Преобразование типов. Сессии. Калькулятор.

Лабораторная работа № В-2.

ЦЕЛЬ РАБОТЫ

Изучение правил и особенностей *PHP* при преобразовании типов данных в строчных и численных переменных. Реализация в *PHP* механизма сессий.

Механизм сессий – это очень удобный способ хранения промежуточных данных при работе *PHP*-программы. Действительно, очень часто возникают ситуации, когда результат выполнения программы необходимо использовать в следующий раз или даже на других страницах сайта. В предыдущих лабораторных работах мы уже научились делать это, передавая эти данные через *GET*-параметры. Но, во-первых, это нельзя сделать для большого объема данных, во-вторых, эти данные все время видны посетителю сайта: а если мы, например, создаем систему тестирования или другой схожий сервис, где эти данные должны быть скрыты, то такой способ абсолютно не допустим.

Конечно, вполне возможно использовать для хранения данных файловую систему. В локальных приложениях это волне допустимо: программа создает файл, в котором хранит промежуточные результаты и при необходимости читает и изменяет их. Но на веб-сервере одна и та же страница может одновременно загружаться десятками, сотнями и даже тысячами пользователей – если все данные хранить в одном файле, то они неизбежно перепутаются. Использовать для каждого пользователя свой временный файл, определяя его по *ip*-адресу – уже лучше, но у нескольких пользователей вполне может быть один и тот же *ip*-адрес. Кроме того, необходим способ надежно определять, когда пользователь закончил работу с сайтом, и следующая загрузка страницы должна очистить данные и начать работу заново. А когда просто завершилась загрузка одной страницы: в *PHP* программа полностью заканчивает свою работу после загрузки страницы, на которой она размещена.

Все эти проблемы очень элегантно решает механизм сессий. Для программиста сессия предоставляет собой массив, в который он может записывать любые элементы и который доступен для всех PHP-программ на любой странице сайта до тех пор, пока посетитель не закрыл браузер. Это позволяет не только не заботится об идентификации пользователя и необходимости очистки данных, но и не думать об способах хранения данных в файле.

Для этого применяется следующий подход. При каждой загрузке страницы на сервер передается небольшой параметр, в котором хранится так называемый идентификатор сессии. Именно по нему *PHP* определяет из какого хранилища брать данные. Причем, если в браузере доступно использование *cookies* – то идентификатор хранится там и передается на сервер именно через них. Если же нет – то *PHP* автоматически и самостоятельно добавляет для каждой ссылки это идентификатор в качестве *GET*-параметра. Тогда, до тех пор, пока пользователь не закрыл браузер, его идентификатор сессии не изменится, а значит он все время будет работать с одним и тем же хранилищем данных. При этом идентификатор никак не связан с *ip*-адресом: он случайным образом генерируется сервером если механизм сессий был включен, но идентификатор передан не был.

С помощью сессий в *PHP*-программировании реализуется большое количество функциональных возможностей. Например, при необходимости сделать доступным часть страниц сайта только после аутентификации, информация об данных пользователя и о возможности просматривать им эти страницы удобно хранить именно в сессии.

ПРОДОЛЖИТЕЛЬНОСТЬ

4 академических часа работы в аудитории, 4 академических часа – самостоятельно.

РЕЗУЛЬТАТ РАБОТЫ

Размещенные на Веб-сервере и доступные по протоколу *HTTP* документ, представляющий из себя арифметический калькулятор для целых чисел и десятичных дробей.

ДОПОЛНИТЕЛЬНЫЕ ТРАБОВАНИЯ К РАБОТЕ

Работа оформляется в виде одного *HTML*-документа с интегрированным *PHP*-кодом. При открытии страницы в браузере отображается *POST*-форма со следующими элементами:

- строковое поле для ввода вычисляемого выражения;
- кнопка «Вычислить».

Выражение может содержать целые числа, знаки арифметических операций (плюс "+", минус "-", умножить "*", разделить "/" или ":"), скобки "(" и ")". Посторонние символы, а также нарушение правил математики для арифметических выражений должны приводить к выводу сообщения об ошибке. При нажатии кнопки «*Вычислить*» на сформированной странице перед формой должен быть выведен результат вычисления выражения (либо сообщение об ошибке). Для вычислений запрещается использовать любые функции и методы *PHP* для разбора строк и преобразования (определения) типов данных, кроме приведенных в рекомендациях к структуре программы.

В подвале сайта должна построчно отображаться история вычислений: выражение и полученный результат (в том числе текст ошибки, если он произошла при анализе выражения). Текущий полученный результат вычислений не должен отображаться в истории, но должен попадать туда при следующем обновлении страницы.

