Программирование в системе *Mathematica.*

Операторы, функции, процедуры

Основы программирования

- Методы программирования
- Образцы (patterns) и их применение
- Функции пользователя
- Функции FixedPoint и Catch
- Реализация рекурсивных и рекуррентных алгоритмов
- Использование процедур
- Организация циклов

Методы программирования

Такие мощные системы, как Mathematica, предназначены, в основном, для решения математических задач без их программирования большинством пользователей. Однако это вовсе не означает, что Mathematica не является языком (или системой) программирования и не позволяет при необходимости программировать решение простых или сложных задач, для которых имеющихся встроенных функций и даже пакетов расширений оказывается недостаточно или которые требуют для реализации своих алгоритмов применения типовых программных средств, присущих обычным языкам программирования. Все обстоит совсем иначе.

Фактически, основой системы Mathematica является *проблемно-ориентированный* на математические расчеты язык программирования сверхвысокого уровня. По своим возможностям этот язык намного превосходит обычные универсальные языки программирования, такие как Фортран, Бейсик, Паскаль или С.

Важно подчеркнуть, что здесь речь идет о языке *программирования* системы Mathematica, а не о языке реализации самой системы. Языком *реализации* является универсальный язык программирования С++, показавший свою высокую эффективность в качестве языка системного программирования.

Как и всякий язык программирования, входной язык системы Mathematica содержит *операторы, функции* и *управляющие стриктуры.* Основные операторы и функции этого языка и относящиеся к ним опции мы фактически уже рассмотрели. Набор описанных ранее типовых операторов и функций характерен для большинства современных языков программирования. Мощь системы Mathematica как средства программирования решения математических задач обусловлена необычно большим (в сравнении с обычными языками программирования) набором функций, среди которых немало таких, которые реализуют сложные и практически полезные

математические преобразования и современные вычислительные методы (как численные, так и аналитические).

Число этих функций только в ядре и библиотеках приближается к тысяче. Среди них такие операции, как символьное и численное дифференцирование и интегрирование, вычисление пределов функций, вычисление специальных математических функций и т. д. — словом, реализации именно тех средств, для создания которых на обычных языках программирования приходится составлять отдельные, подчас довольно сложные программы. Почти столько же новых функций (или модернизированных старых) содержат пакеты расширения (Add-on Packages).

Язык программирования системы Mathematica трудно отнести к какому-либо конкретному типу. Можно разве что сказать, что он является типичным *интерпретатором* и не предназначен для создания исполняемых файлов. Впрочем, для отдельных выражений этот язык может осуществлять *компиляцию* с помощью функции Compile, что полезно при необходимости увеличения скорости счета.

Этот язык вобрал в себя лучшие средства ряда поколений языков программирования, таких как Бейсик, Фортран, Паскаль и С. Благодаря этому он позволяет легко реализовывать все известные типы (концепции) программирования: функциональное, структурное, объектно-ориентированное, математическое, логическое, рекурсивное и т. д. К примеру, вычисление таких функций, как факториал, в Mathematica можно запрограммировать в виде функции пользователя целым рядом способов:

```
f[n_] =n!

f[n_] =Gamma[n-l]

f [n_] =n*f [n-1] ;f [0]=l;f [1]=1;

f[n_] =Product[i/i,n]

f [n_] =Module[t=l,Do[t=t*i,i,n] ;t]

f [n_] =Module [ { t=l } , For [ i=l , i<=n , i++ , t*=i ] ; t]

f[n_] =Fold [Times,1, Range [n] ]
```

Все их можно проверить с помощью следующего теста:

```
{f[0],f[1],f[5],f[10]}

{1, 1, 120, 3628800}
```

Как отмечалось, внутреннее представление всех вычислений базируется на применении полных форм выражений, представленных функциями. И вообще, функциям в системе Mathematica принадлежит решающая роль. Таким образом, Mathematica. фактически, изначально реализует

