HiTex

LaTeX gets a helping hand from Forth

Bill Stoddart

September 17, 2014

Abstract

HiTeX is a simple LaTeX pre-processor that works through token replacement. It provides improved readability of mathematical text in a source document by allowing free use of Unicode characters and eliminating any need for specific spacing and new line commands. HiTeX gains considerable power from the ability to incorporate sections of Forth text within a document. Output generated by Forth can be directed to the output file, or can be used to define place-holders which, when used within maths mode in a HiTeX document, will be replaced by the result of the corresponding computation.

Keywords: LaTeX, Unicode, Computable Document, RVM-Forth

1 Introduction

LaTeX is a versatile type setting system that gives excellent results on both mathematical and normal text. However, the mathematical markup is not always easy to read *as mathematics*. The advent of Unicode should have improved this, allowing us to write, for example, $\sqrt{\alpha}$ instead of the standard latex markup `\sqrt\alpha`. However, Unicode and its utf8 encoding have only partially been adopted by the LaTeX community, with the promising ucs package left unmaintained and unfinished. The projects XeLaTex and LauTeX are complete reworkings of TeX and Latex which are based from the outset on Unicode utf8 input. Our research group produced some papers in XeLaTex, but it was not a happy experience. One problem is that journal

editors and submission portals may not accept documents written with these tools. We also had a problem with Greek characters, due to the fact that a font suitable for publishing an article written in Demotic Greek will not be suitable for providing the Greek letters used in mathematics. Also, we felt that the availability of Unicode should make the markup language sufficiently compact that it would be possible to revise the LaTeX practice of ignoring white space and requiring specific markups for additional space and new lines. We wanted a markup language where spaces and newlines would, by default, be taken into account in the final markup.

It also seemed to us to us that, rather than completely rewrite TeX and LaTeX, which are absolutely brilliant as they are, it would be better to write a simple pre-processor to translate a Unicode mathematical language into classical LaTeX. The result is HiTex. The last page of this document gives an example of HiteX markup and the resulting output.

HiTex is a hybrid of Forth and Latex which has its own variant of the LaTeX mathematical markup language. A HiTeX document contains 3 types of text. Initially, it is in pass-through mode, in which text is just streamed from the input file to the output file. All the HiTeX interpreter is doing at this time is checking for tokens that will take it either into a mathematical mode or into Forth.

Within a mathematical mode, HiTeX performs token replacements, recognising tokens in the HiTeX source, and replacing them by a corresponding token in the LaTeX output file. A token can be any sequence of characters. Some of the tokens are Unicode characters, such as ∀, ∃, *dot* etc, which are replaced by their corresponding LaTeX markups, `\forall, \exists, \bullet`. However, a token can also be something like a new line character, a space, or a sequence of spaces. Where one token is the prefix of another (for example a token consisting of two spaces would be a prefix of a token consisting of three spaces) the longer token is matched first. This ensures a correct match for all tokens.

HiTeX is implemented in RVM-Forth and uses Frank Zeyda'a set package, (see EuroForth 2002 proceedings), which supports arbitrary finite homogeneous sets. We use ascii zero format strings.

The corresponding pairs of tokens used by HiTeX are held in the set `LaTeX-MARKUP`. Here is the beginning of its definition:

```
STRING  STRING  PAIR  { " ∀"   \forall"  ↦ ,
                        " ∃"   " \exist " ↦ ,   ...
```

Text before the opening brace gives the type information required to construct the set. The set consists of pairs of strings. The maplet operator $\mapsto$ combines two strings on the stack into an ordered pair of strings. The following comma compiles this element into the set. The set construction is terminated by a closing brace, at which point the set (i.e. a pointer to the data structure which represents the set) is left on the stack

Within a Forth section a user can add new markups using set union $\cup$ or remove markups using domain subtraction `<<|`.