РЕКОМЕНДАЦИИ К СТРУКТУРЕ ПРОГРАММЫ

Самостоятельно создайте требуемую форму для ввода выражения, пусть для определенности имя поля на ней будет "`val`". Теперь разделим задачу на два этапа:

- разработка функции `calculate()` для подсчета значения выражения без скобок;
- разработка функции `calculateSq()` для подсчета значения выражения со скобками на основе уже разработанной функции `calculate()`.

Обязательно доведите первый этап до конца прежде чем приступить ко второму: это значительно упростит понимание работы программы и облегчит отладку. Без этого велика вероятность запутаться во взаимных вызовах функций и не успеть выполнить работу в срок.

ВЫЧИСЛЕНИЕ ВЫРАЖЕНИЯ БЕЗ СКОБОК

Тогда для вычисления выражения разработаем две пользовательские функции: `calculate()`, которая интерпретирует переданную в нее в качестве параметра строку в виде выражения и вычисляет его значение, и вспомогательную функцию `isnum()`, определяющую является ли переданный в нее строковый параметр на самом деле числом. Т.е. если в нее передана строка "`23450`" – то это число и функция должна вернуть `TRUE`; если же передана строка "`46в62`" или даже "`048`" – то это не число, и функция возвращает `FALSE`.

Листинг В-2. 1

```
if( isset($_POST['val'] )       // если передан POST-параметр val
{
     $res= calculate( $_POST['val'] ); // вычисляем результат выражения
     if( isnum($res) )     // если полученный результат является числом
          echo 'Значение выражения: '.$res;    // вывод значения
     else            // если результат не число - значит ошибка!
          echo 'Ошибка вычисления выражения: '.$res;      // вывод ошибки
}
```

Тогда, если две таких функции определены, то обработчик формы будет выглядеть довольно просто. В программе проверяется наличие переданного с помощью метода *POST* значения параметра "`val`". Если он передан (а все переданные этим методом параметры хранятся в

массиве `$_POST`), то необходимо вычислить его значение. Для этого используется вызов функции `calculate()` и сохранение результата ее работы в переменной `$res`.

Т.к. эту функцию реализуем тоже мы, то запланируем, что в случае ошибки в выражении она будет возвращать текстовое описание ошибки. Тогда для определения успешности вычисления необходимо всего лишь проверить тип возвращаемого значения: если это число – значит оно выводится как результат вычисления; если не число – как сообщение об ошибке. Теперь остается только реализовать эти две функции: рассмотрим сначала функцию `isnum()`.

Листинг В-2. 2

```
function isnum( $x )
{
      // перебираем все символы строки в цикле
      for($i=0; $i<strlen($x); $i++)
            // если в проверяемой строке недопустимый в числе символ
            if( $x[$i]!='0' && $x[$i]!='1' && $x[$i]!='2' && $x[$i]!='3' &&
                $x[$i]!='4' && $x[$i]!='5' && $x[$i]!='6' && $x[$i]!='7' &&
                   $x[$i]!='8' && $x[$i]!='9' && $x[$i]!='.' )
                      return false;  // возвращаем FALSE
      return true; // если все символы строки допустимы - возвращаем TRUE
}
```

Самое очевидное – это перебрать все символы в строке и если хоть один из них окажется не цифрой, то и строка – не число. Поэтому в теле функции размещается цикл, в котором последовательно для каждого *i*-ого символа строки проверяется: если символ отличается от цифры и точки – то возвращается `false`. Если после проверки всех чисел функция продолжает работу – это значит, что все символы строки цифры или точки. К сожалению, такой проверки недостаточно: если в строке будет несколько точек или она будет начинаться с нескольких нулей, то функция все равно вернет `true`. Для и пустая строка тоже окажется числом. Модифицируем функцию: для этого дополнительно проверим, не пустая ли строка передана, посчитаем число точек в строке (их не может быть больше одной), проверим, не начинается ли строка с точки или нуля, не заканчивается ли точкой.