функциональный метод программирования — один из самых эффективных и надежных. А обилие логических операторов и функций позволяет полноценно реализовать и *логический* метод программирования. Множество операций преобразования выражений и функций позволяют осуществлять программирование на основе *правил преобразования.*

Надо также отметить, что язык системы позволяет разбивать программы на отдельные модули (блоки) и хранить эти модули в тексте документа или на диске Возможно создание полностью самостоятельных блоков — именованных процедур и функций с локальными переменными. Все это наряду с типовыми управляющими структурами позволяет реализовать *структурное* и *модульное* программирование.

Столь же естественно язык системы реализует *объектно-ориентированное* программирование. Оно базируется прежде всего на обобщенном понятии объекта и возможности создания множества связанных друг с другом объектов. В системе Mathematica каждая ячейка документа является объектом и порождается другими, предшествующими объектами. При этом содержанием объектов могут быть математические выражения, входные и выходные данные, графики и рисунки, звуки и т. д.

С понятием объекта тесно связаны три основных свойства, перечисленные ниже:

- *инкапсуляция* — объединение в одном объекте как данных, так и методов их обработки;
- *наследование* — означает, что каждый объект, производный от других объектов, наследует их свойства;
- *полиформизм* — свойство, позволяющее передать ряду объектов сообщение, которое будет обрабатываться каждым объектом в соответствии с его индивидуальными особенностями.

Приведенный ниже пример объектно-ориентированного программирования дает три определения, ассоциированные с объектом h:

```
h/ : h [x_] +h [y_] : =hplus [x , y]

h/:p[h[x_],x]:=hp[x]

h/:f_[h[x_]] :=fh[f,x]
```

В принципе, язык программирования системы Mathematica специально создан для реализации любого из перечисленных подходов к программированию, а также ряда других — например, *рекуррентного* программирования, при котором очередной шаг вычислений базируется на данных, полученных на предыдущих шагах. Наглядным примером этого может служить вычисление факториала рекуррентным методом. Возможно также создание *рекурсивных* функций (с обращением к самим себе) и, соответственно, использование рекурсивного программирования. Оно, кстати, играет большую роль в осуществлении символьных преобразований.

Средства языка Mathematica позволяют осуществить и *визуально-ориентированное* программирование. Его смысл заключается в автоматической генерации программных модулей путем визуального выбора интуитивно понятного объекта — чаще всего путем щелчка на кнопке. Mathematica позволяет создавать палитры и панели с различными кнопками, позволяющими управлять программой или вводить новые программные объекты. Однако визуально-ориентированное программирование не является основным. В основном оно ориентировано на создание палитр пользователя с нужными ему функциями.

Образцы и их применение

Образцы (patterns) в системе Mathematica служат для задания выражений различных классов и придания переменным особых свойств, необходимых для создания специальных программных конструкций, таких как функции пользователя и процедуры. Это необычайно гибкое и мощное средство обобщенного представления математических выражений, используемое при любом подходе к программированию.

Признаком образца являются знаки подчеркивания «_» (от одного до трех). Они обычно выглядят слитно, так что надо внимательно следить за общей длиной символов образцов. Наиболее распространенное применение образцов — указание на локальный характер переменных при задании функций пользователя. Например, функция

```
fsc[x_,y_]:= x * Sin[y] + y * Cos[x] + z
```

в списке параметров содержит два образца, x_ и y_. В правой части этого выражения переменные x и y, связанные с образцами x_ и y_, становятся *локальными* переменными, тогда как переменная z будет *глобальной* переменной. Обратите особое внимание на то, что символы образцов используются только в списках параметров — в правой части выражений они уже не применяются.

