Программа htm2php

Преобразователь текста html в список строковых констант для php

Жуков И. Б.
ibzh@yandex.ru
http://ibzh.16mb.com
http://ibzh.novhost.cf

2015

1. Программа *htm2php*. Программа *htm2php* преобразует html-текст в строки, пригодные для вставки в php. Исходно она предназначалась для обработки кода всевозможных счётчиков вроде liveinternet, yandex.метрика и т. п., а также других текстов, которые вам предлагают разместить на всех страницах вашего сайта. (Делать это, разумеется, проще всего с помощью php-скриптов.) Текст данной программы в интернете находится здесь.

2. Программа написана в стиле литературного программирования с использованием русифицированных средств. Чтобы её откомпилировать, вам понадобятся сами средства литературного программирования, которые можно найти по адресу
– http://ibzh.16mb.com или
– http://ibzh.novhost.cf,
а также компилятор языка паскаль, который можно скачать по адресу www.freepascal.com.

3. Программу будем писать по-русски, и для этого введём следующие тексты замены.

format *без_конца* ≡ *until*
define *без_конца* ≡ **until** (*false*)
format *выйди_из_процедуры* ≡ *xclause*
define *выйди_из_процедуры* ≡ *exit*
define *выйди_из_функции* ≡ *exit*
format *выйди_из_функции* ≡ *xclause*
format *да* ≡ *true* ;
define *да* ≡ *true*;
define *делай* ≡ **do**
format *делай* ≡ *do*
define *если* ≡ **if**
format *если* ≡ *if*
define *заверши_программу* ≡ *halt*
format *заверши_программу* ≡ *xclause*
define *и_если* ≡ ∧
define *иди_к* ≡ **goto**
format *иди_к* ≡ *goto*
define *или* ≡ ∨
define *иначе* ≡ **else**
format *иначе* ≡ *else*
define *конец* ≡ **end**
format *конец* ≡ *end*
format *константа* ≡ *const*
define *константа* ≡ **const**
format *константы* ≡ *const*
define *константы* ≡ **const**
define *метки* ≡ **label**
format *метки* ≡ *label*
define *начало* ≡ **begin**
format *начало* ≡ *begin*
define *не* ≡ ¬
format *нет* ≡ *false* ;
define *нет* ≡ *false*;
format *переменная* ≡ *var*
define *переменная* ≡ **var**
define *переменные* ≡ **var**
format *переменные* ≡ *var*

4. Это продолжение определений предыдущего раздела.

```
define повторяй ≡ repeat
format повторяй ≡ repeat
define пока ≡ while
format пока ≡ while
define пока_не ≡ until
format пока_не ≡ until
format прерви_цикл ≡ xclause ;
define прерви_цикл ≡ break;
format программа ≡ program
define программа ≡ program
define процедура ≡ procedure
format процедура ≡ procedure
define то ≡ then
format то ≡ then
define функция ≡ function
format функция ≡ function
```

5. Программа должна из командной строки получить имя входного файла, обработать этот файл и записать итог в выходной файл. Имя выходного файла образуется из имени входного добавлением к нему расширения «.php».

Если программа отработала без ошибок, то операционной системе возвращается код завершения 0. Если была ошибка, то код завршения 1. (Код завершения можно проверить в командном файле, используя переменную окружения %errorlevel%.)

Ниже дан кусочек программы, выводящий справку по её использованию.

```
⟨ Выведи справку о программе и заверши её работу 5 ⟩ ≡
  начало writeln; writeln('Программа преобразует htm-файл в php-строки'); writeln;
  writeln('Использование: htm2php <имя_файла> '); writeln;
  writeln('Итог записывается в файл <имя_файла.php> '); writeln;
  заверши_программу ;
  конец;
```

Этот код используется в разделе 33.