Листинг В-2. 3

```
function isnum( $x )
{
      if( !$x ) return false;              // если строка пустая - это НЕ число!
       if( $x[0]=='.' || $x[0] == '0' )    // число не может начинаться с точки или
       нуля
             return false;
       if( $x[ strlen($x)-1 ] == '.' )     // число не может заканчиваться на точку
             return false;
      // перебираем все символы строки в цикле
      for($i=0, $point_count=false; $i<strlen($x); $i++)
      {
            // если в проверяемой строке недопустимый в числе символ
            if( $x[$i]!='0' && $x[$i]!='1' && $x[$i]!='2' && $x[$i]!='3' &&
                $x[$i]!='4' && $x[$i]!='5' && $x[$i]!='6' && $x[$i]!='7' &&
                   $x[$i]!='8' && $x[$i]!='9' && $x[$i]!='.' )
                      return false;          // недопустимые символы в строке
            if($x[$i]=='.')       // если в строке встретилась точка
            {
                  if( $point_count )         // если точка уже встречалась
                        return false;   // то это не число
                  else                  // если это первая точка в строке
                        $point_count=true;         // запоминаем это
            }
      }
      return true; // все проверки пройдены - это число
}
```

В начале функции идет проверка корректности первого и последнего символа строки – если в первом ноль или точка, или же в последнем символе точка – то это не число. Далее организован

аналогичный предыдущей версии функции цикл. Однако в нем помимо сравнения каждого символа с цифрой или числом производится проверка количества точек в строке. Если их более двух – то это не число.

Для этого в начале цикла инициализируется переменная `$point_count`, ее значение "`true`" будет означать что в строке в процессе перебора символов уже встречалась точка. Именно это мы и делаем: в этом же цикле добавляется еще один условный оператор, в котором проверяется не точка ли текущий символ. Если да, то если значение переменной `$point_count` "`true`", то это означает что точка уже встречалась ранее, а значит строка – не число. Если же нет – то этой переменной присваивается значение "`true`": на следующих итерациях цикла мы будем знать, что символ уже встречался.

Теперь, завершив подготовку, необходимо реализовать функцию `calculate()`, которая и будет собственно вычислять значение выражения. Начнем ее разработку с одного математического действия: сложения.

Листинг В-2. 4

```
function calculate( $val )
{
      if( !$val ) return 'Выражение не задано!'; // если строка пуста – ошибка

       // разбиваем строку на аргументы и заносим их в массив
      $args = explode('+', $val);
       $sum=0;                          // начальное значение суммы аргументов
      for($i=0; $i<count( $args); $i++)    // перебираем все слагаемые
      {
            // если слагаемое не число – прекращаем работу и возвращаем ошибку
            if( !isnum($arg[$i]) )
                        return 'Неправильная форма числа!';
            $sum += $arg[$i];          // суммируем слагаемое с предыдущими
      }
       return $sum; // если все слагаемые числа – возвращаем сумму
}
```

Функция начинает работу с проверки переданного выражения – если это пустая строка, то и делать ничего не надо: сразу возвращаем ошибку. Если выражение передано, оно разбивается на подстроки (разделитель – символ «+»). Фактически такое разбиение формирует массив со слагаемыми: остается только суммировать их для получения требуемого результата. Это и происходит в цикле по всем элементам массива: для каждого слагаемого осуществляется проверка – число слагаемое, или нет. Если нет – работа функции прекращается с возвратом сообщения об ошибке. Если да – то слагаемое прибавляется к текущей сумме, которая и возвращается в виде результата при успешном сложении всех слагаемых. Обратите внимание: любой не являющийся числом аргумент приведет к возврату ошибки: даже строка "*4++4*" приведет к сообщению об ошибке, т.к. второй аргумент – пустая строка, а она не является числом.

Однако согласно заданию к лабораторной работе, программа должна выполнять все четыре математических действия. Поэтому н следующем шаге доработаем функцию таким образом, чтобы она выполняла и сложение, и умножение.

Листинг В-2. 5

```
function calculate( $val )
{
      if( !$val ) return 'Выражение не задано!'; // если строка пуста – ошибка

      if( isnum($val) ) return $val;  // если выражение число – возвращаем его

       // разбиваем строку на аргументы и заносим их в массив
      $args = explode('+', $val);

      if( count($args)>1 ) // если в выражении есть символы «+»
      {
            $sum=0;  // начальное значение суммы аргументов
```

```
            for($i=0; $i<count($args); $i++)     // перебираем все слагаемые
            {
                  $arg = calculate( $args[$i] );  // вычисляем значение слагаемого
                  if( !isnum($arg) )                    // если результат не число
                              return $arg;                    // возвращаем ошибку
                  $sum += $arg;        // суммируем слагаемое с предыдущими
            }
            return $sum;          // если все слагаемые числа - возвращаем сумму
      }

        // разбиваем строку на множители и заносим их в массив
      $args = explode('*', $val);
      if( count($args)>1 ) // если в выражении есть символы «*»
      {
            $sup=1;  // начальное значение произведения аргументов
            for($i=0; $i<count($args); $i++)     // перебираем все множители
            {
                  $arg = $args[$i];               // текущий множитель

                      // проверяем - если множитель не число -возвращаем ошибку
                  if( !isnum($arg) ) return 'Неправильная форма числа!';
                  $sup *= $arg;    // умножаем множитель с предыдущими
            }
            return $sup;    // если все множители числа - возвращаем произведение
      }
        // выражение - не число, но и символов «+» или в нем нет
      return 'Недопустимые символы в выражении';
}
```