Образцами можно задавать некоторые общие свойства функций. Например, запись

```
f[x_,x_] := p[x]
```

означает, что функция f двух идентичных аргументов становится тождественной функции p [x]. Следовательно, вызов функции

```
f[a,a] + f[a,b]
```

даст выход в виде

```
f[a,b] + p[a]
```

а при вызове

```
f[a^2- 1, a^2- 1]
```

будет получен результат

```
p[-1 + a^2]
```

Примеры применения образцов для задания функции вычисления факториала приводились выше. В образце можно указывать его тип данных:

- x_Integer — образец целочисленный;
- x_Real — образец с действительным значением;
- x_Complex — образец с комплексным значением;
- x_h — образец с заголовком h (от слова *head* — голова).

Задание типов данных с помощью образцов делает программы более строгими и наглядными и позволяет избежать ошибок, связанных с несоответствием типов.

В системе Mathematica используются следующие типы образцов.

Обозначение	**Назначение образца**
-	Любое выражение
x_	Любое выражение, представленное именем x
: : pattern	Образец, представленный именем x
pattern ? test	Возвращает True, когда test применен к значению образца
_h	Любое выражение с заголовком h
x_h	Любое выражение с заголовком h, представленное именем x
-	Любая последовательность с одним и более выражений
-	Любая последовательность с нулем или более выражений
:x_< ИЛИ x__	Последовательности выражений, представленные именем x
_h или h__	Последовательности выражений, каждое с заголовком h
x _ h или x__h	Последовательности выражений с заголовком h, представленные именем x
x_ :v	Выражение с определенным значением v
x_h:v	Выражение с заголовком h и определенным значением v
x_.	Выражение с глобально заданным значением по умолчанию
Optional [x h]	Выражение с заголовком h и с глобально заданным значением по умолчанию
Pattern. .	Образец, повторяемый один или более раз
Pattern. . .	Образец, повторяемый ноль или более раз

Еще раз отметим, что символ «_» в образцах может иметь одинарную, двойную или тройную длину. Надо следить за правильностью его применения, поскольку эти варианты различаются по смыслу. Образцы широко применяются при задании функций пользователя и в пакетах расширения системы.

Функции пользователя

Понятие функции ассоциируется с обязательным возвратом некоторого значения в ответ на обращение к функции по ее имени с указанием аргументов (параметров) в квадратных скобках. Возврат функциями некоторых значений позволяет применять их наряду с операторами для составления математических выражений.

Функции подразделяются на встроенные в ядро системы внутренние функции и функции, заданные пользователем. Примером первых могут быть Sin[x], Bessell [n, x] и т.д. Mathematica содержит множество таких функций, охватывающих практически все широко распространенные элементарные и специальные математические функции. Есть и возможность создания функций со специальными свойствами — *чистых* (pure functions) и *анонимных* функций.

Суть *функционального программирования* заключается в использовании в ходе решения задач только функций. При этом возможно неоднократное вложение функций друг в друга и применение функций различного вида. В ряде случаев, особенно в процессе символьных преобразований, происходит *взаимная рекурсия* множества функций, сопровождаемая почти неограниченным углублением рекурсии и нарастанием сложности обрабатываемых системой выражений.

Встроенные стандартные функции и их типовые применения уже были описаны. Так что далее мы рассмотрим только задание функций особого вида, создаваемых пользователем или используемых в управляющих структурах программ.

Хотя в системах Mathematica имеется около тысячи встроенных функций, любому пользователю рано или поздно может потребоваться создание какой-либо своей функции. Кажется естественным задать ее по правилам, принятым во многих языках программирования. Например, функцию для возведения *x* в степень *n* можно было бы определить так:

```
powerxn[x, n] := x^n
```

Однако такая функция отказывается работать:

```
{powerxn[2, 3], powerxn[a, b]}
```

```
{powerxn[2, 3] , powerxn[a, b]}
```

Причина этого кроется в том, что в системе Mathematica символы x и n являются обычными символами, не наделенными особыми свойствами. Будучи использованными в качестве параметров функции, они не способны воспринимать формальные параметры [2,3] или [a, b]. Так что вычислить нашу ущербную функцию можно лишь при предварительном присваивании x и n нужных значений:

```
x := 2; n := 3; powerxn[x, n]
```

```
8
```

