возврата, исчерпав все альтернативы в возникших при этом доказательстве точках бектрекинга, достиг точки отсечения, то далее он распространяется до последней по времени точки бектрекинга, возникшей перед входом в процедуру P.

По семантике отсечения подразделяются на зеленые и красные [Стерлинг, с.127-132, 136-140]. Зеленые отсечения не изменяют декларативное значение (смысл) логической программы - множество возможных ее решений, то есть при них отсекаются ненужные ветви доказательства и уничтожаются лишние точки бектрекинга. Полезность таких отсечений определяется тем, что экономится время доказательства и, что более важно, необходимая память (вся связанная с точками бектрекинга информация запоминается в стеке).

Очевидно, что все вводимые в детерминированную программу отсечения являются зелеными (если они не отсекают единственное решение).

Красные отсечения изменяют декларативное значение программы, они обычно появляются, когда в недетерминированной программе необходимо по каким-либо причинам отбросить часть решений. Примером красного отсечения является отсечение в предикате member:

```
member (X, [X| Xl]):-!
member (X, [Y| Yl]):- member (X, Yl).
```

Вычисление в прототипе (o, i) приводит к поиску только первого вхождения элемента X в список Xl, поэтому результат вычислений будет отличаться от результата вычислений этого предиката, но без отсечения.

Отметим, что введение отсечения приводит к потере модульности предиката (становится важным порядок предложений в программе), а также очень часто - к потере его инвертируемости. Таким образом, программы с отсечениями менее гибки и менее декларативны, чем их аналоги без отсечений.

5. При программировании может оказаться полезным еще один стандартный пролог-предикат not, реализующий ограниченную форму отрицания - отрицание как безуспешное выполнение. Точнее, not(G) успешно завершает работу тогда, когда его аргумент - цель G - не может быть доказана (т.е. возникает неуспех).

Большинство реализаций Пролога запрещает использовать внутри цели G к моменту ее доказательства свободные переменные, за исключением анонимных, во избежание некорректностей, связанных с логической трактовкой их значений.

Семантика предиката not может быть определена двумя пролог-предложениями с помощью предикатов fail и cut (при этом G - метапеременная):

```
not(G):- G, !, fail.
not(G).
```

Таким образом, можно считать предикат cut слабой формой логического отрицания.

Во многих случаях при решении задачи подходит как предикат not, так и отсечение. Если же выбирать между этими предикатами, то, несмотря на мень-

шую эффективность, использование not предпочтительнее, чем менее понятная конструкция с отсечением.

ЗАДАНИЕ 2. ПРОЛОГ ДЛЯ ЗАДАЧ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Постановка задачи

Разработать пролог-программу, решающую определенную вариантом задания задачу из области искусственного интеллекта с использованием методов этой области: различных алгоритмов перебора (включая упорядоченный перебор) и способов ограничения перебора на основе эвристической информации (альфа-бета процедура, эвристическое отсечение ветвей дерева перебора и др).

Требования к программе

1. В вариантах заданий, предполагающих для поиска нужного решения задачи перебор с возвратом, необходима эффективная программная реализация этого перебора;
2. Программа должна быть организована как законченная система с удобным и понятным интерфейсом. Следует предусмотреть:
   -повторное выполнение основных функций системы после незначительного редактирования исходных данных;
   -сохранение исходных данных во внешнем файле;
   -выдачу необходимых в текущий момент подсказок и пояснений;
   -досрочное окончание работы пользователя с системой.

3. В вариантах заданий, в которых возможно несколько различных решений (для одних и тех же исходных данных), необходимо организовать случайный выбор одного из таких решений.