6. Общий набросок программы выглядит следующим образом. Исходный файл считывается в буфер *a_in*, затем выполняется преобразование строк файла, итог которого помещается в буфер *a_out*. И, наконец этот буфер записывается на диск.

```
  @{$COPERATORS+@}
программа htm2php;
  переменные ⟨ Общие переменные 10 ⟩
    ⟨ Процедуры и функции 8 ⟩
  метки ⟨ Метки программы 28 ⟩
    начало ⟨ Открой входной файл 33 ⟩
    ⟨ Прочитай входной файл в буфер a_in 35 ⟩;
    ⟨ Закрой файл 36 ⟩
    ⟨ Выполни преобразование файла в строки. Итог сохрани в буфере в буфере a_out 26 ⟩
    ⟨ Открой выходной файл 37 ⟩
    ⟨ Запиши выходной файл 38 ⟩
    ⟨ Закрой выходной файл 39 ⟩
    конец.
```

7. Между следующими командами будем размещать код для отладки программы. В обычной работе он не нужен, и поэтому он помещается в комментарии. Если потребуется что-то отлаживать, то нужно заменить определения **отладка** и **конец_отладки** на **begin** и **end** , соответственно.

```
define отладка ≡ @{  { begin }
format отладка ≡ begin
define конец_отладки ≡ @}  { end; }
format конец_отладки ≡ end  ;
```

8. Когда программа обнаруживает ошибки, она вызывает процедуру *ошибка*, выводящую сообщение об ошибке и завершающую выполнение программы с кодом 1. Сообщение указывается строкой в аргументе функции.

```
⟨ Процедуры и функции 8 ⟩ ≡
процедура ошибка(s : string);
  начало writeln; writeln(s);
  заверши_программу (1);
  конец;
```

См. также разделы 12 и 25.

Этот код используется в разделе 6.

9. Основная задача.

10. Сначала определим переменные, необходимые для обработки данных, а также процедуры доступа к данным и основных проверок. Указателями рассматриваемых мест в буферах *a_in* и *a_out* служат переменные i и j, соответственно. Наибольшие занятые места в буферах хранятся в переменных *s_in* и *s_out*. Наибольший размер входного буфера определяется константой *limit*. Выходной буфер делается немного больше, чтобы не выполнять проверки при вставке каждого знака в выходной буфер. (Дополнительные 10 мест в выходном буфере могут использоваться для размещения знаков, завершающих строку: кавычек, точки с запятой, точки, соединяющей константу со следующей строкой и др.)

Поскольку преобразование файла состоит в вставке в его содержимое дополнительных знаков, входной файл должен быть несколько меньше величины *limit*, иначе не хватит места в выходном буфере.

define *limit* = 10240 { наибольший размер входного файла }

⟨ Общие переменные 10 ⟩ ≡
a_in: **array** [0 .. *limit*] **of** *byte*; { здесь размещается входной файл }
a_out: **array** [0 .. *limit* + 10] **of** *byte*; { здесь составляется выходной файл }
s_in: *dword* = 0; { число входных символов }
s_out: *dword* = 0; { число выходных символов }
i, *j*: *dword*; { индексы в буферах *a_in* и *a_out*, соответственно }

См. также разделы 16 и 34.

Этот код используется в разделе 6.

11. Простейшие действия с буферами

define *входной_знак* ≡ *a_in*[*i*] { даёт текущий входной знак }
define *есть_входные_знаки* ≡ (*i* ⩽ *s_in*) { проверка, что есть, что читать }
define *нет_входных_знаков* ≡ (*i* > *s_in*) { проверка обратного условия }
define *есть_место_в_выходном_буфере* ≡ (*j* < *limit*) { проверка, что есть, куда писать }
define *нет_места_в_выходном_буфере* ≡ (*j* ⩾ *limit*) { проверка обратного условия }
define *копируй_знак* ≡
  начало *a_out*[*j*] ← *входной_знак*;
  отладка *write*(*char*(*a_out*[*j*]));
  конец_отладки*i*+ = 1; *j*+ = 1; { переносит знак из одного буфера в другой }
  конец
define *помести_знак*(#) ≡
  начало *a_out*[*j*] ← #;
  отладка *write*(*char*(*a_out*[*j*]));
  конец_отладки*j*+ = 1; { Помещает знак в выходной буфер }
  конец
define *следующий_входной_знак*(#) ≡ *a_in*[#] { даёт *k*-ый входной знак }
define *последний_входной_знак*(#) ≡ (# = *s_in*) { проверяет, что *k*-ый знак был последним }

12. То, что мы подошли слишком близко к концу буфера, проверяет следующая процедура, она выдаёт сообщение об ошибке и завершает программу.

⟨ Процедуры и функции 8 ⟩ +≡
процедура *проверь_место_в_выходном_буфере*;
  начало если *есть_место_в_выходном_буфере* **то выйди_из_процедуры** ;
  ошибка(´Выходной␣буфер␣слишком␣мал´);
  конец;

13. Опишем суть задачи. На вход поступает HTML-текст, который должен построчно заключаться в кавычки. Текст помещается между кавычками один в один, кроме следующих символов, которые должны обрабатываться особым образом:
– одинарные кавычки;
– двойные кавычки;
– знаки обратной косой черты;
– знак "$";
– переводы строки.