Разумеется, заданная таким образом функция является неполноценной. Для того чтобы функция пользователя нормально воспринимала переданные ей аргументы, в списке параметров надо использовать *образцы* в виде переменных, но имеющие после своих имен символы подчеркивания. Образцы способны быть *формальными параметрами* функций и воспринимать значения *фактических параметров* (в нашем случае значений 2 и 3). Таким образом, правильной будет запись функции пользователя в виде

```
powerxn[x_, n_] := x^n
```

Теперь вычисление по заданной функции пользователя пройдет гладко, причем как в численном, так и в символьном виде:

```
{powerxn[2, 3], powerxn[z, y], powerxn[x, n]}
```

```
{8, z^y, 8}
```

Заметим, что для уничтожения определения заданной функции можно использовать команду-функцию Clear [Name_f unction], где Name_f unction — имя функции.

Можно также задать функцию пользователя, содержащую несколько выражений, заключив их в круглые скобки:

```
f[x_] := (t = (1 + x)^2; t = Expand[t])
```

Переменные списка параметров, после имен которых стоит знак «_», являются локальными в теле функции или процедуры. На их место подставляются фактические значения соответствующих параметров, например:

```
f [a + b]
```

```
1+2a+a^2+2b+2ab+b^2
```

```
t
```

```
1+2a+a^2+2b+2ab+b^2
```

Обратите внимание на то, что переменная t в функции f является глобальной. Это поясняет результат последней операции. Применение глобальных переменных . в теле функции вполне возможно, но создает так называемый *побочный эффект* — в данном случае меняет значение глобальной переменной t. Для устранения побочных эффектов надо использовать образцы и другие специальные способы задания функций, описанные ниже. Итак, можно сформулировать ряд правил для задания функций пользователя:

- такая функция имеет идентификатор — имя, которое должно быть уникальным и достаточно понятным;
- в списке параметров функции, размещенном в квадратных скобках после идентификатора, должны использоваться образцы переменных, а не просто переменные;
- может использоваться отложенное (: =) или немедленное (=) присваивание;
- тело функции может содержать несколько выражений, заключенных в круглые скобки, при этом возвращается значение последнего выражения;
- переменные образцов в списке параметров являются локальными и действуют только в пределах тела функции;
- в теле функции могут использоваться глобальные переменные, но при этом возможны побочные эффекты;
- возможно обращение к функции из тела этой же функции (рекурсия).

Параметрами функций могут быть списки при условии допустимости их комбинации. Например, допустимо задать x списком, an — переменной или числом:

```
powerxn[{1, 2, 3, 4, 5}, z]
```

```
{1, 2^z, 3^z, 4^z, 5^z}
```

```
powerxn[{1, 2, 3, 4, 5}, 2]
```

```
{1, 4, 9, 16, 25}
```

После своего задания функции пользователя могут использоваться по тем же правилам, что и встроенные функции.

Суперпозиция функций

При функциональном программировании часто используется *суперпозиция* функций. Для ее реализации используются следующие функции:

- Nest [expr, x, n] — n раз применяет выражение (функцию) expr к заданному аргументу *x,*
- NestList [f, x, n] — возвращает список результатов (n+1)-кратного применения функции f к заданному аргументу x;
- Fold[f, x, list] — дает последний элемент в FoldList [f, x, list];
- FoldList [f, x, {a,b,...}] — возвращает список {x,f [x,a],f [f [x,a],b],...};
- ComposeList [{ f , f ,...}, x] — генерирует список в форме {x,a[x] ,a[a[x]],...}.