2 Including computation results in a document, an integer maths example

A interesting case is where the token to be inserted in a document is produced by a Forth computation. To define a token that captures an integer result, we can use the defining word `n†`. Here is an example of its use.

```
1234 n† s
```

This defines a new dictionary entry `s` which, when executed, gives the address of an asciiz string containing the text "1234". We adopt a naming convention that strings generated in this way that will subsequently be used as tokens will be given a name that *begins* with †. Words that create such tokens have names that end in †.

We look at a simple example where we add the values of two constants and display the original values and their sum in a document.

2.1 Source code of the supporting Forth section

In the following Forth section the definitions $\dagger\alpha$, $\dagger\beta$ and $\dagger\alpha{+}\beta$ will return asciiz strings containing the text "10", "20" and "30" respectively. Let us suppose that these are the numeric strings that are to be placed in the LaTeX output in response to seeing $\dagger\alpha$, $\dagger\beta$ or $\dagger\alpha{+}\beta$ respectively in the HiTeX source document. The tokens are paired up in a set, which is combined with `LaTeX-MARKUP` using set union. The updates are disseminated to the requisite HiTeX data structures with the `CONFIG` command.

```
%FORTH
10 CONSTANT α  20 CONSTANT β

α n† †α  β n† †β  α β + n† †α+β

STRING STRING PROD { " †α" †α ↦ ,
" †β" †β  ↦ , " †α+β" †α+β ↦ , }

LaTeX-MARKUP ∪ to LaTeX-MARKUP CONFIG END
```

Here is how these tokens can be used in a HiTeX math environment, along with the result.

HiTex markup	Final output
`$α=†α,  β=†β,  α+β=†α+β$`	$\alpha = 10,\ \beta = 20,\ \alpha + \beta = 30$

3 A floating point example

Floating point results are captured in a similar way, but using the defining word `f†` to define the output tokens. After the first line of Forth code the definition †√2 returns the address of an asciiz string representing $\sqrt{2}$ to 6 decimal places (our default output precision).

3.1 The supporting Forth section

```
%FORTH 2. FSQRT f† †√2   3. 2. F/ FSQRT f† †√(3/2)

STRING STRING PROD { " †√2" †√2 ↦ ,
" †√(3/2)" †√(3/2) ↦ ,  " √" " \sqrt " ↦ , }

LaTeX-MARKUP ∪ to LaTeX-MARKUP CONFIG END
```

And here is an example of HiTeX markup and the resulting final output.

HiTex markup

```
\[
√2=†√2
√«3/2»=†√(3/2)
\]
```

Final output

$$\sqrt{2} = 1.41421$$
$$\sqrt{3/2} = 1.22474x$$

4 Configuraton tasks

A Forth section can be used for general configuration tasks, both of the HiTeX application and of the underlying Forth system.

In the example above, French «guillemets »were used as HiTex scope delimiters. These are preferred to the standard tex/latex delimiters `{` and `}`, as we use the latter as set brackets, and consider them to be essential mathematical symbols.

HiTeX holds its scope delimiters in the `VALUE`s `{SCOPE` and `SCOPE}` .

The following Forth section shows how we change these delimiters to Unicode bold brackets.

We also change the precision of the floating point output, recalculate the string produced by printing $\sqrt{3/2}$, update our markups, and reconfigure.

4.1 The supporting Forth section

```
%FORTH  " ❨"  to  {SCOPE     " ❩"  to  SCOPE}

( Regenerate the result for √(3/2) at higher precision)
8  SET-PRECISION   3.  2.  F/  FSQRT  f†  †√(3/2)

( remove the previous entry for the placeholder " †√(3/2)")
STRING {  " †√(3/2)"  , }  LaTeX-MARKUP  <<|

( Add the new entry )
STRING STRING PROD { " †√(3/2)"   †√(3/2)  ↦  ,  }   ∪
to  LaTeX-MARKUP  CONFIG  END
```