Для понимания принципа работы этой функции необходимо вспомнить о рекурсии в программировании: это вызов в функции самой себя. Ясно, что если делать это просто так – то образуется бесконечный цикл вызовов. Поэтому крайне важно правильно определить базу рекурсии: условий при которых функция заканчивает свою работу без рекурсивного вызова самой себя. В нашем случае в базе к попытке вычисления пустой строки добавляется вычисление состоящего из одного числа выражения. Действительно, если выражение просто число – то дальнейшее вычисления не нужны: достаточно просто вернуть его как результат работы функции.

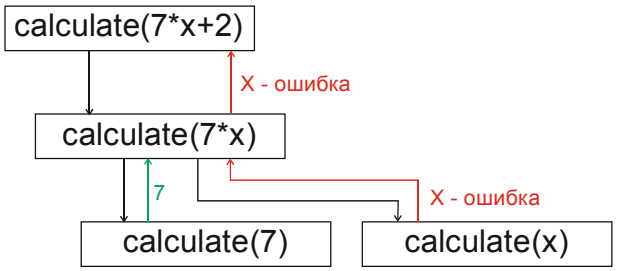
Далее идет разбивка выражение на слагаемые. Если их больше одного, то происходит их суммирование. Но, в отличии предыдущей функции, каждое слагаемое может содержать несколько множителей. Например, выражение "*2+3*4+7*9*" будет разбито на три слагаемых: "*2*", "*3*4*" и "*7*9*". Просто так сложить их мы не можем, поэтому необходимо вычислить каждое из них, что и делается рекурсивным вызовом функции: вычисляем выражения "*2*", "*3*4*" и "*7*9*". Если дальнейшая реализация функции правильная и в результате вычисления мы получим "*2*", "*12*" и "*63*", то окончательный результат вычисления будет равен их сумме. При этом если вычисление значения какого-либо из слагаемых не является числом, т.е. привело к ошибке, то все дальнейшие вычисления прекращаются, а полученная ошибка сразу возвращается. Если же слагаемое только одно – это значит, что возможно в функцию было передано произведение и его необходимо вычислить, поэтому их суммирование и перебор не нужны.

Обратите внимание, что если убрать проверку количества слагаемых, то функция будет работать бесконечно долго. Например, при попытке вычислить значение "*2*3*" мы получим одно слагаемое, которое равно собственно выражению "*2*3*". В цикле мы пытаемся вычислить его значение, рекурсивно вызывая функцию. Ясно, что вызов функции с этим же параметром приведет к такому же результату, т.е. опять-таки вызову функции с тем же самым параметром "*2*3*". И так до бесконечности. Если же перед перебором слагаемых проверять их количество (`if(count($args)>1 )…`), то если их окажется меньше двух бесконечный вызов функций не случится.

Итак, если выражение не просто число и не содержит знаков "*+*", то возможно это произведение. Тогда, по аналогии со слагаемыми, разбиваем выражение на множители. В принципе, обработка произведения аналогична суммированию – только лишь начальное произведение множителей

равно "1", а не "0". Кроме того, рекурсивный вызов из блока произведений не производится, т.к. это конечный результат вычислений.

Если же и множитель в выражении только один, то это означает что в нем нет знаков "*+*", нет знаков "*" и это не число – делаем вывод, что выражение ошибочно, о чем и сообщаем, возвращая соответствующее выражение. Вообще такая реализация задачи хороша тем, что возникшая на любом этапе вычислений ошибка сразу же останавливает дальнейшую работу и будет возвращена в качестве результата самого первого вызова функции.



Например, при вызове функции для выражения "*7*x+2*" из нее будет вызвана функция с аргументом "*7*x*" (попытка вычислить первое слагаемое). Из этой функции – сначала функция с аргументом "*7*" (попытка вычислить первый множитель, вернет 7), а затем – с аргументом "*x*" (второй множитель). Этот вызов вернет ошибку, которая тут же передастся на верхний уровень: функция не будет продолжать вычислять произведение, а тут же вернет текст ошибки. Первая вызванная функция так же не будет продолжать суммирование, а сразу вернет текст ошибки – таким образом всегда сообщение о первой же встреченной ошибке будет возвращено как результат работы функции.

