UTF8 strings and characters

The beauty of UTF-8

Bytes starting with '0' (0xxxxxxx) are reserved for ASCII-compatible single byte characters. With multi-byte codepoints the number of 1's in the leading byte determines the number of bytes the codepoint occupies. Like this :

- 1 byte : 0xxxxxxx
- 2 bytes : 110xxxxx 10xxxxxx
- 3 bytes : 1110xxxx 10xxxxxx 10xxxxxx
- 4 bytes : 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

The design of UTF-8 has some benefits over other encodings :

- It is backwards compatible with ASCII and produces compact data for western languages. ASCII is also used in markup language tags and other metadata which gives UTF-8 an advantage with any language. However that backwards compatibility does not extend to code, since code has to be recrafted to avoid mangling utf8 strings.
- The integrity of multi-byte data can be verified from the number of '1'-bits at the beginning of each byte.
- You can always find the start of a multi-byte codepoint even if you jumped to a random byte position.
- A byte at a certain position in a multi-byte sequence can never be confused with the other bytes. This allows using the old fast string functions like Pos() and Copy() in many situations. See examples below.

Note that similar integrity features are also exists in UTF-16. (D800 range signals first surrogate, DC00 range signals second part of surrogate)

- Robust code. Code that deals with codepoints must always be done right with UTF-8 because multi-byte codepoints are common. For UTF-16 there is plenty of sloppy code which assumes codepoints to be fixed width.
- Most textual data moving in internet is encoded as UTF-8. Processing the data directly as UTF-8 eliminates useless conversions. Many (Unix-related) operating systems use UTF-8 natively.

Examples

Simply iterating over characters as if the string was an array of equal sized elements does not work with Unicode. This is not something specific to UTF-8, the Unicode standard is complex and the word

"character" is ambiguous. If you want to iterate over codepoints of a UTF-8 string, there are basically two ways:

- iterate over bytes - useful for searching a substring or when looking only at the ASCII characters in the UTF8 string, for example when parsing XML files.
- iterate over codepoints or characters - useful for graphical components like SynEdit, for example when you want to know the third printed character on the screen.

Searching a substring

Due to the special nature of UTF8 you can simply use the normal string functions for searching a sub-string. Searching for a valid UTF-8 string with Pos will always return a valid UTF-8 byte position:

```
uses LazUTF8;
...
procedure Where(SearchFor, aText: string);
var
  BytePos: LongInt;
  CodepointPos: LongInt;
begin
  BytePos:=Pos(SearchFor,aText);
  CodepointPos:=UTF8Pos(SearchFor,aText);
  writeln('The substring "',SearchFor,'" is in the text "',aText,'"',
    ' at byte position ',BytePos,' and at codepoint position ',CodepointPos);
end;
```

Search and copy

Another example of how Pos(), Copy() and Length() work with UTF-8. This function has no code to deal with UTF-8 encoding, yet it works with any valid UTF-8 text always.

```
function SplitInHalf(Txt, Separator: string; out Half1, Half2: string):
Boolean;
var
  i: Integer;
begin
  i := Pos(Separator, Txt);
  Result := i > 0;
  if Result then
  begin
    Half1 := Copy(Txt, 1, i-1);
    Half2 := Copy(Txt, i+Length(Separator), Length(Txt));
  end;
end;
```

Iterating over string looking for ASCII characters

If you only want to find characters in ASCII-area, you can use Char type and compare with Txt[i] just like in old times. Most parsers do that and they continue working.

```
procedure ParseAscii(Txt: string);
var
  i: Integer;
begin
  for i:=1 to Length(Txt) do
    case Txt[i] of
      '(': PushOpenBracketPos(i);
      ')': HandleBracketText(i);
    end;
end;
```

Iterating over string looking for Unicode characters or text

If you want to find all occurrances of a certain character or substring in a string, you can call PosEx() repeatedly.