Функции Fixed Point и Catch

В функциональном программировании вместо циклов, описываемых далее, может использоваться следующая функция:

- FixedPoint [f, expr] — вычисляет expr и применяет к нему f, пока результат не перестанет изменяться;

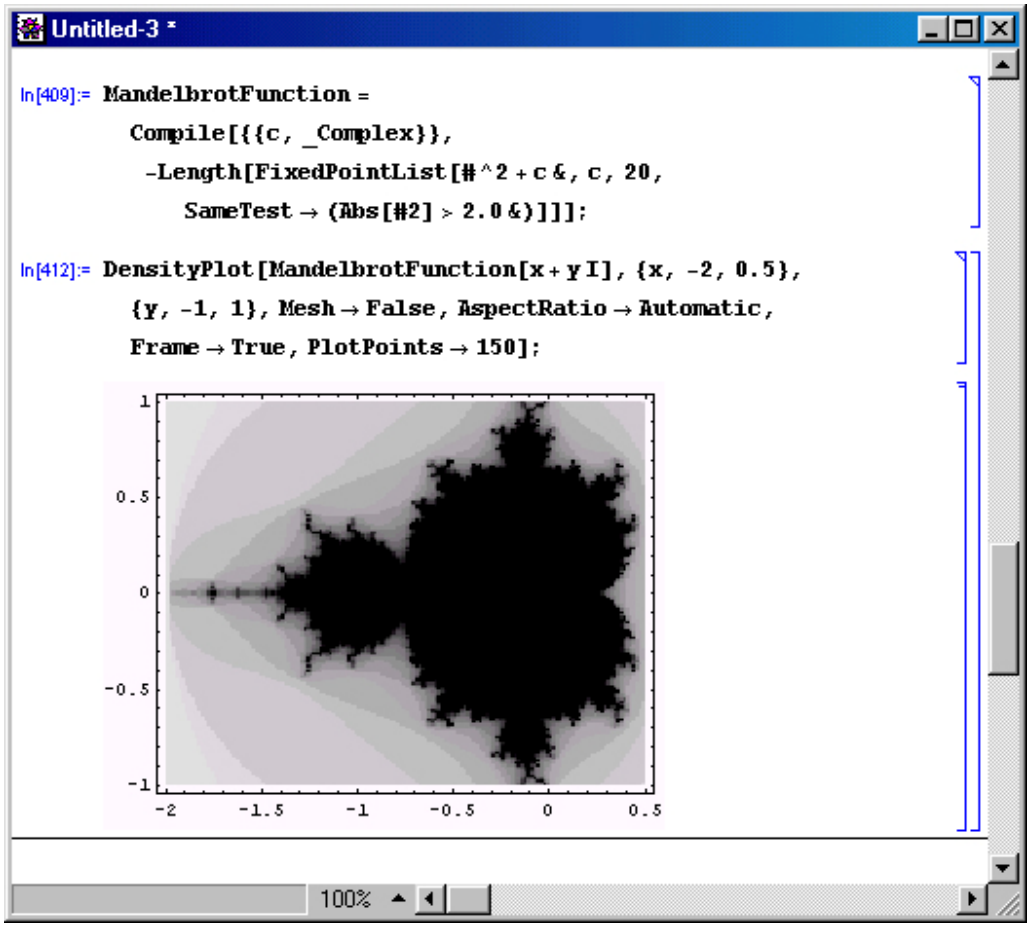
- FixedPoint [f, expr, SameTest->comp] — вычисляет expr и применяет к нему f, пока два последовательных результата не дадут True в тесте SameTest.

Еще одна функция такого рода — это Catch:

- Catch [expr] — вычисляет expr, пока не встретится Throw [value], затем возвращает value;
- Catch [expr, form] — вычисляет expr, пока не встретится Throw [value, tag], затем возвращает value;
- Catch [expr, form, f] — возвращает f [value, tag] вместо value.

Пример программирования графической задачи

В качестве такого примера рассмотрим задачу на построение сложного графика функции Мандельброта. Пример задания соответствующей функции MandelbrotFunction и применения графической функции DensityPlot для наглядного визуального представления функции MandelbrotFunction на комплексной плоскости представлен на рисунке 4.1.



