3. ИТЕРАЦИЯ, РЕКУРСИЯ И РЕКУРСИВНЫЕ ДАННЫЕ

Итерация является основой вычислительных методов для решения многих задач и представляет собой повторение преобразования, приближающего к решению.

Итерация – это процесс вычислений, основанный на повторении фиксированной последовательности операций, при котором на каждом шаге повторения используется результат предыдущего шага.

Окончание итерации происходит по достижению некоторого условия. Если итерационный процесс сходится, то это значит, что процесс заканчивается результативно. Если расходится, то:

- либо следует изменить начальные условия;
- либо основа итерационного процесса неправильно построена;
- либо задача не имеет решения.

В традиционных языках программирования итеративные задачи решаются при помощи операторов цикла. В логических и функциональных языках подобных операторов не существует, как и не существует прямого способа выражения повтора. *Пролог* обеспечивает два вида повторения: *откат* (который ищет все решения для одного целевого утверждения) и *рекурсия* (в которой процедура вызывает сама себя).

3.1. Поиск с возвратом в итеративных задачах

Когда выполняется процедура поиска с возвратом (откат), происходит поиск другого решения цели. Это реализуется путём:

- возврата к последней из проверенных подцелей, имеющей альтернативные решения;
- использования следующей альтернативы этой подцели;
- и новой попытки движения вперёд.

Пример 3-1. Использование поиска с возвратом.

```
predicates  nondeterm просмотр
facts       страна (symbol)
clauses     страна (россия). страна (франция). страна (англия).
            просмотр :-                    % 1-ое правило
                  страна (С), write (С), nl,
                  страна (лимония).
            просмотр.                      % 2-ое  правило
goal  просмотр.
```

Чтобы просмотреть страны, надо найти решение **страна (С)**, вывести значение **write (С)**, начать новую строку **nl** и проверить **страна (лимония)**, то есть инициировать неудачное завершение.

В этом случае неудачное завершение означает следующее:

Принять, что решение цели не было достигнуто, поэтому вернуться назад и искать альтернативное решение.

Переменная **C** последовательно связывается и освобождается при проверке всех альтернатив для подцели **страна (C)**. Последняя неудача приводит к согласованию правила %2 процедуры **просмотр**, которое всегда успешно. Программа завершается. При отсутствии второго правила после вывода всех известных программе стран выводилось бы сообщение **no**.

Пример 3-2. Использование отката с петлями.

Типичным примером итеративной программы является циклическая программа на языке С, выполняющая эхо–печать символов:

```
void main ( ) { char c; do {c = getchar ( ); putchar (c); } while (c != 13); }
```

В *Прологе* нет циклов, поэтому проблема итерации решается искусственным путём. Для решения задачи эхо–печати символов определяется предикат **repeat** с двумя предложениями. Программа на *Прологе* имеет вид:

```
predicates  nondeterm repeat
            nondeterm typewriter
goal        typewriter.
clauses     repeat.                 % 1
            repeat :– repeat.       % 2
            typewriter :–
                  repeat,           % 3
                  readchar (C),     % 4
                  write (C),        % 5
                  C = '\13'.        % 6   char_int (C, 13)
```

Предикат **typewriter** работает следующим образом:

1. Выполняется предикат **repeat** (%1), который ничего не делает.
2. Переменная **C** получает связанное значение символа (%4).
3. Отображается связанная переменная **C** (% 5).
4. Предикат **=** работает как проверка отношения равенства, то есть – соответствия значения переменной **C** коду клавиши **<Enter>** (% 6).
5. Если **ДА**, то завершение программы.
6. Если **НЕТ**, то возврат для поиска альтернатив. Так как ни предикат **write**, ни предикат **readchar** не являются альтернативными, то происходит возврат к предикату **repeat**, который всегда имеет альтернативные решения (%2).
7. Ввод (% 4) и отображение (% 5) могут продолжаться вперёд с последующим сравнением с кодом **<Enter>** (% 6).

Следует заметить, что переменная **C** освобождается *(теряет своё значение)* после возврата, только поэтому выполнение предиката **readchar** становится возможным.