Now our markup for $\sqrt{3/2}$ and the corresponding output are as follows

HiTex markup

```
\[ √❨3/2❩ = †√(3/2) \]
```

Final output

$$\sqrt{3/2} = 1.2247449$$

5 Implementation note 1

The defining words n† and f† have a lot in common, and both are defined in terms of a more primitive word P2† as follows:

```
: P2† ( ? xt "<spaces><name>"-- ; exec: -- az,  Creates <name>.
On execution <name> will return the address of an az string
consisting of the text output by the execution of xt )
  CREATE  'EMIT @ PUSH  ['] C, 'EMIT !
     EXECUTE  0 EMIT
  POP 'EMIT ! ;

: n† ['] . P2† ;   : f† ['] F. P2† ;
```

P2† takes an execution token fromthe stack, plus whatever extra parameters are required for the token's execution. It CREATEs a new dictionary entry and vectors EMIT to compile its output into the dictionary. It executes xt, and restores EMIT

6 A more general example

.

We provide for an arbitrary section of Forth source code to produce output, which we assume will be in the HiTeX markup format, rather than in Latex. This output must therefore be processed by the HiTeX maths pre-processor before being inserted in the LaTeX document. This is done with the pair of words [: ... :]. For example, suppose A .SET gives the output {1,2,3} This is not suitable to be immediately passed into the output document, since LaTeX will not see the braces as set delimiters, but as scope delimiters, and they will not appear on the final output. The phrase [: A .SET :] †A creates the Forth word †A which returns the address of the string obtained by passing the text output by the Forth between [: and :] through the HiTeX math pre-processor. Thus this defines †A as the string " \{1,2,3\}", which is the correct LaTeX markup for the value of set A.

6.1 The supporting Forth section

```
%FORTH
INT { 1 , 2 , 3 , } CONSTANT A   INT { 2 , 3 , 4 , } CONSTANT B

 [: A .SET :] †A

[: B .SET :] †B

[: A B ∪ .SET :] †A∪B
[: A B ∩ .SET :] †A∩B
[: A B \ .SET :] †A\B

STRING STRING PROD { " †A" †A ↦ ,  " †B" †B ↦ ,  " †A∪B" †A∪B ↦ ,
" †A∩B" †A∩B ↦ ,  " †A\B" †A\B ↦ , }
LaTeX-MARKUP ∪ to LaTeX-MARKUP CONFIG  END
```

HiTeX markup

```
\[
A = †A
B = †B
A∪B = †A∪B
A∩B = †A∩B
A\B = †A\B \]
```

Final output

$$A = \{1, 2, 3\}$$
$$B = \{2, 3, 4\}$$
$$A \cup B = \{1, 2, 3, 4\}$$
$$A \cap B = \{2, 3\}$$
$$A \setminus B = \{1\}$$

7 Implementation note 2

HiTeX reads a source file into an input buffer, and places its LaTeX output in an output buffer. An asciiz string computed within a Forth section, and whose address is on the top of the stack, can be sent directly to the output buffer with the phrase:

```
DUP AZLENGTH TO-OUTBUFF
```

The outermost HiTeX interpreter passes text from the input buffer to the output buffer until it encounters a token that causes it to enter either Math mode, or Forth. The mathmode interpreter checks at each point in the input buffer whether the following characters match one its tokens. These tokens are those from the domain of `LaTeX-MARKUP` plus other tokens that require special action. If the token is from the domain of `LaTeX-MARKUP` the

corresponding token from the range of LaTeX-MARKUP is added to the output buffer. Other tokens are special cases which require additional action. For example, a new line character in the input buffere requires a line count to be incremented, and the new line itself plus the LaTeX markup for a newline must be passed to the output buffer.

The input and output buffers are managed by a collection of VALUEs holding buffer start addresses, pointers to the current position in each buffer, etc. When text generated within a Forth section is to be processed by the HiTeX maths pre-processor, e.g. when using a [: . :]. structure, these buffer management values are saved, and the pointers etc are set to work from temporary buffers. After the text is processed, the resulting LaTeX markup is compiled into the dictionary and the temporary buffers are free for future use.