Для завершения реализации функции `calculate()` необходимо самостоятельно добавить в нее поддержку вычитания и деления. Обратите внимание: блоки реализуются аналогичным образом, но крайне важен их правильный порядок: сложение, вычитание, умножение, деление. Кроме того, для вычитания и деления начальным накапливаем значением будет значения первого аргумента в цепочке, а не 0 или 1 для суммы и произведения соответственно. Тогда цикл перебора будет начинаться не с *0*-ого элемента (первого), а с *1*-ого (второго) – учтите это при программировании. Также самостоятельно реализуйте возможность использования в качестве символа деления и "*/*" и "*:*".

Для выполнения всех требований лабораторной работы также необходимо реализовать механизм сохранение истории вычислений, для чего удобно применить сессии.

Листинг В-2. 6

```
<?php
      session_start();    // подключаем механизм сессий
      if( !isset($_SESSION['history']) )  // если первая загрузка страницы
            $_SESSION['history']=array();      // создаем в сессии массив для истории
?>

<?php
          // код для вычисления выражения, статический HTML-код страницы
?>

<php
      // для всех элементов строк с историей вычислений
      for($i=0; $i<count($_SESSION['history']); $i++)
          echo $_SESSION['history'][$i].'<br>';      // выводим строку
      if($_POST['val'] )  // если было вычисление
          // сохраняем его в истории
          $_SESSION['history'][]=$_post['VAL'].' = '.$res;
?>
```

Для этого немного модифицируем код главной страницы: во-первых, в самом начале файл, до начала любого статического *HTML*-кода или до начала любой *PHP*-программы, добавляется

подключающий механизм сессий *PHP*-код. Это необходимо делать именно таким образом, т.к. для сессий используются так называемые заголовки файла, которые должны быть переданы до вывода его содержимого. Поэтому, если перед `session_start()` будет хотя бы один байт содержимого документа, сессии не будут подключены – вместо них появится сообщение об ошибке.

Сразу после подключения сессий, в программе проверяется: не первая ли это загрузка документа. Если первая, то в массиве `$_SESSION` с данными сессии ничего нет – в таком случае мы создаем там пустой массив для хранения истории: при следующей проверке история, пусть даже и в виде пустого массива, уже будет там, а значит проверка на первую загрузку пройдена не будет.

После этого можно добавлять статический *HTML*-код, вычислять результат выражения и т.д. В конце документа, в его подвале, размещается последний фрагмент *PHP*-программы. В первую очередь он выводит содержимое истории построчно, как и сказано в задании к лабораторной работе. Затем, если в *PHP*-программу было передано выражение для вычисления, он добавляет в историю собственно выражение и результат его вычисления (число или текст ошибки), который хранится в переменной `$res` (см. листинг В-2. 1). Таким образом, текущий результат вычисления не выводится в истории, но сохранится в ней для последующего вывода – что и требовалось по условиям лабораторной работы.

Обратите внимание, что если просто обновить страницу (нажать *F5*), то каждый раз в историю будет добавляться новая строка. Действительно, при обновлении страницы в нее поступают те же параметры, что и в прошлый раз, следовательно, *PHP*-программа сработает та же, т.е. добавит в массив с историей вычислений новую строку. С одной стороны, это логически правильно, но с пользовательской точки зрения – абсолютно неверно. Давайте модифицируем код так, чтобы избежать подобной ситуации.

Для этого можно использовать следующий подход. Введем в сессии некоторый элемент, в котором будет хранится номер загрузки документа. Т.е. при каждой перезагрузке документа в рамках одной сессии он будет увеличиваться на 1. Для этого в обработчик первой загрузки страницы (см. листинг В-2. 6) включим инициализацию этого элемента в массиве данных сессии: `$SESSION['iteration']=0`. Кроме этого, при каждой загрузке страницы это счетчик будет увеличиваться на 1. Тогда, при первой загрузке в этом элементе будет хранится 1, при второй – 2, и т.д.

Листинг В-2. 7

```
if( !isset($_SESSION['history']) )   // если первая загрузка страницы
{
      $_SESSION['history']=array(); // создаем в сессии массив для истории
      $SESSION['iteration']=0;      // первая загрузка документа
}
$SESSION['iteration']++;
```

Величину этого счетчика будем загружать в качестве значения для одноименного скрытого поля в форме. Тогда, при каждой отправке данных формы, в качестве параметра в *PHP*-программу будет передаваться значение счетчика на момент генерации *HTML*-кода формы.