Проверка цели с вводимыми символами **ab <Enter>**

1. Подцель % 3 достигается, так как правило %1 истинно.
2. Подцель % 4 **C = 'a'** и подцель % 5 **a** достигаются.
3. Подцель % 6 *не достигнута*, так как **'a' = '\13'** ложь.
4. Откат *(backtracking)* к подцели % 3, при этом **C** теряет своё значение.
5. Подцель % 3 достигается по альтернативному правилу % 2, при этом правило % 2 достигается, так как истинна его правая часть из-за правила % 1.
6. Подцель % 4 **C = 'b'** и подцель % 5 **b** достигаются.
7. Подцель % 6 *не достигнута*, так как **'b' = '\13'** ложь.
8. Откат к подцели % 3, при этом **C** теряет своё значение.
9. Попытка разрешить правило % 2 путём другой альтернативы для правой части, то есть решить правило % 2 через правило % 2.
10. Подцель % 4 и подцель % 5 достигаются, при этом переменная **C** конкретизируется значением **'\13'**.
11. Подцель % 6 *достигнута*, так как **'\13' = '\13'** истина.

Цель доказана.

Как видно из примера, хотя **поиск с возвратом** и может повторить операции сколь угодно раз, но он **не способен запомнить** что-либо из одного повторения для другого. Все переменные теряют свои значения, когда обработка возвращается и проходит те шаги, на которых эти значения устанавливались. С другой стороны, итерационные программы обычно используют фиксированный объём памяти, не зависящий от числа итераций.

Есть другой эффективный метод реализации повторений – рекурсия.

3.2. Рекурсия

В математике рекурсией называется способ описания функций или процессов через самих себя.

В языках программирования вычисление обычно осуществляется с помощью подпрограммы. Идея подпрограммы, которая сама обращается к другой подпрограмме, общеизвестна. Рекурсивной функцией является подпрограмма, которая содержит обращение самой к себе.

В численных задачах рекурсия используется только для удобства, позволяя сжато описывать подлежащий выполнению алгоритм.

Применение рекурсии при использовании языков логического и функционального программирования является обязательным, так как по самой своей природе они не могут обеспечить каких-либо средств для явного повто-

рения. Кроме этого, рекурсия необходима из-за рекурсивной природы обрабатываемых данных. В логических программах рекурсия играет ту же роль, что и повторение в программах, состоящих из команд.

Разумеется, для поддержки рекурсии требуется специальное программное и аппаратное обеспечение, так как нужна полная информация об иерархии связей и рабочих регистров для того, чтобы можно было восстановить первоначальное состояние рекурсивной процедуры. Именно этим рекурсивные системы программирования отличаются от нерекурсивных. При выполнении рекурсии каждый рекурсивный вызов, не завершённый к данному моменту, требует для параметров и локальных переменных определённой структуры данных, называемой *фрагментом стека*. Таким образом, размер области памяти для вычисления, включающего **N** рекурсивных обращений к процедурам, линейно зависит от **N**.

Логическая программа обладает процедурной рекурсией, если подцель хотя бы одного правила в процедуре совпадает с заголовком этого правила.

Существует три правила построения рекурсивных процедур:

1. Первым предложением рекурсивной процедуры должна быть *нерекурсивная фраза*, определяющая вид процедуры *в момент завершения рекурсии*.
2. Рекурсивное правило перед рекурсивным вызовом должно содержать подцель, вырабатывающую *новые значения аргументов*.
3. Рекурсивная подцель, использующая новые значения аргументов, должна стоять *в конце* рекурсивного правила.

Общеизвестное использование рекурсии – классический пример вычисления факториала: **0! = 1, n! = n * (n – 1)!**

Пример 3-3. Вычисление факториала при помощи рекурсии.

Рецепт для *Пролога*:

> Если **N = 0**, то **FN = 1**,
> иначе найти факториал **(N – 1)** и умножить его на **N**.

Порядок поиска решения 4!:

> чтобы найти 4! надо найти 3!
> чтобы найти 3! надо найти 2!
> …
> 0! можно найти без обращения к другим факториалам.
> Если есть 0!, то
> умножить его на 1, чтобы получить 1!
> умножить результат на 2, чтобы получить 2!
> …
> умножить результат на 4, чтобы получить 4!