Варианты задания

1. Система, отвечающая на вопросы о родственных отношениях

Основным модулем системы должна быть база знаний, в которой хранится информация о членах (не менее 15-20 человек) семьи из нескольких (не менее 4-х) поколений и родственных отношениях между ними. Системе должны быть известны не менее 12-15 различных родственных отношений. Некоторые отношения - базовые - должны быть представлены в базе явно - в виде соответствующих фактов Пролога; например, к числу базовых отношений могут быть отнесены понятия «родитель», «супруги», «мужчина» и «женщина». Важно, что к базовым отношениям могут быть сведены все остальные отношения родства, например, «внук», «дядя», «невестка». Поэтому остальные отношения должны быть представлены в базе знаний неявно, т.е. должны выводиться пролог-процедурами исходя из базовых отношений. Таким образом, база знаний должна быть дедуктивной (обладать возможностями дедуктивного вывода). Отметим, что базовый набор отношений может быть выбран не единственным способом: например, вместо бинарного отношения «родитель» могут быть взяты отношения «мать» и «отец».

В возможности системы входит:

- ответы на запросы, которые могут быть двух видов:

1. Определить для двух конкретных членов семьи, в каком родственном отношении они находятся, например: «В каком родстве Елена и Петр?».
2. Определить для заданного члена семьи, кто состоит с ним в конкретном родственном отношении: например, «Кто является тетей Ольги?» или «Кто внуки Андрея?».

- модификация в диалоге с пользователем базы знаний: введение в нее новой информации о членах семьи и коррекция старой информации.

В случае ввода новых утверждений-отношений, не являющихся базовыми, они должны быть «разложены» на более простые утверждения с помощью базового набора отношений, которые и записываются в базу знаний.

Заметим, что при вопросах второго типа, если разница поколений заданных членов семьи превышает 2 поколения, необходимо в ответе произвести «синтез» искомого отношения из нескольких известных системе отношений, например: «Елена - сестра внука Петра» или «Елена - сестра Николая, внука Петра».

Предполагается, что все лица (члены семьи), известные системе, имеют разные имена - это необходимо для корректной работы системы. Чтобы исключить противоречия в базе знаний, желательно, чтобы при вводе пользователем новой информации система проверяла ее на непротиворечивость по отношению к текущему состоянию базы знаний и осуществляла только ввод допустимых фактов. Например, факт «Женя - дочь Андрея» недопустим, если в базе знаний хранится информация о том, что Женя - муж Ольги.

Интерфейс с пользователем может быть организован с помощью стандартных средств: меню, форматов для ввода/вывода, или же простейшего синтаксического анализа запросов с помощью ключевых слов [Ин, с.452-472].

Для соблюдения правил русского языка при выводе ответов на вопросы пользователя можно встроить в систему достаточно полный список имен, с указанием их именительного и родительного падежей (именно эти падежи используются в естественно-языковых фразах-вопросах указанных выше видов и ответах на них).

2. **Программа синтаксического анализа предложений естественного языка**

Предлагается рассмотреть некоторый ограниченный подъязык письменной речи одного из европейских языков (русский, английский, испанский и проч.). Входящие в такой подъязык предложения состоят в общем случае из группы подлежащего (именной группы) и группы сказуемого (глагольной группы). Группа подлежащего в свою очередь состоит из артикля, нескольких прилагательных и/или числительных, а также существительного или местоимения, а группа сказуемого - из глагола в одной из личных форм и нескольких дополнений или обстоятельств в виде групп существительных с предлогом или без, например, «Маленький мальчик долго играл с большой собакой в саду».

Необходимо зафиксировать структуру подъязыка с помощью некоторого обобщения контекстно-свободной грамматики [Стерлинг, с. 203-210; Клоксин, с. 234-255; Ин, с.473-488]. Грамматика должна учитывать согласование в лице и числе именной группы в роли подлежащего и глагольной группы в роли сказуемого и, возможно, согласование составных частей самой именной группы.

Построенная на базе указанной грамматики пролог-программа должна произвести синтаксический анализ (разбор) поступающего на вход предложения естественного языка, а также построить и визуализировать дерево разбора этого предложения. Встроенный в программу словарь слов (или основ) выбранного естественного языка должен включать не менее 20 единиц для каждой части речи (существительное, прилагательное, глагол и т.п.) и допускать расширения.

Реализацию синтаксического анализа следует осуществить эффективно, с использованием разностных списков или других разностных структур.