Листинг В-2. 8

```
<input type="hidden" name="iteration" value="<?php echo $SESSION['iteration']; ?>">
```

Если теперь в качестве условия возможности обработки данных формы использовать не просто проверку существования переданных параметров, но и совпадение увеличенного на единицу переданного из формы значения счетчика с хранящимся в сессии, то повторная обработка при обновлении страницы происходить не будет.

Листинг В-2. 9

```
// если было вычисление и это не обновление страницы
if($_POST['val'] && $_POST['iteration']+1=$_SESSION['iteration'])
```

```
    // сохраняем его в истории
    $_SESSION['history'][]=$_post['VAL'].' = '.$res;
```

Действительно, если значение счетчика на момент формирования *HTML*-кода формы было 60, то и в обработчик поступит именно это значение. Но при обработке значение счетчика будет уже 61 (это следующая загрузка страницы), значит 60+1=61, т.е. необходимо обработать данные формы. Если же после этого страница была обновлена, то это уже 62 загрузка, но данные формы остались прежними: 60+1≠62, т.е. обработка не требуется. Реализуйте этот механизм для данной лабораторной работы.

Вычисление выражения со скобками

Итак, лабораторная работа полностью выполнена, но в выражении недопустимо использование скобок: доработаем PHP-программу для исправления такого несоответствия. Для этого, во-первых, разработаем новую вспомогательную функцию `SqValidator()`, которая бы проверяла корректность расстановки скобок в выражении: она должна соответствовать правилам математики. В основе работы будем использовать классический вариант алгоритма с подсчетом открывающихся и закрывающихся скобок.

Листинг B-2. 10

```
function SqValidator( $val )
{
    $open=0;                        // создаем счетчик открывающихся скобок
    for($i=0; $i<strlen($val); $i++)          // перебираем все символы строки
    {
        if( $val[$i]=='(' )  // если встретилась «(»
                $open++;          // увеличиваем счетчик
          else
        if( $val[$i]==')' )        // если встретилась «)»
        {
            $open--;        // уменьшаем счетчик
            if( $open<0 )         // если найдена «)» без соответствующей «(»
                return false;   // возвращаем ошибку
        }
    }
     // если количество открывающихся и закрывающихся скобок разное
     if( $open!==0 )
           return false;// возвращаем ошибку
    return true;          // количество скобок совпадает - все ОК
}
```

В начале работы инициализируем переменную в которой будем хранить количество встретившихся открывающихся скобок: естественно, что перед разбором строки она хранит ноль. Затем в цикле мы перебираем все символы переданной в функцию строки (проверяемого выражения). Если текущий символ является открывающейся скобкой "*(*", то счетчик `$open` увеличивается на единицу. Если закрывающейся скобкой – то счетчик уменьшается на 1. При этом если его значение стало меньше нуля – это значит, что количество встретившихся закрывающихся скобок больше количества открывающихся, т.е. анализируется выражение вида "*(2+3))-1*" – т.е. оно нарушает правила и поэтому функция возвращает ошибку.

Таким образом каждая встретившаяся открывающаяся скобка увеличивает счетчик, каждая закрывающаяся – уменьшает. Если в конце работы функции значение счетчика `$open` не равно нулю – значит количество скобок разное и функция опять-таки возвращает ошибку. Такой алгоритм хорош тем, то не только проверяет совпадение числа открывающихся и закрывающихся скобок, но и проверяет порядок их следования, т.е. чтобы каждому символу "*(*" соответствовал свой символ "*)*". Без этого выражение "*9+)3+4(*" было бы при проверке признано верным.

Во-вторых, вновь возвращаясь к собственно реализации вычислению значения выражения, необходимо разработать новую функцию `calculateSq()`, которая собственно и будет анализировать и вычислять значение выражения со скобками. Причем у нас уже имеется вычисляющая выражение без скобок функция `calculateSq()`, поэтому задача вновь

разрабатываемой функции на самом деле проще: выделить в выражении все что в скобках, вычислить эти части выражения, а затем вычислить и все выражение целиком.

