Macros

New Perspectives on TEX Macros

Jonathan Fine

Abstract

Using the TEX macro language as an example, this article indicates how SGML can be used the specify the source file syntax for literate programming. (This is part of the philosophy behind the author's SIMSIM project, which will allow TEX to typeset SGML documents.) Some of the advantages are shown. The problems of implementation are not discussed.

Introduction

This article is about TEX macros, SGML and literate programming. It also explains some of the philosophy behind the author's SIMSIM package, which will provide a basis for the formatting by TEX of SGML documents. The author hopes for a first release to selected test sites by the end of 1995.

Knuth implemented literate programming by defining the WEB file format, and producing two auxiliary programs, WEAVE and TANGLE, which transform a WEB file into TEX and Pascal files respectively. This was done in the early 1980s. Today it might be better to use an SGML document type definition in the place of the WEB file format. This would allow existing and future SGML tools to process the program source code.

TANGLE can also reorder the code, so that the programmer can present the code the program in an order which suits the programmer (and reader) rather than the compiler. This is considered by its practitioners to be an essential feature of literate programming. (The author thanks the referee for pointing this out.)

Consider now TEX macros. Here is a macro definition, written in an unspecified SGML DTD.

```
<mac n=echo> <par n=Token> writes the
|Token| to the console.
    message { string Token }
</mac>
```

Its meaning should be clear. The intention is to define a macro, whose name is `echo`. It takes a single parameter, which the author is calling a Token. The replacement text is given by the lines in the `<mac>` element that begin with a leading space. Notice that there are no backslashes. Instead, the character string `message` is standing for the control sequence whose name is `message` (and which is usually referred to by `\message`). As usual, `{` and `}` stand for characters with category code 1 and 2 respectively. Finally, `Token` stands for `#1`, as `Token` was the first parameter to be declared.

Compare this definition to the text

```
\def\echo#1{\message{\string#1}}
```

that would be used in an ordinary macro file to express the same meaning.

Here is another example.

```
<mac n=gobble> <discard> takes a
token (or balanced list) and
throws it away.
</mac>
```

which has empty replacement text. Here

```
\def\gobble#1{}
```

is the ordinary form for this definition.

The SGML form requires more effort to write, but that is because it is more expressive. For complicated macros, it is a great help, to have named rather than numbered macro parameters.

More examples

The benefits of the SGML approach grow, the larger and more complicated the macros are. Here is an example. (The closing `</mac>` is to be understood. The SGML omitted end tags feature will supply it if the next tag is also a `<mac>`, or any other element that cannot occur within a `<mac>` element. In the same way, the short reference feature can recognise a leading blank and within the `<mac>` element, translate it into `<code-line>`. Similarly, within `<code-line>` the carriage return can be translated into the end tag `</code-line>`.)

```
<mac n=show> <par n=Token> is like
the \show primitive of &TeX except
that it is not like an error message.
    immediate write 16
    {
      > ~~~ string Token =
        meaning Token
    }
```

The ~ stands for an ordinary space character. Dirty tricks are required to get a sequence of such characters into the replacement text of a macro. TEX runs more efficiently if numeric constants such as `16` are replaced by tokens that have been `\chardef`'d to the appropriate value. It is much easier to write (and read) `16` than it is the control sequence `\sixt@@n` that is used in the source file for `plain` and LATEX. The characters `>` and `=` stand for themselves, as 'other' characters.

The replacement text of a TEX macro is a sequence of tokens. The syntax and semantics of the source code file format should allow the programmer to specify, perhaps implicitly, the sequence of tokens desired. The analog of TANGLE should produce a file from which TEX can produce (at high speed) the specified macro definition. The technical means to accomplish this are not discussed in this article. Suffice to say that everything described here is known to be possible.

From SGML tags to TEX actions

As an SGML document is parsed, information becomes available to the text processing application. Typesetting (or any other processing) of an SGML document consists of linking actions to the start and end tags, and to other events. Suppose the document to be processed has an element called `<tag-name>`, with an attribute called `text`. The code below

```
1. <gi n=tag-name> This tag has a text
2. attribute, whose value will be typeset
3. in a box.
4.     begingroup
5.       //  some code is omitted
6.       let end-element endgroup
7.       hbox { (tag-name*text) }
```

specifies processing for such an element.

Line 4 tells us that once the tag has been parsed, a group is begun. Line 5 is a comment. Line 6 says that `\endgoup` is the action to be performed when the element comes to an end. Note that because SGML allows hyphens, periods and digits to occur in a name, it is convenient to allow the same for control sequences. Incidentally, it is much easier to type and to read a hyphen, than it is an underscore.

It is line 7 that sets the value of the text attribute in a horizontal box. The sequence of characters

```
        (tag-name*text)
```

stands for a single token, whose expansion will be the current value of the `text` attribute of the `<tag-name>` element. Thus, the text

```
    <tag-name text="This and that">
```

will cause the SGML parser to define the token referred to by

```
        (tag-name*text)
```

to be a macro whose expansion is the sequence

```
        This and that
```

of letters. Just quite what that token is, should be of no concern to the programmer. Indeed, it should not be possible for the programmer to access this token, except throught the `(tag*att)` construct.

In the same way

```
    <ent n=TeX> typesets the &TeX logo.
        'T kern <dim v=-.1667em>
        lower <dim v=.5ex> hbox { 'E }
        kern <dim v=-.125em> 'X
```

specifies the action to be linked to the SGML entity `&TeX`. By way of explanation, the right quote `'` is an escape character. Thus, `'T` stands for a letter `T` with (for technical reasons) category code 'other'. The `<dim>` element in the macro definition should be translated, by the `TANGLE` equivalent, to an appropriate quantity. So long as the semantics are well defined, the translation can be made.

Conclusion

In the humanities, it is becoming more widely accepted that structured documents should be stored with a rigorous syntax, and that SGML provides a means of specifying that syntax. In addition, a growing collection of SGML software tools are becoming available.

Literate programming (which if not in the humanities is at least an art) also requires a rigorous document syntax. There is a strong case for using SGML in this context also. For this to succeed, there must be available suitable typesetting tools, that will accept SGML documents. The author's SIMSIM project is intended to provide such.

◇ Jonathan Fine
203 Coldhams Lane
Cambridge CB1 3HY
UK
Email: `J.Fine@pmms.cam.ac.uk`