14. Стро́ки в php могут заключаться или в одинарные кавычки, или в двойные. В последнем случае строки обрабатываются особым образом: в них раскрываются переменные, начинающиеся со знака "$", и команды, начинающиеся со знака "\". Поэтому части текста HTML, содержащие указанные знаки, должны заключаться в одинарные кавычки.

Далее идёт подстановка, указывающая, что знак обрабатывается особым способом в двойных кавычках

define *вх_знак_особый_в_двойных_кавычках* ≡ ((*входной_знак* = "\")*или*(*входной_знак* = "$")*или*(*входной_знак* = """))

15. С другой стороны, части строки, содержащие одинарные кавычки, должны заключаться в двойные. Нужно ли менять вид кавычек с одинарных на двойные, определяет следующая подстановка

define *вх_знак_особый_в_обычных_кавычках* ≡ (*входной_знак* = "´")

16. То, какие сейчас кавычки используются, будет определяться переменной *двойные_кавычки*

⟨ Общие переменные 10 ⟩ +≡
двойные_кавычки: *boolean*;

17. Если эта переменная истина, то HTML-текст сейчас помещается между двойными кавычками, иначе между одинарными. Переключение производится следующими подстановками

define *теперь_двойные_кавычки* ≡ *двойные_кавычки* ← *true*
define *теперь_обычные_кавычки* ≡ *двойные_кавычки* ← *false*

18. Следующие подпрограммы меняют текущий вид кавычек.

```
define смени_кавычки_на_двойные ≡
        begin помести_знак("´"); помести_знак("."); помести_знак("""");
        теперь_двойные_кавычки;
        end
define смени_кавычки_на_одинарные ≡
        begin помести_знак(""""); помести_знак("."); помести_знак("´");
        теперь_обычные_кавычки;
        end
```

19. Рассмотрим теперь общий цикл обработки строки. Строка начинается со знака одинарной или двойной кавычки. Правило такое: если первый знак в исходном файле — одинарная кавычка, то преобразованная строка должна начинаться с двойной кавычки. Во всех прочих случаях — с одинарной. Этот кусочек кода обрабатывает начало непустой строки.

```
⟨ Обработай строку входного файла 19 ⟩ ≡
  если вх_знак_особый_в_обычных_кавычках то
    begin помести_знак("""); теперь_двойные_кавычки;
    end
  иначе begin помести_знак("'"); теперь_обычные_кавычки;
    end;
```

См. также разделы 20, 21, 22, 23 и 24.

Этот код используется в разделе 26.

20. Следующий кусочек копирует данные до конца строки. Если встречается особый знак, то производится переключение на другой тип кавычки.

```
  define конец_строки ≡ (входной_знак = 13)
⟨ Обработай строку входного файла 19 ⟩ +≡
  пока (неконец_строки) делай
    начало копируй_знак;
    если двойные_кавычки и_если вх_знак_особый_в_двойных_кавычках то
      смени_кавычки_на_одинарные
    иначе если (недвойные_кавычки) и_если вх_знак_особый_в_обычных_кавычках то
        смени_кавычки_на_двойные;
    если нет_входных_знаков то прерви_цикл ;
    проверь_место_в_выходном_буфере;
    конец;
```

21. В конце строки ставится завершающая кавычка соответствующего вида. Кроме того устанавливается переменная числа готовых для вывода символов s_out. (На самом деле s_out указывает на следующее место за последним готовым символом, так что в буфер $a\_in[s\_out]$ нужно будет просто поместить точку с запятой перед выводом.)

```
⟨ Обработай строку входного файла 19 ⟩ +≡
  если двойные_кавычки то помести_знак("""")
  иначе помести_знак("'");
  s_out ← j;
  если нет_входных_знаков то ⟨ Иди к завершению обработки файла 29 ⟩
```

22. Сейчас мы указываем на знак с кодом 13, нам надо с него перейти на следующий. Также не забудем о том, что нужно пропустить символ с кодом 10, который во многих системах употребляется вместе со знаком 13.

```
⟨ Обработай строку входного файла 19 ⟩ +≡
  i+ = 1;
  если (входной_знак = 10) то i+ = 1;   { передвинем указатель в a_in на одно место вправо }
```

23. Рассмотрим теперь, как должен обрабатываться конец строки. Если это не последняя строка файла, то нужно поставить точку и символы «"\n"␣.». Последние символы заставят при обработке php сохранить имевший место перевод на другую строку. Если следующая строка пустая, то логично поместить знаки \n в те же кавычки.