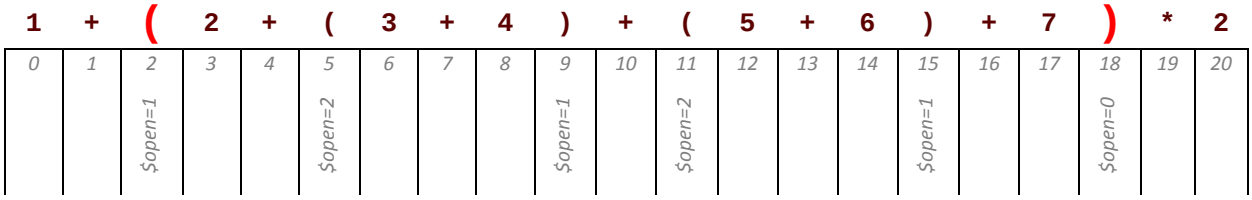
Листинг В-2. 11

```
function calculateSq( $val )
{
       // проверка на корректность использования скобок в выражении
      if( !SqValidator($val) ) return 'Неправильная расстановка скобок';

       $start= strops('(', $val);     // ищем первую открывающуюся скобку

      if( $start===false )            // если в выражении нет скобок
             return calculate($val);       // используем функцию calculate()

       ////////////////////////////////////////////////////////////////////
       // ищем соответствующую открывающейся закрывающуюся скобку
       ////////////////////////////////////////////////////////////////////
       $end=$start+1; // первое место поиска – следующий символ
      $open=1;         // количество найденных открывающихся скобок пока 1 шт.

      // цикл пока скобка не найдена или не дошли до конца строки
      // признаком найденной скобки является обнуление счетчика скобок
      while( $open && $end<strlen($val) )
      {
            if( $val[ $end ]=='(' )          // символ «(» увеличивает счетчик
                  $open++;
            if( $val[ $end ]==')' )          // символ «)» уменьшает счетчик
                  $open--;
               $end++;
       }

       ////////////////////////////////////////////////////////////////////////////
       // формируем новое выражение, путем замены содержимого скобок на вычисленное
       ////////////////////////////////////////////////////////////////////////////
       $new_val = substr($val, 0, $open); // часть исходного выражение левее скобок

       $new_val.=calculateSq( substr($val, $open+1, $end-$open-2) ); // часть в скобках

       $new_val .= substr($val, $end);     // часть исходного выражение правее скобок

      return calculateSq( $new_val ); // вычисляем новое выражение и возвращаем его
}
```

Для работы данная функция, как и `calculate()`, использует рекурсивный подход. В качестве базы рекурсии используется, во-первых, проверка корректности расстановки скобок с помощью разработанной ранее функции `SqValidator()`: если скобки расставлены неверно, то и дальнейшая работа бессмысленна – функция сразу возвращает сообщение об ошибке. Во-вторых, если калькулятор может считать выражение со скобками, это не значит, что он не должен считать выражения без них. Поэтому, и это основная часть базы, осуществляется поиск открывающейся скобки в выражении: если таких скобок нет, то это значит, что значение выражения можно посчитать уже имеющейся в распоряжении функцией `calculate()`. Причем в этом случае в выражении гарантировано отсутствуют и закрывающиеся скобки: иначе функция `SqValidator()` возвратила бы ошибку. Но даже если бы такой проверки не было бы, то передача в функцию `calculate()` выражения с закрывающейся скобкой (да и с любой другой) также привело бы к ошибке. Поэтому дальнейший код выполняется только для правильных выражений с имеющимися скобками – для остальных случаев мы уже умеем получать правильный результат.

Давайте представим, как мы сами вычисляем значение содержащего скобки выражение и постараемся запрограммировать эти же действия на *PHP*. Вкратце этот процесс можно описать следующим образом: если в выражении встретилась открывающаяся скобка, то человек ищет соответствующую ей закрывающуюся и вычисляет находящееся в них выражение, после чего заменяет все то что в скобках полученным числом.

Поэтому, следующая часть функции с помощью стандартной функции *PHP* `strops()` ищет в строке позицию первого вхождения символа "`(`" и сохраняет ее в переменной `$start`. Затем, начиная со следующего символа, мы в цикле перебираем все символы строки до тех пор, пока она не закончится или же не будет найдена соответствующая открывающейся скобке закрывающаяся. При этом используется такой же алгоритм с увеличивающимся и уменьшающимся счетчиком скобок, как и в функции `SqValidator()`: каждый встреченный символ «*(*» увеличивает счетчик скобок на единицу, «*)*» – уменьшает. Значение счетчика `$open` "`0`" является признаком нахождения искомой закрывающейся скобки и цикл прекращается. Например, разберем следующее выражение.