Усложнение рассмотренного варианта задания - создание пролог-программы, результатом синтаксического анализа которой является не дерево разбора, а эквивалентное по смыслу предложение другого естественного языка. Фактически такая программа производит перевод предложения с одного естественного языка на другой (например, с английского на французский). Для такого усложнения следует брать пару родственных языков. При выполнении задания кроме описывающей первый язык формальной грамматики потребуется также набор продукций вида: грамматическая конструкция первого языка → грамматическая конструкция второго языка.

3. Экспертная система классификации объектов (диагностического типа)

Назначение экспертной системы (ЭС) - проведение диалога-консультации, состоящего из нескольких вопросов к пользователю о наблюдаемых признаках некоторого объекта, и затем решение на основе ответов пользователя, к какому классу (типу, виду) принадлежит этот объект, например, на основе внешних признаков музыкального инструмента (размера, формы, наличия клавишей, струн, смычка и т.п.) система определяет вид музыкального инструмента (скрипка, балалайка, рояль и др.).

Основными модулями экспертной системы должны быть [Братко, с. 414-421]:

- база знаний, состоящая из продукций (правил вывода) вида: условие → следствие, каждая из которых воплощает некоторый фрагмент знания, необходимый для классификации объектов (знаний в конкретной проблемной области);
- механизм вывода (решатель), который осуществляет поиск решения, т.е. нужной, решающей цепочки продукций, представляющей последовательные шаги заключения о классе объекта.

Важно, что полученное системой решение (о классе объекта) при необходимости может быть пояснено пользователю - это одна из отличительных особенностей ЭС. Для этого экспертная система должна уметь отвечать на вопросы вида «как?» и «почему?».

Вопросы «как?» (т.е. как получено указанное решение?) задаются по окончании диалога-консультации, в качестве ответа система визуализирует решающую цепочку продукций. Вопросы «почему» (почему запрашивается этот признак?) могут быть заданы в ходе диалога, в ответ система показывает правило продукции, которое она пробует в текущий момент применить. Модуль ЭС, реализующий такие поясняющие ответы, называют подсистемой объяснения.

Вывод решения (поиск решающей цепочки) может производиться в прямом направлении - от известных фактов, т.е. наблюдаемых признаков объекта, к следствиям - заключениям о классе или подклассе объекта (так называемый прямой вывод - см. [Ин, с. 406-444]), или наоборот - от гипотез о возмож-

ном классе или подклассе объекта к фактам, их подтверждающим (так называемый обратный вывод - см. [Братко, с.426-455]).

Поскольку в процессе вывода выявляются обычно недостающие для нужного заключения факты, которые необходимо запросить у пользователя, то ход диалога-консультации определяется фактически механизмом вывода и набором правил из БЗ. В силу этого интерфейс с пользователем часто не выделяют в отдельный модуль, а относят к механизму вывода.

В качестве возможной проблемной области (области экспертизы) может быть взята классификация плодов (признаки: цвет, форма, вкус, количество косточек и пр.), грибов (признаки: цвет шляпки, толщина ножки, наличие пятен в окраске, особенности роста - отдельно, группой и проч.), пород собак (признаки: вид и длина шерсти, размер, масть, тип ушей - стоячие, лежачие и проч.), а также транспорта, оружия, бабочек, автомобилей, напитков и т.п. Мощность базы знаний должна позволять экспертной системе распознавать на менее 25 различных классов объектов по нескольким (не менее пяти) признакам.

4. Программа, составляющая головоломку крисс-кросс

Рассматриваемая головоломка крисс-крос предлагает задачу более простую, чем кроссворд, а именно: заданы набор слов и схема, подобная сетке кроссворда. В эту схему необходимо вписать все слова. Схема состоит из пересекающихся, но не соприкасающихся линий из клеток [Уэзерелл, с. 55-56]; причем схема должна быть связной - см. рисунок 4.

