Dangers of NeXTSTEP Plists

Recently I wrote my own NeXTSTEP plist parser and serializer in Python. It was an educational exercise and if you are curious about the exact implementation details of that you can go check out the code here. However this post is not about how to write your own parser, in-fact I strongly recommend that you do not. This post is to give you a word of warning about some common misconceptions around how easy it is to parse NeXTSTEP plists using Foundation classes like NSDictionary.

Most of you are probably familiar with the fact that Xcode uses NeXTSTEP plists for the format when serializing project files. If you have ever done your own exploration of the project file you may be familiar with the fact you are able to read these files off disk very easily using NSDictionary or NSPropertyList (or the CoreFoundation counterparts: CFDictionary and CFPropertyList). This works great, you can easily examine the contents and structure without having to fight with loading any of Xcode's private frameworks for doing this. To test this out I'm going to create a new XML plist file:

```
$ cat xml
<plist version="1.0">
<dict>
    <key>hourglass</key>
    <string>⌛</string>
</dict>
</plist>
```

Here is the plist, and to verify that this is a valid XML plist I will run it through `plutil`:

```
$ plutil -lint xml
xml: OK
```

Everything seems to check out as being ok. I wrote a small program that will parse plist using NSDictionary to load it from a file and print out the key:value pairs written in the plist.

```
#import <Foundation/Foundation.h>

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        NSString *runningDirectory = [[[[NSProcessInfo processInfo] arg
        NSArray *plistTypes = @[ @"xml" ];
        for (NSString *typeName in plistTypes) {
            NSLog(@"Plist Type: %@", typeName);
            NSString *plistPath = [runningDirectory stringByAppendingPa
            NSDictionary *plist = [NSDictionary dictionaryWithContentsO
            NSLog(@"Contents:");
            for (NSString *keyString in [plist allKeys]) {
                NSLog(@"\t%@ = %@", keyString, plist[keyString]);
            }
            NSLog(@"=======================");
        }
    }
    return 0;
}
```

I am going to compile it `clang main.m -framework Foundation -o plist_tester` and when run, this is the output I get:

```
$ ./plist_tester
plist_tester[] Plist Type: xml
plist_tester[] Contents:
plist_tester[]  hourglass = ⏳
plist_tester[] =======================
```

We can write the emoji in the file because XML style plists support UTF-8 encoding. Now let's do the same thing but with a NeXTSTEP plist.

```
$ cat nextstep
{
    "hourglass" = "\U23F3";
}
```

Here is the NeXTSTEP plist version of that XML plist that was first created. NeXTSTEP plists are serialized in ASCII. This means that for us to write representations of unicode characters we must instead write the escaped version of the character sequence. Now I'm going to add this plist to the program:

```
NSArray *plistTypes = @[ @"xml", @"nextstep" ];
```

Then compile and run this again and I get the resulting output:

```
$ ./plist_tester
plist_tester[] Plist Type: xml
plist_tester[] Contents:
plist_tester[]  hourglass = ⏳
plist_tester[] =======================
plist_tester[] Plist Type: nextstep
plist_tester[] Contents:
plist_tester[]  hourglass = ⏳
plist_tester[] =======================
```

Now let's try adding another emoji to the plist:

```
$ cat xml
<plist version="1.0">
<dict>
    <key>hourglass</key>
    <string>⏳</string>
    <key>panda</key>
    <string>🐼</string>
</dict>
</plist>
$ plutil -lint xml
xml: OK

$ cat nextstep
{
    "hourglass" = "\U23F3";
    "panda" = "\U0001F43C";
}
$ plutil -lint nextstep
nextstep: OK
```

Both plists have been updated to have a panda emoji. Running the program again yields this output:

```
$ ./plist_tester
plist_tester[] Plist Type: xml
plist_tester[] Contents:
plist_tester[]  hourglass = ⏳
plist_tester[]  panda = 🐼
plist_tester[] =======================
plist_tester[] Plist Type: nextstep
plist_tester[] Contents:
plist_tester[]  panda = F43C
plist_tester[]  hourglass = ⏳
plist_tester[] =======================
```

The panda emoji is not going to display from a NeXTSTEP plist because it doesn't know how to interpret escaped unicode characters that are longer than

4 hex digits. So while the hourglass can be parsed, only the `0001` of the `0001F43C` will be parsed and converted into the corresponding unicode character. The following is a section of code taken from CFOldStylePlist.c which is part of CoreFoundation's implementation of reading NeXTSTEP plists:

```
static UniChar getSlashedChar(_CFStringsFileParseInfo *pInfo) {
    UniChar ch = *(pInfo->curr);
    pInfo->curr ++;
    switch (ch) {
        case '0':
        case '1':
        case '2':
        case '3':
        case '4':
        case '5':
        case '6':
        case '7':  {
            uint8_t num = ch - '0';
            UniChar result;
            CFIndex usedCharLen;
            /* three digits maximum to avoid reading \000 followed by 5
            if ((ch = *(pInfo->curr)) >= '0' && ch <= '7') { // we use
                pInfo->curr ++;
                num = (num << 3) + ch - '0';
                if ((pInfo->curr < pInfo->end) && (ch = *(pInfo->curr))
                    pInfo->curr ++;
                    num = (num << 3) + ch - '0';
                }
            }
            CFStringEncodingBytesToUnicode(kCFStringEncodingNextStepLat
            return (usedCharLen == 1) ? result : 0;
        }
        case 'U': {
            unsigned num = 0, numDigits = 4;    /* Parse four digits */
            while (pInfo->curr < pInfo->end && numDigits--) {
                if (((ch = *(pInfo->curr)) < 128) && isxdigit(ch)) {
                    pInfo->curr ++;
                    num = (num << 4) + ((ch <= '9') ? (ch - '0') : ((ch
                }
            }
            return num;
        }
        case 'a':   return '\a';    // Note: the meaning of '\a' varies
        case 'b':   return '\b';
        case 'f':   return '\f';
        case 'n':   return '\n';
        case 'r':   return '\r';
        case 't':   return '\t';
        case 'v':   return '\v';
        case '"':   return '\"';
        case '\n':  return '\n';
    }
    return ch;
}
```

This method is used when an escaped character sequence is encountered, the intention is to translate the escaped representation from ASCII into Unicode. You will see in the case of the escaped `U` it will parse up to 4 hex digits to compose the character.

Now you may be wondering how is this possible since Xcode seems to handle all sorts of emoji. Xcode's implementation of deserializing the NeXTSTEP plist files is different from that of what is used in (Core)Foundation. There are assumptions made about what the output encoding is assumed to be, as well as supporting writing out this format of plist when (Core)Foundation does not. The NeXT/OpenStep plist format assumes that strings are written as ASCII, whereas Cocoa assumes strings are written in Unicode. As a result, Cocoa will happily read unescaped Unicode data from NeXT/OpenStep plists (while the parser will fail to read properly escaped sequences longer than 4 digits). This makes the format invalid as it is no longer ASCII data on disk, however will still be parsed correctly by classes like NSDictionary because of Cocoa's assumption that all strings are Unicode.

I wrote this post as a means to document some unusual behavior around parsing this legacy format that is still highly used today, and I have filed a radar to add more documentation around this behavior for the classes that would be used.