Первая открывающаяся скобка встречается в символе №2 – поэтому значение переменной `$start` равняется "`2`", а счетчика `$open` – "`1`". Далее, по мере работы цикла, при проверке символа №5 открывающаяся скобка встретится вновь, а значит счетчик будет увеличен на 1: `$open=2`. Символ №9 опять уменьшит ее на 1, 11-ый и 15-ый символы сделают тоже самое, пока символ № 18 не обнулит счетчик: это значит, что соответствующая закрывающаяся скобка располагается именно в 18-ом символе. Таким образом, даже если внутри скобок присутствуют другие пары скобок сколь угодно большой степени вложенности, алгоритм найдет именно соответствующую самой первой закрывающуюся скобку.

Итак, в данном примере, после завершения работы цикла, в переменных будут следующие значения: `$start=2`, `$end=19` (обратите внимание, переменная `$end` содержит номер следующего за закрывающейся скобкой символа). Теперь остается только сформировать нового выражение для вычисления состоящее из трех частей:

- то что левее открывающейся скобки;
- то что правее закрывающейся скобки;
- значения выражения в скобках.

Именно это мы и делаем в последней части кода функции: вычисляем значение нового выражение состоящего из "`1+`", "`27`" и "`*2`", т.е. "`1+27*2`" (27 – значение выражения внутри скобок). При этом дважды рекурсивно вызывается та же самая функция `calculateSq()` – для вычисления значения найденного выражения внутри скобок и для значения нового выражения без скобок. Таким образом, каждый вызов функции `calculateSq()` убирает из выражения одну пару скобок до тех пор, пока их совсем не останется и его вычисление будет возможно функций `calculate()`.

Самостоятельно модифицируйте код обработки данных формы так, чтобы обработчик корректно обрабатывал выражения со скобками, тем более что для этого необходимо всего лишь добавить в него два символа.

СПРАВОЧНАЯ ИНФОРМАЦИЯ

Подключение механизма сессий	`session_start();` Начинает (или продолжает) работу с сессией. Без вызова этой функции механизм работать не будет. Важно: функция должна быть вызвана до любого статического или динамического формирования *PHP*-кода. Если перед функцией будет передан хотя бы один байт данных, вызов приведет к ошибке.
Инициализация данных в сессии	`if(isset($_SESSION['key'])) $_SESSION['key']=$value;` Все данные сессии доступны через обращение к суперглобальному массиву: он доступен из любого места программы, в том числе из

	любой функции. Его можно использовать как любой другой массив: создавать в нем любое количество элементов, удалять их, изменять значение и т.д. – все эти данные будут доступны при обращении к массиву после перезагрузки документа или даже из другой страницы сайта. Поэтому, если возможна ситуация, когда в программе перед записью данных в элемент массива идет обращение к нему, его необходимо предварительно инициализировать.
Запись данных в сессию	`$_SESSION['key']=$value;` При перезагрузке страницы в элементе массива `$_SESSION['key']` сохранится значение переменной `$value`.
Чтение данных из сессии	`echo $_SESSION['key'];` Читаем и выводим сохраненное в массиве значение.

Вычисление выражения средствами PHP	`eval('$res='.$task.';');`
	`echo $res;`
	Функция интерпретирует строковый аргумент как PHP-код. В данном примере если в переменной $task хранится какое-либо вычисляемое выражение, например, "*2*3+5/2*", то будет выполнен PHP-код: "`$res=2*3+5/2;`", т.е. в переменной `$res` будет сохранено вычисленное значение выражения. Остается только вывести его в браузер.

КОНТРОЛЬНЫЕ ВОПРОСЫ К ЛАБОРАТОРНОЙ РАБОТЕ

Для успешной защиты работы помимо соответствующего требованиям результата необходимо уверенно отвечать на нижеперечисленные и другие вопросы.

1. Как изменится работа функции из листинга В-2. 4, если из нее убрать проверку на пустую строку?
2. Что такое рекурсия в программировании? Приведите примеры рекурсии.
3. Что такое база рекурсии?
4. Почему математические действия в функции calculate() выполняются именно в такой последовательности?
5. Что такое сессия в PHP?
6. Почему нельзя начинать инициализировать данные сессии после любого вывода?
7. Можно ли инициализировать данные сессии внутри тега <body>, но при до вывода данных средствами PHP?
8. Как PHP разделяет пользователей для доступа к разным хранилищам сессий?
9. Как получить доступ к данным сессии из PHP?
10. Как долго можно обращаться к данным сессии?
11. Можно ли хранить в сессии массивы и строки больше 1024 символов?
12. Как записать данные в сессию?
13. Можно ли прочитать данные из сессии если они туда не записаны? Как быть том случае, если такая ситуация возможна?
14. Как быстро вычислить любое выражение средствами PHP?
15. Можно ли в PHP интерпретировать символ строки как число?