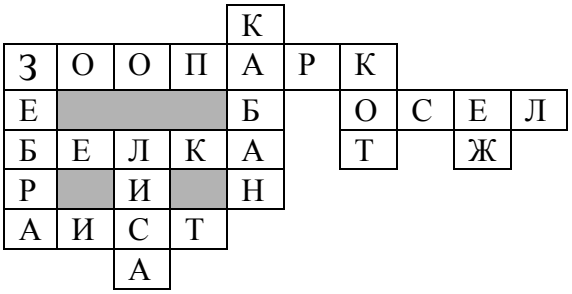


Рис. 4

Программа составления головоломки, получая на вход набор из 7-15 различных слов одного из естественных языков - русского или английского - должна построить подходящую для них схему или несколько схем. Схема построена правильно, если она связная и все слова могут быть вписаны в нее, причем допускается лишь один вариант ее заполнения (т.е. головоломка имеет единственное решение).

Если для данного списка слов не существует решения-схемы, то программа должна сообщить об этом, указав по-возможности причину неудачи (например, наличие слова, «непересекающегося» с другими словами). В остальных случаях построенная схема должна быть изображена на экране компьютера (в текстовом или графическом режиме), и по желанию пользователя может быть показано решение головоломки, т.е. заполнение схемы исходными словами.

В случаях, когда для заданного набора слов возможно несколько схем, предпочтение следует отдавать более качественным решениям - т.е. компактным и связанным схемам. Связность схем зависит от количества пересечений в ней, она может определяться как среднее или минимальное число пересечений на слово. Компактность схемы определяется площадью наименьшего объемлющего ее прямоугольника. Указанный эвристический критерий качества схемы рекомендуется формализовать тем или иным способом (возможно, в виде эвристической функции) и использовать при поиске правильных схем.

Отметим, что длина слова из заданного набора и количество слов одинаковой длины служат важным ключом как к разгадке самой головоломки, так и к написанию программы их составления. Оптимальная организация перебора вариантов схем при поиска программой правильной схемы требует установления определенного порядка, в каком будут рассматриваться слова исходного набора: имеет смысл применить эвристическое упорядочивание набора и переупорядочивание нерассмотренных слов на любом шаге поиска, а также эвристическое отсечение части вариантов схем, поскольку в иных случаях перебор может оказаться слишком большим. Проверку же условия единственности решения головоломки целесообразно проводить как можно раньше, чтобы исключить лишнюю работу - построение до самого конца вариантов схем, допускающих несколько решений головоломки.

5. **Программа построения прямоугольного лабиринта**

Рассмотрим лабиринты, расположенные в прямоугольной области M x N и состоящие из стенок внутри и на границе области, являющихся сторонами некоторых квадратов (клеток) из покрывающей эту область равномерной сетки [Уэзерелл, с.57-58]. Лабиринты имеют один вход - на одной из сторон прямоугольной области и один выход - на противоположной стороне. Такой лабиринт может быть получен из равномерной сетки стенок в результате выбивания ровно двух граничных стенок на противоположных сторонах рассматриваемого прямоугольника и удаления некоторого количества внутренних стенок.

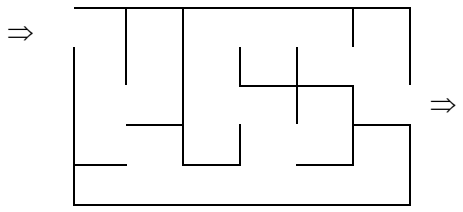


Рис. 5

Лабиринт проходим, если внутри него между стенками существует путь, соединяющий вход и выход. Будем говорить в этом случае, что лабиринт имеет решение. Решение единственно, если среди всех таких путей есть минимальный по длине путь, являющийся частью всех остальных путей. Клетка лабиринта достижима, если существует путь, соединяющий ее с входом или выходом.

Требуемая пролог-программа должна по заданным M и N ($5 \le M, N \le 30$) строить прямоугольный лабиринт, который имеет единственное решение и со-

стоит только из достижимых клеток. Тот факт, что все клетки лабиринта достижимы, означает, что в лабиринте отсутствуют замкнутые области из клеток.

Программа должна визуализировать построенный лабиринт, отметить вход и выход и по указанию пользователя показать путь между ними. При каждом обращении к программе, даже с одними и теми же значениями M и N, она должна порождать разные лабиринты. Для этого необходимо в некоторые моменты построения лабиринта производить «случайный» выбор варианта дальнейшего продолжения.