Риунок 4.1 - *Пример задания функции MandelbrotFunction и построения ее графика плотности*

Использование процедур

В основе процедурного программирования лежит понятие *процедуры* и типовых средств управления — циклов, условных и безусловных выражений и т. д. Процедурный подход — самый распространенный в программировании, и разработчики Mathematica были вынуждены обеспечить его полную поддержку. Однако программирование систем Mathematica и в этом случае остается функциональным, поскольку элементы процедурного программирования существуют в конечном счете в виде функций.

Процедуры являются полностью самостоятельными программными модулями, которые задаются своими именами и отождествляются с выполнением некоторой последовательности операций. Они могут быть заданы в одной строке с использованием в качестве разделителя символа «;» (точка с запятой). Вот пример задания однострочной процедуры, отождествленной с именем r:

```
r = (1 + x)^2; r= Expand[r]; r - 1
```

$2X+X^2$

Обратите внимание на то, что в теле процедуры символ r используется как вспомогательная переменная. Эта процедура возвращает символьное выражение

```
Expand[ (1+x)^2] - 1.
```

В общем случае в теле процедуры могут находиться произвольные выражения, разумеется, с синтаксисом, присущим языку программирования системы. Процедура может не возвращать никаких значений, а просто выполнять определенный комплекс операций. Область записи подобных элементарных процедур ограничена ячейкой (строкой) ввода.

Для задания процедуры со списком локальных переменных {a, b,...} и телом proc может использоваться функция Module [{a, b,...} ,proc]. С применением этой функции мы столкнемся позже.

Для создания полноценных процедур и функций, которые могут располагаться в любом числе строк, может использоваться базовая структура — *блок:*

- Block [{x, y,...}, procedure] — задание процедуры с декларацией списка локальных переменных x, y,...;
- Block[{x = x0, y=y0,...}, procedure] — задание процедуры с декларацией списка переменных x, y,... с заданными начальными значениями.

Пример использования базовой структуры:

```
g[x_] := Block[{u}, u = (1 + x)^2; u = Expand[u] ] g[a + b]
```

$1+2a+a^2+2b+2ab+b^2$

```
u
```

```
u
```

```
u = 123456; g[2]
```

```
9
```

```
u
```

```
123456
```

Обратите внимание: последние действия показывают, что переменная и, введенная в тело базовой структуры, является действительно локальной переменной, и присвоение ей символьного выражения (1 + x)^2 в теле блока игнорируется вне этого блока. Если переменная и до применения в функции была не определена, то она так и остается неопределенной. А если она имела до этого некоторое значение (в нашем случае — 123 456), то и по выходе из процедуры она будет иметь это значение.

Организация циклов

Многие задачи в системе Mathematica решаются с использованием линейных алгоритмов и программ. Они могут быть представлены непрерывной цепочкой выражений, выполняемых последовательно от начала до конца.

Однако в большинстве случаев серьезные вычисления базируются на использовании циклических и разветвленных алгоритмов и программ. При этом, в зависимости от промежуточных или исходных данных, вычисления могут идти по разным ветвям программы, циклически повторяться и т. д. Для реализации разветвленных программ язык программирования должен содержать *управляющие структуры,* то есть специальные конструкции языка, реализующие в программах ветвление. Они используются при различных методах программирования, в том числе при процедурном и функциональном программировании.