Рекурсивная процедура вычисления факториала на языке *Пролог*:

```
domains     n = integer        fn = real
predicates  nondeterm fact (n, fn)
clauses     fact (0, 1).                % 1
            fact (N, FN) :–             % 2
                  M = N – 1,            % 3
                  fact ( M, F ),        % 4
                  FN = N * F.           % 5
goal        fact (4, X).
```

Поиск решения для цели:

1. Цель **fact (4, X)** *не согласуется* с % 1.
2. Цель **fact (4, X)** *согласуется* с % 2.
3. Свободная переменная **M** конкретизируется значением **3**.
4. Получается новая цель **fact (3, F)**.
5. Решение целей %1 – %4 будет повторяться до тех пор, пока переменная **M** не конкретизируется значением **0**.
6. Цель **fact (0, F)** согласуется с % 1, в результате чего переменная **F** связывается и конкретизируется значением **1**.
7. Начнёт выполняться цель % 5, и будет получено решение **X = 24**.
   В таблице показано состояние стека при вычислении:

N	F	N * F
1	1	1
2	1	2
3	2	6
4	6	24

При повторном вызове процедуры *Пролог* создаёт новую копию предиката **fact** таким образом, что он становится способным вызвать сам себя как *полностью самостоятельную процедуру*. При этом **код выполнения не копируется**, но **все значения аргументов и промежуточных результатов сохраняются в стеке**, который создаётся при вызове правила. Таким образом, в стеке будут **N, M, F**.

Рассмотренный пример ярко демонстрирует неэффективное использование оперативной памяти, так как при поиске решения необходимо помнить слишком много промежуточных результатов. Следующий пример показывает, как можно решить эту проблему, используя итерацию.

Пример 3-4. Вычисление факториала при помощи итерации.

```
domains
      n = integer        fn = real
predicates
      nondeterm fact (integer, n, real, fn)
```

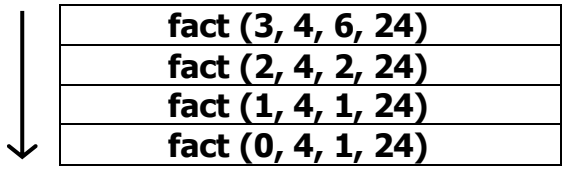
```
clauses     fact (I, N, FI, FN) :–          % 1
                I < N,                      % 1-1
                J = I + 1,                  % 1-2
                F = FI * J,                 % 1-3
                fact (J, N,  F, FN).        % 1-4
            fact (N, N, FN, FN).            % 2
goal        fact (0, 4, 1, X).
```

Поиск решения для цели:

1. Согласование цели **fact (0, 4, 1, X)** по %1происходит, пока подцель **I < N** истинна. Решение подцелей показано в таблице:

% 1-1	% 1-2	% 1-3
I < N	**J = I + 1**	**F = FI * J**
0 < 4	1	1 * 1
1 < 4	2	1 * 2
2 < 4	3	2 * 3
3 < 4	4	6 * 4
4 < 4	*не согласуется*	

2. Альтернатива неудачному согласованию подцели % 1-1 будет найдена в правиле % 2 **fact (N, N, FN, FN)**, потому что на момент вызова первые два аргумента конкретизированы значением **4**. Третий аргумент сцеплен с переменной **F** и конкретизирован значением **24**. Согласно заголовку правила %2 становится возможным связывание последнего аргумента и конкретизация его значением **24**.
3. Начинается возврат из рекурсии, и получается решение **X = 24**:



3.3. Рекурсивные данные

Тип данных является рекурсивным, если он допускает компоненты, содержащие такие же, как и они сами.

3.3.1. Бинарные деревья

К бинарным деревьям относится класс деревьев, где каждый предок имеет не более двух потомков. Каждый переход дерева – самостоятельное дерево, именно поэтому структура рекурсивна.

В *Прологе* бинарное дерево можно задать при помощи *множественного объявления*: необходимо описать **обычное дерево** функтором с тремя аргументами **или пустое дерево** функтором без аргументов:

```
domains
        tree = tree (symbol, tree, tree) ;    % обычное дерево
        nil                                   % пустое дерево
```

, где **symbol** – элемент, находящийся в вершине;
tree – левое и правое поддеревья.

Следует заметить, что представление дерева в программе – это **не предложение** *Пролога*, а всего лишь **структура** – сложный объект.

Процедуры обработки деревьев рассматриваются на примере дерева, показанного на рис. 2.

