Analyzing Strings with sscanf

The counterpart or "opposite" of `sprintf` is `sscanf`. It looks at a string, and extracts values from it according to another string with format specifiers. It then places the values it has extracted in its arguments. The arguments must be variables, or other things that you can assign values to. For example, after the following code segment, the variable `f` will contain **-3.6**, and the variable `i` will contain **17**:

```
float f;
int i;
string the_string = "foo -3.6 fum dum 17";
sscanf(the_string, "foo %f fum dum %d", f, i);
```

Note that `sscanf` isn't an ordinary method or built-in function, but a very special construction in Pike. The reason is that `sscanf` has to change its arguments, and since Pike uses call-by-value, no ordinary method or built-in function can do that.

The format string consists of characters, that should match the contents of the first string exactly. Blanks are significant. For example, the part `fum dum` in the format string must match a part of the analyzed string (`the_string`) exactly. The format string can also contain "percent specifiers", which match various things. For example, `%d` us used to match an integer.

Here is a list of format specifiers:

Specifier	Meaning
%d	An integer in normal decimal (that is, base-10) notation.
%o	An integer in octal (base-8) notation.
%x	An integer in hexadecimal (base-16) notation.
%D	An integer in decimal, or (if it starts with `0`) octal, or (if it starts with `0x`) hexadecimal notation. Hence, `sscanf("12", "%D", i)`, `sscanf("014", "%D", i)` and `sscanf("0xC", "%D", i)` all yield the value 12 in `i`.
%f	A floating-point number.
%c	The character code of a single character.
%s	A string. If `%s` is followed by `%d`, `%s` will read any non-numerical characters. If followed by `%[]`, `%s` will read any characters not present in the set in the `%[]`. If followed by normal text, `%s` will match all characters up to, but not including, the first occurrence of that text.
%*N*s	As above, but a string of exactly *N* characters
%[*characters*]	A string containing any of the characters in the list *`characters`*. A minus sign can be used to give a range of values, so e. g. `%[a-d]` means a string consisting of any of the characters `a`, `b` and `c`. A ^ sign means "not", so e. g. `%[^abc]` means any character except `a`, `b` and `c`. They can be combined, so `%[a-cf]` means `a`, `b`, `c`, and `f`.

%{*format*%}	Repeatedly matches the format specifier *format* as many times as possible, and gives an array of arrays with the results. Example: `%{%d%}` matches zero or more integers.
%%	A single percent (%) character

If an asterisk (*) is put between the percent and the operator, e. g. `%*d`, the operator will only match its argument, and not assign any variables.

`sscanf` returns the number of percent specifiers that were successfully matched.

The matching done by `sscanf` is rather simple-minded. It looks at the format string up to the next %, and tries to match that with the analyzed string. If successful, it then goes on to the next part of the format string. If a part of the format string does not match, `sscanf` immediately returns (with the number of percent specifiers that were successfully matched). Variables in the argument list that correspond to percent specifiers that were not successfully matched will not be changed.

Some examples of `sscanf`:

- This call to `sscanf` will return **1**, and the variable `a` will be given the value `"oo"`:

  ```
  sscanf("foo", "f%s", a);
  ```

- The return value from `sscanf` will be **2**. `a` will be given the value **4711**. `b` will be given the value `"bar"`.

  ```
  sscanf("4711bar", "%d%s", a, b);
  ```

- The return value from `sscanf` will be **1**, `a` will be given the value `"test"`:

  ```
  sscanf(" \t test", "%*[ \t]%s", a)
  ```

- This removes `"the "` from the beginning of the string in `str`. If `str` does not begin with `"the "`, it will not be changed

  ```
  sscanf(str, "the %s", str);
  ```

- This assigns `"foo"` to `s1` and `"fum"` to `s2`, and the array `({ ({ 1 }), ({ 2 }), ({ 3 }) })` to `a`. The return value will be **3**.

  ```
  sscanf("foo % 1 2 3 fum",
         "%s %% %{%d%} %s", s1, a, s2);
  ```

A feature with `sscanf` is that you can define variables inside it:

```
sscanf("foo % 1 2 3 fum",
       "%s %% %{%d%} %s",
       string s1, array(string) a, string s2);
```

In contrast to variables defined in a `for` statement, these variables are available in the rest of the block, and not just inside the `sscanf`.