Циклы типа Do

К важнейшим управляющим структурам в языках программирования относятся *циклы.* С их помощью осуществляется циклическое исполнение некоторого выражения expr заданное число раз. Это число нередко определяется значением некоторой управляющей переменной (например, i, j и т. д.), меняющейся либо с шагом +1, либо от начального значения imin до конечного значения imax с шагом di. Циклы могут быть одинарными или множественными — вложенными друг в друга. Последние используют ряд управляющих переменных. Такого рода циклы организуются с помощью функции Do: O Do [expr, {imax}] — выполняет imax раз вычисление expr; O Do [expr, {i, imax}] — вычисляет expr с переменной i, последовательно принимающей значения от 1 до imax (с шагом 1);

- Do [expr, {i, imin, imax}]—вычисляет expr с переменной i, последовательно принимающей значения от imin до imax с шагом 1;
- Do [expr, {i, imin, imax, di}] — вычисляет expr с переменной i, последовательно принимающей значения от 1 до imax с шагом di;
- Do [expr, {i, imin, imax}, {j, jmin, j max},...] — вычисляет expr, организуя ряд вложенных циклов с управляющими переменными j, i и т. д.

Следующий пример показывает применение цикла Do для задания функции, вычисляющей *n-е* число Фибоначчи:

```
fibonacci[(n_Integer)?Positive] :=

Module[fnl = 1, fn2 =0,

Do[fnl, fn2 = fnl + fn2, fnl, n- 1] ; fnl]

fibonacci[10]

55

fibonacci[100]

354224848179261915075

fibonacci[-10]

fibonacci[-10]
```

Обратите внимание на применение в этом примере функции Module. Она создает программный модуль с локальными переменными (в нашем случае fnl и fn2), в котором организовано рекуррентное вычисление чисел Фибоначчи.

Циклы типа For

Другой вид цикла — цикл For — реализуется одноименной функцией:

```
For[start, test, incr, body]
```

В ней сначала один раз вычисляется выражение start, а затем поочередно вычисляются выражения body и incr до тех пор, пока условие test не перестанет давать логическое значение True. Когда это случится, то есть когда test даст False, цикл заканчивается.

Следующий пример показывает создание простой программы с циклом For и результат ее выполнения:

```
Print["i x"]
```

```
For [x=0; i=0, i < 4, i++

[x += 5*i, Print[i, " ", x]]]

i x

15 ,

2 15

3 30

4 50

Return[x]

Return[50]
```

Программа, приведенная выше, позволяет наблюдать за изменением значений управляющей переменной цикла i и переменной x, получающей за каждый цикл приращение, равное 5*i. В конце документа показан пример на использование функции возврата значений Return [x]. В цикле For не предусмотрено задание локальных переменных, так что надо следить за назначением переменных — при использовании глобальных переменных неизбежны побочные эффекты.

Циклы типа While

Итак, функция For позволяет создавать циклы, которые завершаются при выполнении (эволюции) какого-либо условия. Такие циклы можно организовать и с помощью функции While [test, expr], которая выполняет expr до тех пор, пока test не перестанет давать логическое значение True.

Ниже дан практический пример организации и использования цикла While:

```
i := 1; x := 1; Print["i x"] ;

While[i < 5, i += 1; x += 2*i; Print[i, " ", N[x]]]

i x

2 5.

3 11.

4 19.

5 29.
```

```
Return[x]

Return[29]
```

Циклы типа While, в принципе, могут заменить другие, рассмотренные выше, типы циклов. Однако это усложняет запись и понимание программ. Аппарат локальных переменных в этом типе циклов не используется.

Директивы-функции прерывания и продолжения циклов

В указанных типах циклов и в иных управляющих структурах можно использовать следующие директивы-функции:

- Abort [ ] — вызывает прекращение вычислений с сообщением $ Aborted;
- Break [ ] — выполняет выход из тела цикла или уровня вложенности программы, содержащего данный оператор (циклы типа Do, For и While или тело оператора-переключателя Switch). Оператор возвращает Null-значение (без генерации секции выхода);
- Continue [ ] — задает переход на следующий шаг текущего цикла Do, For или While;
- Interrupt [ ] — прерывает вычисления с возможностью их возобновления;
- Return [ ] — прерывает выполнение с возвратом значения Null;
- Return [expr] — прерывает выполнение с выводом значения выражения expr;
- Throw [value] — задает прекращение выполнения цикла Catch, если в ходе эволюции expr встречается значение value (см. примеры выше).