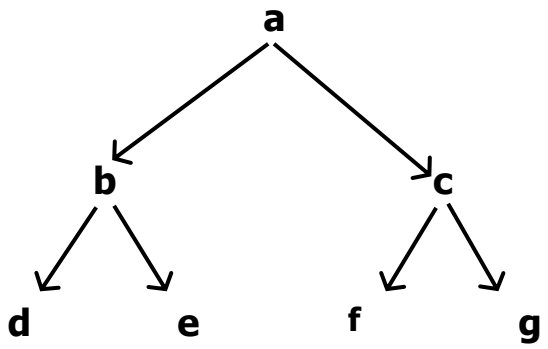


Рис. 2. Пример дерева

Пример 3-5. Проверка принадлежности элемента дереву.

Декларативное понимание: **X** – элемент дерева, если **X** находится в корне, или **X** является элементом левого или правого поддеревьев.

```
domains     tree = tree ( symbol, tree, tree ) ;
            nil
predicates  nondeterm member (symbol, tree)
clauses     member (X, tree (X, _, _)).          % 1
            member (X, tree (_, Left, _)) :-     % 2
                  member (X, Left).
            member (X, tree (_, _, Right)) :-    % 3
                  member (X, Right).
```

Ниже рассматривается проверка разных целевых утверждений.

```
goal        member (d, tree (a,  tree (b, tree (d, nil, nil), tree (e, nil, nil)),
            tree (c,  tree (f, nil, nil), tree (g, nil, nil)))).
```

Поиск решения:

1. % 1 не согласуется **no**.
2. % 2 конкретизация **X = d** и **Left = tree (b, tree (d, nil, nil), tree (e, nil, nil))**.
3. % 1 не согласуется **no**.
4. % 2 конкретизация **X = d** и **Left = tree (d, nil, nil)**.
5. % 1 согласуется и получается ответ **yes**.

```
goal        member (m, tree (a,  tree (b, tree (d, nil, nil), tree (e, nil, nil)),
                    tree (c,  tree (f, nil, nil), tree (g, nil, nil)))).
```

Поиск решения:

1. % 1 не согласуется **no**.
2. % 2 конкретизация **X = m** и **Left = tree (b, tree (d, nil, nil), tree (e, nil, nil))**.
3. % 1 не согласуется **no**.
4. % 2 **X = m** и **Left = tree (d, nil, nil)**.
5. % 1 не согласуется **no**.
6. % 2 **X = m** и **Left = nil**.
7. % 3 **X = m** и **Right = nil**.
8. % 3 **X = m** и **Right = tree (e, nil, nil)**.
9. % 2 **X = m** и **Left = nil**.
10. % 3 **X = m** и **Right = nil**.
11. % 3 **X = m** и **Right = tree (c, tree (f, nil, nil), tree (g, e, e))**.
12. % 2 **X = m** и **Left = tree (f, nil, nil)**.
13. % 2 **X = m** и **Left = nil**.
14. % 3 **X = m** и **Right = nil**.
15. % 3 **X = m** и **Right = tree (g, nil, nil)**.
16. % 2 **X = m** и **Left = nil**.
17. % 3 **X = m** и **Right = nil**.
18. Получается ответ **no**.

Пример 3-6. Обход дерева.

Существует три возможности линейного упорядочения при обходе: сверху вниз, слева направо и снизу вверх.

Декларативное понимание: Если дерево пустое – ничего не делать. Если дерево не пустое, то действовать в соответствии со способом обхода.

Обход сверху вниз: напечатать корень, перейти на левое поддерево, перейти на правое поддерево.

Обход слева направо: перейти на левое поддерево, напечатать корень, перейти на правое поддерево.