Желательно, чтобы построенный лабиринт был интересным - содержал извилистые стенки и почти замкнутые комнаты, тогда и решение-путь в нем будет достаточно извилистым.

Заметим, что стратегии генерации лабиринта могут быть весьма различны: удаление внутренних стенок, при этом можно сначала построить путь, а затем достроить оставшуюся часть лабиринта, или же их добавление. При любой стратегии единственность решения, как и достижимость клеток лабиринта, целесообразно проверять как можно раньше, либо же гарантировать выполнение этих требований самим методом построения лабиринта.

6. **Игровая программа**

Предлагается выбрать игру из класса игр двух лиц (игроков) с полной информацией [Братко, с.472-475], к которому относятся, например, шахматы и шашки. Игра должна быть достаточно сложной, чтобы практически исключалась возможность полного просмотра дерева игры и обнаружения выигрышной стратегии (если таковая существует), например, шашки без дамок, калах [Уэзерелл, с. 76-78], шахматный эндшпиль, реверси и крестики-нолики на неограниченной доске.

В таких играх для выбора компьютером очередного хода используется так называемая статическая оценочная функция, оценивающая позицию игры как таковую без учета ее продолжений, и альфа-бета процедура [Уэзерелл, с. 78-86; Братко, с. 479-484] поиска наилучшего хода, исходя из заданной игровой позиции. Альфа-бета процедура основана на частичном просмотре возможных продолжений игры на заданное количество D ходов игроков, т.е. просмотре дерева игры от заданной игровой позиции на глубину дерева D, и оценки возможных игровых позиций с помощью статической оценочной функции. В отличие от минимаксной процедуры, решающей ту же задачу, альфа-бета процедура выполняет просмотр дерева и оценку вершин более экономно.

Требуемая игровая программа должна использовать альфа-бета процедуру, оформленную в виде отдельного модуля. Она должна уметь играть как против человека (пользователя), так и против самой себя или другой игровой программы, и при этом визуализировать текущую позицию игры, а ввод ходов игроков осуществлять в наиболее удобной форме.

В программе следует предусмотреть возможность изменения (перед началом игры) глубины D просмотра дерева игры альфа-бета процедурой ($2 \le D \le 6$, шаг глубины соответствует ходу одного игрока), а также случайный выбор хода из нескольких равноценных - чтобы с программой было интересно играть.

7. **Программа генерации геометрических головоломок**

В таких головоломках требуется найти закономерность в наборе из нескольких составных геометрических фигур. Составными элементами (не менее

8 различных элементов, включая цвет) каждой фигуры могут быть: точка, квадрат, треугольник, прямоугольник, круг, различные отрезки линий и др. Такие элементы могут быть соединены в фигуре различным образом: могут быть вложены друг в друга, могут пересекаться или соприкасаться [Стерлинг, с.182-184, 189]. Кроме того, контуры элементов могут быть разного цвета, а их внутренняя часть - заштрихованной или цветной. Примеры наборов таких составных фигур показаны на рис. 6а и 6б. Могут быть и более сложные фигуры - например, схематические изображения человечков или животных.