⟨ Обработай строку входного файла 19 ⟩ +≡
```
поместиЗнак("␣"); поместиЗнак("."); поместиЗнак("␣"); поместиЗнак("""");
повторяй поместиЗнак("\"); поместиЗнак("n"); проверьМестоВВыходномБуфере;
  отладка writeln; write('i␣=␣', i, '␣␣␣␣␣s_in␣=␣', s_in, '␣␣␣␣␣j␣=␣', j, '␣␣␣␣␣s_out␣=␣', s_out);
  writeln;
  конец_отладки;
пока_не (непустая_строка);
если нет_входных_знаков то ⟨ Иди к завершению обработки файла 29 ⟩
поместиЗнак("""); поместиЗнак(".");
```

24. Следующее, что нужно сделать, это поместить перевод на следующую строку в создаваемом php-файле

⟨ Обработай строку входного файла 19 ⟩ +≡
```
поместиЗнак(13); поместиЗнак(10);
```

25. Теперь надо разобраться с тем, как определяется то, что следующая строка пустая. Делать мы это будем при помощи функции *пустая_строка*, которая будет просматривать знаки во входном буфере и возвращать *да*, если следующая строка пустая. Если строка из одних пробелов, то они все отбрасываются. Если дошли до конца строки, то перескакиваем с него на начало следующей строки, а если встретился символ, отличный от пробела и конца строки, то возвращаемся к началу строки.

```
define входной_знак_не_пробел ≡ ((входной_знак ≠ "␣")и_если(входной_знак ≠ 9))
```

⟨ Процедуры и функции 8 ⟩ +≡
```
функция пустая_строка: boolean;
  переменная k: word;
  начало k ← i;  { Сохраним i }
  пустая_строка ← нет;  { наиболее вероятный случай }
  пока (есть_входные_знаки) делай
    начало если конец_строки то
      начало i+ = 1;
      если входной_знак = 10 то i+ = 1;  { перейдём к началу следующей строки }
      пустая_строка ← да;
      отладка writeln; writeln('<пустая_строка:true;␣␣␣␣␣␣i=', i, '␣>');
      конец_отладки
      выйди_из_функции ;
      конец;
    если входной_знак_не_пробел то
      начало i ← k;  { вернёмся к началу строки }
      отладка writeln; writeln('<пустая_строка:false;␣␣␣␣␣␣i=', i, '␣>');
      конец_отладки
      выйди_из_функции ;
      конец;
    i+ = 1;
    конец;
  отладка writeln; writeln('<пустая_строка:false;␣␣␣␣␣␣i=', i, '␣>');
  конец_отладки
  конец;
```

26. Теперь можно описать следующий кусочек.

⟨ Выполни преобразование файла в строки. Итог сохрани в буфере в буфере *a_out* 26 ⟩ ≡
  $i \leftarrow 0$; { Установим указатель на начало буфера }
  ⟨ Пропусти пустые строки, если есть 27 ⟩
  повторяй ⟨ Обработай строку входного файла 19 ⟩
  без_конца ;
завершение_обработки: ⟨ Обработка конца вывода 30 ⟩

Этот код используется в разделе 6.

27. ⟨ Пропусти пустые строки, если есть 27 ⟩ ≡
  повторяй пока_не (*непустая_строка*);

Этот код используется в разделе 26.

28. Метки в программе — страшный сон для структурного объектно-ориентированного программиста.

⟨ Метки программы 28 ⟩ ≡
  завершение_обработки;

Этот код используется в разделе 6.

29. ⟨ Иди к завершению обработки файла 29 ⟩ ≡
  goto *завершение_обработки*;

Этот код используется в разделах 21 и 23.

30. ⟨ Обработка конца вывода 30 ⟩ ≡
  если *s_out* $<$ 3 **то** *ошибка*(´Входной␣файл␣не␣содержит␣данных´);
  $j \leftarrow$ *s_out*; *помести_знак*(";");

Этот код используется в разделе 26.