Обход снизу вверх: перейти на левое поддерево, перейти на правое поддерево, напечатать корень.

```
domains     treetype = tree (symbol, treetype, treetype);
            nil

predicates
            TopDown (treetype)          % Обход сверху вниз
            LeftRight  (treetype)       % Обход слева направо
            DownTop (treetype)          % Обход снизу вверх
            run                         % Тест

goal        run.
```

```
clauses
        % Обход сверху вниз
        TopDown (nil).
        TopDown (tree (X, Left, Right)) :–
                write (X), write (' '),
                TopDown (Left),
                TopDown (Right).
        % Обход слева направо
        LeftRight (nil).
        LeftRight (tree (X, Left, Rigth)) :–
                LeftRight (Left),
                write (X), write (' '),
                LeftRight (Rigth).
        % Обход снизу вверх
        DownTop (nil).
        DownTop (tree (X, Left, Right)) :–
                DownTop (Left),
                DownTop (Right),
                write (X), write (' ').
        % Тест
        run :–
                TopDown (tree (a, tree (b, tree (d, nil, nil),
                        tree (e, nil, nil)), tree (c, tree (f, nil, nil),
                        tree (g, nil, nil)))), nl,
                LeftRight (tree (a, tree (b, tree (d, nil, nil),
                        tree (e, nil, nil)), tree (c, tree (f, nil, nil),
                        tree (g, nil, nil)))), nl,
                DownTop (tree (a, tree (b, tree (d, nil, nil),
                        tree (e, nil, nil)), tree (c, tree (f, nil, nil),
                        tree (g, nil, nil)))), exit.
```

Результат работы программы:

```
a    b    d    e    c    f    g
d    b    e    a    f    c    g
d    e    b    f    g    c    a
```

Пример 3-7. Вычисление высоты дерева.

Декларативное понимание: Высота пустого дерева равна 0. Высота не пустого дерева определяется как максимальная высота левого и правого поддеревьев плюс 1.

```
domains     tree = tree (symbol, tree, tree) ;
            nil
predicates
            % вычисление высоты дерева
            height ( tree, integer )
            % определение максимума из двух чисел
            max (integer, integer, integer)
```

```
clauses
        % вычисление высоты дерева
        height (nil, 0).
        height (tree ( _, Left, Right ), H) :-
                height (Left, LH),
                height (Right, RH),
                max (LH, RH, MH),
                H = 1 + MH.
        % определение максимума из двух чисел
        max (A, B, A) :- A >= B, !.
        max (_, B, B).
goal    height (tree (a, tree (b, tree (d, nil, nil),   tree (e, nil, nil )),
        tree (c,  tree (f, nil, nil), tree (g, nil, nil))), H),
        write (H),   exit.
```

Описание дерева не должно вызывать негативную реакцию, потому что в *Прологе* деревья строятся программно и обеспечивают хороший способ хранить элементы данных так, чтобы их можно было быстро искать. Дерево, построенное для этой цели, называется *деревом поиска*. С точки зрения пользователя, структура дерева не несёт никакой информации – оно просто более быстрая альтернатива списку.

Бинарный поиск, использующий структуру дерева, в *Прологе* побивает все рекорды скорости *(в пять раз)* подобных программ в традиционных языках программирования.

Дерево в *Прологе* формируется в зависимости от направления упорядочения (по возрастанию или по убыванию). Для упорядочения по возрастанию элементы в левом поддереве предшествуют элементу корня, и в правом поддереве они следуют за ним. При поиске каждый раз проверяется корень. За счёт расположения элементов, можно устранить половину проверок, и поэтому поиск идёт очень быстро.

Алгоритм поиска аналогичен алгоритму создания дерева. Если дерево пусто, то ничего не делать. Иначе проверить все элементы в левом поддереве. Если поисковый элемент не найден, перейти в правое поддерево.