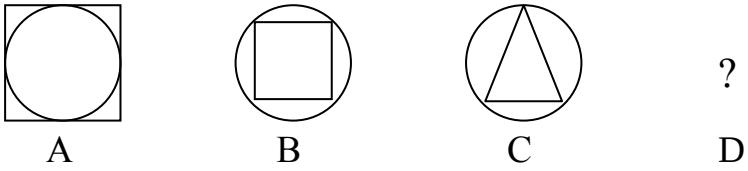


Рис. 6а

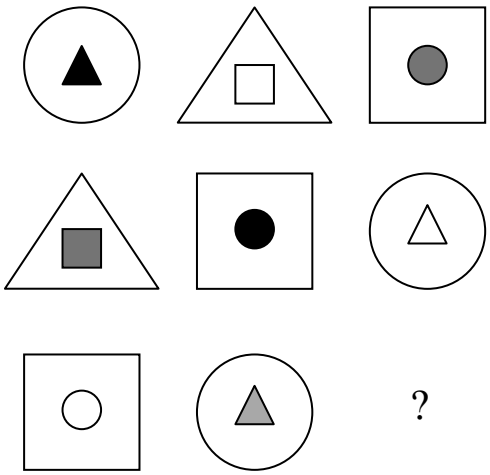


Рис. 6б

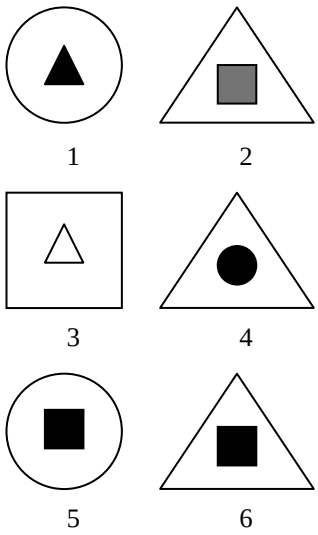


Рис. 6в

Поиск закономерности предполагает обнаружение недостающей фигуры (обозначенной на рисунках вопросительным знаком). В первом случае (рис. 6а) необходимо обнаружить аналогию, т.е. найти фигуру D, относящуюся к C, как фигура B - к фигуре A. Во втором случае (рис. 6б) это фигура, завершающая закономерность.

В обоих типах задачи поиска закономерностей предлагается несколько вариантов ответа (например, для второй задачи - рис. 6в), среди которых надо выбрать правильный. Правильный ответ второй задачи - фигура 6 на рис. 6в, а правильный ответ первой задачи - фигура из треугольника, в который вписан круг.

Программа поиска закономерностей должна сгенерировать задачу на поиск геометрической закономерности - то есть набор составных фигур без некоторой недостающей фигуры и набор различных составных фигур, включающих эту недостающую фигуру, показать ее пользователю и предложить ему на

выбор либо самому решить задачу (в этом случае она проверяет правильность решения и сообщает о результате пользователю); либо, если он затрудняется задачу решить, показать ему правильное решение.

Важно предусмотреть возможность решения пользователем за сеанс работы с программой нескольких задач - для этого программа при каждом обращении к ней должна порождать разные задачи, причем в разные сеансы последовательности генерируемых задач должны быть разными. Чтобы этого достичь, следует в некоторые моменты процесса генерации задачи производить случайный выбор составных элементов фигур и их конфигурации, используя датчик случайных величин и доступное программе значение, заранее неизвестное: дату и/или текущее время. Для расширения множества генерируемых задач необходимо, чтобы геометрические фигуры задачи включали не менее 5 элементов, включая цвет и штриховку.

Программа построения геометрических головоломок должна быть организована таким образом, чтобы допускать быструю перенастройку системы для генерации задач, в которые входят фигуры с другими составными элементами. Такая перенастройка должна осуществляться без переработки пролог-процедуры генерации головоломки - допустимы лишь добавления новых процедур отрисовки новых элементов фигур.

8. **Программа разработки маршрутов транспортных перевозок**

Рассматривается сеть одноколейных железнодорожных путей, соединяющих N ($5 \le N \le 10$) городов; известны длины всех путей (дорог) в километрах. Задан некоторый набор из M ($3 \le M \le 9$) заявок на грузоперевозки по этой железной дороге в течение текущих суток. Каждая заявка включает в общем случае следующую информацию:

- пункт (город) отправления грузового состава;
- пункт (город) назначения грузового состава;
- максимальную скорость движения грузового состава по путям (не более 100 км в час);
- час суток, не позднее которого груз должен прибыть в пункт назначения.

Требуемая программа должна составить - исходя из заданного набора заявок - расписание движения грузовых составов, состоящее из M маршрутов. Каждый маршрут соответствует заявке и фиксирует кроме пунктов (городов) отправления и назначения следующее:

- время (в часах) отправления из исходного пункта;
- время (в часах) прибытия в конечный пункт;
- промежуточные пункты, через которые проходит маршрут, и время остановки в них;
- скорость движения на каждом участке пути между двумя городами, входящими в маршрут.