If you want to test for different text inside a loop, you can still use the fast Copy() and Length(). UTF-8 specific functions could be used but they are not needed.

```
procedure ParseUnicode(Txt: string);
var
  Ch1, Ch2, Ch3: String;
  i: Integer;
begin
  Ch1 := 'Й';  // Characters to search for. They can also
  Ch2 := 'ۆ';  //  be combined codepoints or longer text.
  Ch3 := 'Å';
  for i:=1 to Length(Txt) do
  begin
    if Copy(Txt, i, Length(Ch1)) = Ch1 then
      DoCh1(...)
    else if Copy(Txt, i, Length(Ch2)) = Ch2 then
      DoCh2(...)
    else if Copy(Txt, i, Length(Ch3)) = Ch3 then
      DoCh3(...)
  end;
end;
```

The loop could be optimized by jumping over the already handled parts.

Iterating over string analysing individual codepoints

This code copies each codepoint into a variable of type String which can then be processed further.

```
procedure IterateUTF8(S: String);
var
  CurP, EndP: PChar;
  Len: Integer;
  ACodePoint: String;
begin
  CurP := PChar(S);        // if S='' then PChar(S) returns a pointer to #0
  EndP := CurP + length(S);
  while CurP < EndP do
  begin
    Len := UTF8CodepointSize(CurP);
    SetLength(ACodePoint, Len);
    Move(CurP^, ACodePoint[1], Len);
    // A single codepoint is copied from the string. Do your thing with it.
    ShowMessageFmt('CodePoint=%s, Len=%d', [ACodePoint, Len]);
    // ...
    inc(CurP, Len);
  end;
end;
```

Accessing bytes inside one UTF8 codepoint

UTF-8 encoded codepoints can vary in length, so the best solution for accessing them is to use an iteration. To iterate through codepoints use this code:

```
uses LazUTF8;
...
procedure DoSomethingWithString(AnUTF8String: string);
var
  p: PChar;
  CPLen: integer;
  FirstByte, SecondByte, ThirdByte, FourthByte: Char;
begin
  p:=PChar(AnUTF8String);
  repeat
    CPLen := UTF8CodepointSize(p);

    // Here you have a pointer to the char and its length
    // You can access the bytes of the UTF-8 Char like this:
    if CPLen >= 1 then FirstByte := P[0];
    if CPLen >= 2 then SecondByte := P[1];
    if CPLen >= 3 then ThirdByte := P[2];
    if CPLen = 4 then FourthByte := P[3];

    inc(p,CPLen);
```

```
  until (CPLen=0) or (p^ = #0);
end;
```

Accessing the Nth UTF8 codepoint

Besides iterating one might also want to have random access to UTF-8 codepoints.

```
uses LazUTF8;
...
var
  AnUTF8String, NthCodepoint: string;
begin
  NthCodepoint := UTF8Copy(AnUTF8String, N, 1);
```

Showing codepoints with UTF8CharacterToUnicode

The following demonstrates how to show the 32bit code point value of each codepoint in an UTF8 string:

```
uses LazUTF8;
...
procedure IterateUTF8Codepoints(const AnUTF8String: string);
var
  p: PChar;
  unicode: Cardinal;
  CPLen: integer;
begin
  p:=PChar(AnUTF8String);
  repeat
    unicode:=UTF8CodepointToUnicode(p,CPLen);
    writeln('Unicode=',unicode);
    inc(p,CPLen);
  until (CPLen=0) or (unicode=0);
end;
```

Decomposed characters

Due to the ambiguity of Unicode, compare functions and Pos() might show unexpected behavior when e.g. one of the string contains decomposed characters, while the other uses the direct codes for the same letter. This is not automatically handled by the RTL. It is not specific to any encoding but Unicode in general.

macOS

The file functions of the FileUtil unit also take care of macOS specific behaviour: macOS normalizes filenames. For example the filename 'ä.txt' can be encoded in Unicode with two different sequences (#$C3#$A4 and 'a'#$CC#$88). Under Linux and BSD you can create a filename with both encodings. macOS automatically converts the a umlaut to the three byte sequence. This means:

```
if Filename1 = Filename2 then ... // is not sufficient under macOS
if AnsiCompareFileName(Filename1, Filename2) = 0 then ... // not sufficient
under fpc 2.2.2, not even with cwstring
if CompareFilenames(Filename1, Filename2) = 0 then ... // this always works
(unit FileUtil or FileProcs
```

See also

- Character and string types