Пример 3-8. Использование дерева для сортировки символов.

```
domains     chartree = tree (char, chartree, chartree);
            end

predicates
        nondeterm do (chartree)                 % меню и операции с деревом
        action (char, chartree, chartree)       %  реализация пунктов меню
        createTree (chartree, chartree)         % создание дерева
        insert (char, chartree, chartree)       % добавление элемента в дерево
        writeTree (chartree)                    % обход дерева слева направо
        nondeterm repeat                        % итерация
```

```
clauses
        % меню и операции с деревом
        do (Tree) :–
                repeat,
                nl, write (" 1 – Модификация дерева"),
                nl, write (" 2 – Просмотр дерева "),
                nl, write (" 0 – Выход из программы "),
                nl, write (" Введите цифру -> "), nl,
                readchar (X),
                action (X, Tree, NewTree),      % Tree – входное дерево
                                                % NewTree – выходное дерево
                do (NewTree).           % повтор операций с новым деревом

        % пункт меню – создание (модификация) дерева
        action ('1', Tree, NewTree) :–
                write ("# конец ввода" ), nl,
                write ("Вводите символы: ”), nl,
                createTree (Tree, NewTree).

        % пункт меню – просмотр дерева
        action ('2', Tree, Tree) :–
                writeTree (Tree),
                nl, write ("Нажмите любую клавишу..."),
                readchar (_).

        % пункт меню – выход из программы
        action ('0', _, end) :– exit.

        % создание (модификация) дерева
        createTree (Tree, NewTree) :–
                readchar (C),       % ввод символа
                C <> '#', !,        % Если символ не равен #,
                write (C, " "),             % то вывод введённого символа
                % добавление введённого символа в TempTree
                insert (C, Tree, TempTree),
                % рекурсивный вызов, NewTree – свободная переменная
                createTree (TempTree, NewTree).
                % второй аргумент принимает значение первого,
                % когда введён символ #
                createTree (Tree, Tree).

        % добавление элемента в дерево
        % описание конца поддерева
        insert (New, end, tree (New, end, end)) :– !.
% Если введённый элемент New < корня Element, то добавление влево
        insert (New, tree (Element, Left, Right),
                tree (Element, NewLeft, Right )) :–
                New < Element , !,
                insert (New, Left, NewLeft).
% Если введённый элемент New ≥ Element, то добавление вправо
        insert (New, tree (Element, Left, Right),
```

```
            tree (Element, Left, NewRight)) :–
            insert (New, Right, NewRight).

      % обход дерева слева направо
      writeTree (end).                          % дерево пустое
      writeTree (tree (Item, Left, Right)) :–   % обход непустого дерева
            writeTree (Left), write (Item, ' '), writeTree (Right).
      % предикат для повторений
      repeat.
      repeat :– repeat.
% целевое утверждение
goal  write ("Сортировка символов"), nl,
      do (end).    % дерево end на входе – пустое!!!
```

Пример 3-9. Использование деревьев для сортировки строк.

Операции чтения и записи переназначены к открытым файлам. Имена входного и выходного файлов запрашиваются в самом начале работы программы. Спецификации файлов должны указываться полностью (устройство, путь, имя и при необходимости расширение). Входной файл может содержать произвольный текст. Строки текста входного файла сортируются программой в алфавитном порядке и записываются в выходной файл.

```
domains
      treetype = tree (string, treetype, treetype) ;
      empty
      file = infile ;
      outfile

predicates
      main                                % цель
      readInput (treetype)                % формирование дерева
      readInputAux (treetype, treetype)   % формирование дерева из empty
      insert (string, treetype, treetype) % размещение строки в дереве
      writeOutput (treetype)              % обход дерева

goal  main.

clauses
      % цель
      main :-
            write ("File to read:  "),
            readln (In), nl,
            openread (infile, In),
            write ("File to write: "),
            readln (Out), nl,
            openwrite (outfile, Out),

            % переназначение всех операций чтения
            readdevice ( infile ),
            readInput (Tree),
```

```
        % переназначение всех операций записи
        writedevice (outfile),
        writeOutput (Tree),
        closefile (infile),
        closefile (outfile).

    % формирование дерева
    readInput (Tree) :-
        readInputAux (empty, Tree).

    % формирование дерева
    readInputAux (Tree, NewTree) :-   % 1
        readln (S), !,                % чтение строки из файла
        insert (S, Tree, Tree1),      % размещение строки в дереве
        readInputAux (Tree1, NewTree).
    % вызывается, когда %1 терпит неудачу при достижении EOF
    readInputAux (Tree, Tree).

% размещение элемента в дереве Tree, дающее новое дерево NewTree
    insert (NewItem, empty, tree(NewItem, empty, empty)) :- !.
    insert (NewItem, tree (Element, Left, Right),
        tree (Element, NewLeft, Right)) :-
        NewItem < Element, !,
        insert (NewItem, Left, NewLeft).
    insert (NewItem, tree (Element, Left, Right),
        tree (Element, Left, NewRight)) :-
        insert (NewItem, Right, NewRight).

    % обход дерева
    writeOutput (empty).
    writeOutput (tree (Item, Left, Right)) :-
        writeOutput (Left), write (Item), nl, writeOutput (Right).
```