Считается, что скорость движения каждого состава на любом участке пути постоянна. Она может быть разной для разных составов, но не выше указанной в заявке и не ниже 20 км в час. Каждый грузовой состав может делать остановки на промежуточных пунктах маршрута, но они не должны по длительности превышать 3 часа. Через любой город в одно и то же время может проезжать только один состав.

Основное требование, предъявляемое к составлению маршрутов перевозок - протяженность каждого маршрута между заданными городами отправления и назначения не должна превышать минимальное расстояние между ними по этой железной дороге более, чем в 1,5 раза.

Программа должна визуализировать карту дорог, на которой затем должны быть высвечены найденные маршруты. Желательно также в динамике показать прохождение грузовых составов по этим маршрутам в течение суток (шаг пересчета местонахождения составов - 0,5 часа).

9. **Программа составления учебного расписания**

Назначение программы - составление недельного расписания занятий для курса, включающего N ($7 \le N \le 12$) студенческих групп. Исходной информацией для составления расписания являются:

- учебный план курса, определяющий названия изучаемых предметов (не более 7 разных), и количество учебных занятий в неделю для изучения каждого предмета (не более 5 занятий в неделю);
- список-распределение преподавателей по группам, в котором для каждой учебной группы и каждого изучаемого предмета указывается фамилия преподавателя, который будет вести этот предмет в этой группе.

Известно, что суммарное количество учебных занятий по учебному плану не превышает 19 занятий в неделю, а недельная нагрузка каждого преподавателя не превышает 7 занятий.

При генерации расписания для студенческих групп желательно более или менее равномерное распределение занятий по дням недели (от понедельника до субботы включительно). Оптимальным следует считать 2-3 занятия в день. В составленном расписании должны быть выполнены следующие требования:

1. В каждый день недели у любой учебной группы не может быть больше 4-х занятий, а у любого преподавателя - не больше 3-х занятий;
2. В каждый день недели у любой группы по расписанию не может быть больше одного «окна» (перерыва) между занятиями, причем протяженность «окна» - не более чем одно занятие;
3. Для каждого преподавателя определяемое расписанием распределение его учебной нагрузки по дням недели должно быть достаточно «плотным», т.е. в нем не должно быть более одного дня всего лишь с одним занятием, а «окна» между занятиями в каждый учебный день в сумме не должны быть больше 2 занятий.

Основным результатом работы программы является построенное учебное расписание занятий для всех групп, определяющее для каждой группы и каждого учебного дня все занятия этого дня. Для каждого занятия в расписании должны быть заданы: номер и время начала занятия, предмет, фамилия преподавателя. Возможные номера занятий - от 1 до 6, они обозначают порядок следования по времени в течение дня указанных занятий: например, за номером 1 может быть закреплено время 9.00, за номером 2 - 11.00 и т.д.

Составленное учебное расписание группы должно быть в понятном и обозримом виде выведено на экран компьютера. Необходимо также предусмотреть возможность вывода (показа) определяемого этим расписанием индивидуального расписания каждого преподавателя, т.е. распределения по дням его учебной нагрузки. Это расписание для каждого учебного дня указывает, есть ли

занятия в этот день, и если есть, то для каждого занятия - их номер и время, предмет, номер группы.

Методические указания

Так как большинство описанных вариантов заданий предполагает проведение в программе большого перебора при поиске решения, то может возникнуть переполнение памяти, отведенной под стек или кучу. Часто конечной причиной переполнения является неэффективность организации перебора в программе, что требует ее соответствующей перестройки.

Используемый пролог-интерпретатором стек сохраняет, во-первых, всю информацию, необходимую для запоминания точек бектрекинга, а во-вторых, информацию, необходимую для выполнения функциональных вызовов и рекурсии (размер стека линейно растет с ростом количества точек бектрекинга и рекурсивных обращений). Соответственно возможны две основные причины переполнения стека.