31. На этом основная программа закончилась, остались только рутинные операции по открытию и закрытию файлов.

32. Работа с файлами.

33. Имя входного файла принимается из командной строки. В ней должно быть указано только одно имя

```
⟨ Открой входной файл 33 ⟩ ≡
  если ParamCount < 1 то
    ошибка('Не␣указан␣входной␣файл.' + #13#10 + 'Для␣справки␣наберите␣htm2php␣-?');
  если ParamCount > 1 то ошибка('Слишком␣много␣параметров␣во␣входной␣строке');
  если ParamStr(1) = '-?' то ⟨ Выведи справку о программе и заверши её работу 5 ⟩
  assign(f, ParamStr(1)); @{$I-@}reset(f); @{$I+@}
    если IOResult ≠ 0 то ошибка('Не␣могу␣открыть␣файл␣"␣' + ParamStr(1) +
        '␣"␣возможно,␣он␣имеет␣недопустимое␣имя');
```

Этот код используется в разделе 6.

34. Здесь нам понадобится переменная файлового типа. Будем рассматривать все файлы, как последовательности байт.

```
⟨ Общие переменные 10 ⟩ +≡
f: file of byte;
```

35.

```
⟨ Прочитай входной файл в буфер a_in 35 ⟩ ≡
  s_in ← 0;
  если eof(f) то ошибка('Входной␣файл␣пуст');
  пока (s_in < limit) делай
    начало read(f, a_in[s_in]);
    если eof(f) то прерви_цикл ;
    s_in += 1;
    конец;
  если не eof(f) то ошибка('Входной␣файл␣слишком␣длинный');
```

Этот код используется в разделе 6.

36. Следующий кусочек закрывает входной файл и, если выполняется отладка, выводит его содержимое на экран.

```
⟨ Закрой файл 36 ⟩ ≡
  начало close(f);
  отладка writeln('Прочитан␣файл:');
  for i ← 0 to s_in do write(char(a_in[i]));
  i ← 0; write('|'); writeln; writeln('Пишется␣файл:');
  конец_отладки;
  конец;
```

Этот код используется в разделах 6 и 39.

37.

```
⟨ Открой выходной файл 37 ⟩ ≡
  assign(f, ParamStr(1) + '.php'); @{$I-@}rewrite(f); @{$I+@}
    если IOResult ≠ 0 то ошибка('Не␣могу␣создать␣файл␣"␣' + ParamStr(1) + '.php␣"');
```

Этот код используется в разделе 6.

38.

⟨ Запиши выходной файл 38 ⟩ ≡
```
j ← 0;
пока (j ⩽ s_out) делай
  начало write(f, a_out[j]); j+ = 1;
  конец;
```
Этот код используется в разделе 6.

39.

⟨ Закрой выходной файл 39 ⟩ ≡
```
⟨ Закрой файл 36 ⟩
writeln; writeln('Готово.␣␣Итоговый␣файл:␣<' + ParamStr(1) + '.php>');
```
Этот код используется в разделе 6.

40. Указатель.

⟨ Выведи справку о программе и заверши её работу 5 ⟩ Используется в разделе 33.
⟨ Выполни преобразование файла в строки. Итог сохрани в буфере в буфере *a_out* 26 ⟩
Используется в разделе 6.
⟨ Закрой выходной файл 39 ⟩ Используется в разделе 6.
⟨ Закрой файл 36 ⟩ Используется в разделах 6 и 39.
⟨ Запиши выходной файл 38 ⟩ Используется в разделе 6.
⟨ Иди к завершению обработки файла 29 ⟩ Используется в разделах 21 и 23.
⟨ Метки программы 28 ⟩ Используется в разделе 6.
⟨ Обработай строку входного файла 19, 20, 21, 22, 23, 24 ⟩ Используется в разделе 26.
⟨ Обработка конца вывода 30 ⟩ Используется в разделе 26.
⟨ Общие переменные 10, 16, 34 ⟩ Используется в разделе 6.
⟨ Открой входной файл 33 ⟩ Используется в разделе 6.
⟨ Открой выходной файл 37 ⟩ Используется в разделе 6.
⟨ Пропусти пустые строки, если есть 27 ⟩ Используется в разделе 26.
⟨ Процедуры и функции 8, 12, 25 ⟩ Используется в разделе 6.
⟨ Прочитай входной файл в буфер *a_in* 35 ⟩ Используется в разделе 6.