Условные выражения и безусловные переходы

Для подготовки полноценных программ помимо средств организации циклов необходимы и средства для создания разветвляющихся программ произвольной структуры. Обычно они реализуются с помощью условных выражений, позволяющих в зависимости от выполнения или невыполнения некоторого *условия* (condition) выполнять те или иные фрагменты программ.

Функция IF

Как у большинства языков программирования, условные выражения задаются с помощью оператора или функции IF. Система Mathematica имеет функцию If, формы которой представлены ниже:

- If [condition, t, f] — возвращает t, если результатом вычисления condition является True, и f, если результат равен False;
- If [condition, t, f, u]—то же, но дает и, если в результате вычисления condition не было получено ни True, ни False.

Следующий пример показывает создание программной процедуры с циклом Do, выход из которой реализуется с помощью функции I f и директивы прерывания Aborted!]:

```
x := 1; Print["i x"];
```

```
Do[{If [i == 5, Abort[], None],
i += 1; x += 2*i; Print[i, " ", N[x]]},
{i, 1, 100}]
i x
2 5
3 11.
4 19.
5 29.
$Aborted
Return[x]
Return[1]
```

Тот же пример, но с применением директивы выхода из цикла Break [] в функции If показан ниже:

```
x := 1; Print["i x"];
Do[{If [i == 5, Break[], None],
i += 1; x += 2*i; Print[i, " ", N[x]]},
{i, 1, 100}]
i x
2 5.
3 11.
4 19.
5 29.
Return[x]
Return[29]
```

В данном случае никаких специальных сообщений о выходе из цикла не выдается. Функция If обеспечивает ветвление максимум по двум ветвям программы. Для ветвления по многим направлениям можно использовать древовидные структуры программ с множеством функций If. Однако это усложняет исходный текст программы.

Функции-переключатели

Для организации ветвления по многим направлениям в современных языках программирования используются операторы-переключатели. В системе Mathematica множественное ветвление организовано с помощью функций Which и Switch:

- Which [testl, valuel, test2, value2,...] — вычисляет в порядке следования каждый из testi, сразу возвращая именно ту величину из valuei, которая относится к первому testi, давшему True;
- Switch [expr, forml, valuel, form2, value2,...] — вычисляет селектор expr, затем сравнивает его последовательно с каждой из меток f ormi, вычисляя и возвращая то valuei, которое соответствует первому совпадению.

Приведем примеры работы функции which:

```
Whichtl == 2,1,2== 2, 2, 3 == 3, 3]

2

Which[l == 2, x, 2 == 2, y, 3 == 3, z]

y
```

Следующие примеры иллюстрируют работу функции Switch:

```
Switch[1, 1, a, 2, b, 3, c]

a

Switch[2, 1, a, 2, b, 3, c]

b

Switch[3, 1, a, 2, b, 3, c]

c

Switch[8, 1, a, 2, b, 3, c]

Switch[8,

1, a,
```

```
2, b,

3, c]
```

Обратите внимание на последний пример — при неверном задании первого параметра (селектора) просто повторяется запись функции.

Следующий пример показывает возможность выбора с применением вещественных значений селектора и меток:

```
Switch[8., 1.5, a, 2.5, b, 8., c]

c

Switch[1.5, 1.5, a, 2.5, b, 8., c]

a

Switch[8, 1.5, a, 2.5, b, 8., c]

Switch[8,

1.5, a,

2.5, b,

8., c]
```

Опять-таки, обратите внимание на последний пример — здесь использован селектор в виде *целого* числа 8, тогда как метка выбора — *вещественное* число 8. Выбор при этом не происходит, поскольку целочисленное значение 8 не является тождественным вещественной восьмерке.