В первом случае переполнение стека происходит из-за возникновения неоправданно большого количества точек бектрекинга. Часто, используя отсечение для сокращения ненужных точек бектрекинга, можно избежать такого переполнения стека.

Во втором случае переполнение возникает из-за большого количества рекурсивных обращений. Сократить число рекурсивных обращений может преобразование пролог-программы в так называемую хвостовую рекурсию, поскольку для такого вида рекурсии пролог-интерпретатором может быть использован метод, называемый оптимизацией остатка рекурсии [Стерлинг, с.132-133].

Основная идея такой оптимизации состоит в том, что оптимизируемая рекурсивная пролог-процедура выполняется так, как если бы она была итерационной, т.е. без стека. Метод применим далеко не ко всякой рекурсивной пролог-процедуре; чтобы применить его при выполнении предложения

A:- B_1, B_2, ..., B_n.

процедуры A необходимо соблюдение следующих условий :

1. В указанном предложении рекурсивное обращение единственно и стоит в конце правой части, т.е. является последней целью правой части: Bn=A (отсюда произошло название: хвостовая рекурсия).
2. С момента входа в пролог-процедуру до вычисления этого рекурсивного обращения не осталось или не возникало точек бектрекинга, т.е. вычисление конъюнкции всех целей B_i , кроме последней цели B_n, было детерминированным и не осталось иных применимых предложений процедуры A.

Таким образом, при возникновении в ходе вычисления пролог-программы переполнения стека полезно провести перестройку рекурсивных процедур пролог-программы (если, конечно, это возможно). Такая перестройка означает перестановку рекурсивных обращений в конец пролог-предложения и вставку перед ними отсечения (тем самым становятся выполнены условия оптимизации).

В практике программирования на языке Пролог иногда необходимы:

- глобальные переменные;
- циклы, управляемые неуспехом.

Глобальные переменные моделируются в Прологе с помощью базы фактов и предикатов assert и retract. Например, запись в базу факта a(Value) означает присвоение переменной a значения Value [Стерлинг, с.158].

Циклы, управляемые неуспехом, получили свое название потому, что для порождения шагов цикла используется не рекурсия, а механизм бектрекинга включаемый неуспехом fail [Стерлинг, с.158]. Рассмотрим, например, простой рекурсивный цикл для ввода и вывода символов до «звездочки» (предикат begin служит для входа в цикл):

```
begin : - readchar (C), while (C).
while (C): - C='*'.
while (C): - write (C), readchar (Z), while (Z).
```

Такой цикл можно запрограммировать иначе, если использовать специальный незавершаюшийся предикат repeat без аргументов, который порождает при бектрекинге бесконечные вычисления:

```
repeat.
repeat: - repeat.
```

Цикл будет выглядеть так:

```
begin (X): - repeat, readchar (C), while (C), !.
while (C): - C='*', !.
while (C): - write (C), fail.
```

Сам предикат while вычисляется успешно лишь в момент выхода из цикла (применение первого предложения). Отсечение в первом предложении процедуры while предохраняет от незавершающихся вычислений (оно отсекает точку бектрекига, связанную с предикатом while), а отсечение в процедуре begin - от повторного входа в цикл repeat (оно отсекает точку бектрекинга, возникающую при выполнении цели repeat).

Заметим в заключение, что циклы, управляемые отказом, не имеют логической интерпретации, и в этом смысле они хуже рекурсивных циклов.

ЛИТЕРАТУРА

[Стерлинг] Стерлинг л., Шапиро Э. Искусство программирования на языке Пролог - М.: Мир, 1990.

[Братко] Братко И. Программирование на языке Пролог для искусственого интеллекта - М.: Мир, 1990.

[Клоксин] Клоксин У., Меллиш К. Программирование на языке Пролог - М.: Мир, 1987.

[Ин] Ин Ц., Соломон Д. Использование Турбо-Пролога - М.: Мир, 1993.

[Набебин] Набебин А. А. Логика и Пролог в дискретной математике - М.: Издательство МЭИ, 1996.

[Уэзерелл] Уэзерелл Ч. Этюды для программистов - М.: Мир, 1982.