We return to the point of distinguishing between tokens such as †A and †A+B. The first of these tokens matches the start of the second, i.e. the first token is a prefix of the second. How do we ensure that †A+B won't be mistaken for †A?

We do this by searching for tokens in the same order as they occur in a sequence. We place our tokens in a sequence in such a way that any token that has prefixes that are also tokens will occur before its prefixes in the sequence. Reverse lexical order will achieve this.

The properties of our set implementation and the reversible features of RVM-Forth make this simple to implement. Every set is held as an ordered set, and the CHOICE operator selects the maximal element of each set, or if invoked after backtracking will select the maximal element not yet chosen.

For strings the ordering is lexical. Thus "†A" comes before "†A+B''

We can create a sequence in which tokens in the domain of LaTeX-MARKUP occur in reverse lexical order using the following code:

```
LaTeX-MARKUP  DOM  SET2SEQ
```

Where the definition of SET2SEQ is:

```
: SET2SEQ ( x:P(X) -- y:seq(X), ran(y)=x )  (: set :)
   set [ <RUN set CHOICE RUN> ] ;
```

In this code the square brackets enclose a sequence construction. (They are not the Forth Standard square brackets). The set before the open square

bracket provides type information. The code bracketed by `<RUN ... RUN>` chooses an element of `set` and compiles it as the next element sequence. Execution then reverses back to `CHOICE`, which makes a different choice if one is available, and this is then added to the sequence. This is repeated until no further choices are available, at which point execution continues beyond `]` The result is a sequence of strings held in reverse lexical order. This code is based on the premise that our sets are ordered; we know how but we can't control how. But the order of elements in a sequence is entirely under programmer control.

8 Conclusions and Future Work

HiTeX has been very valuable to us for writing dense mathematical documents. Its main limitation is that it does not support a verbatim mode which accepts Unicode - that's why we have used screen shots for the most of the Forth source code and HiTeX markup examples in this document.

Appendices

A HiTeX markup example

```
{ ρ | ρ ∈ ℰ ∧
   ∀x'.(x' ∈ choice(⟦s⟧^ν(ρ)) ⇒
      { ρ' | ρ' ∈ {ρ ⊕ ⦉ x ⇝ {x'} ⦊} ∧
         ⟦x⟧^ν(ρ ⊕ ⦉ x ⇝ {x'} ⦊) ⊆ choice(⟦t⟧^ν(ρ))
      } ≠ {}
   )
}
      ∩
{ ρ | ρ ∈ ℰ ∧
   ∀x'.(x' ∈ choice(⟦t⟧^ν(ρ)) ⇒
      { ρ' | ρ' ∈ {ρ ⊕ ⦉ x ⇝ {x'} ⦊} ∧
         ⟦x⟧^ν(ρ ⊕ ⦉ x ⇝ {x'} ⦊) ⊆ choice(⟦s⟧^ν(ρ))
      } ≠ {}
   )
}
```

$$
\begin{array}{l}
\{\,\rho \mid \rho \in \mathcal{E} \wedge \\
\quad \forall\, x'.(x' \in \mathit{choice}([\![s]\!]^{\nu}(\rho)) \Rightarrow \\
\quad\quad \{\,\rho' \mid \rho' \in \{\rho \oplus (\!|\, x \rightsquigarrow \{x'\} \,|\!)\} \wedge \\
\quad\quad\quad [\![x]\!]^{\nu}(\rho \oplus (\!|\, x \rightsquigarrow \{x'\} \,|\!)) \subseteq \mathit{choice}([\![t]\!]^{\nu}(\rho)) \\
\quad\quad \} \neq \{\} \\
\quad) \\
\} \\
\quad\quad \cap \\
\{\,\rho \mid \rho \in \mathcal{E} \wedge \\
\quad \forall\, x'.(x' \in \mathit{choice}([\![t]\!]^{\nu}(\rho)) \Rightarrow \\
\quad\quad \{\,\rho' \mid \rho' \in \{\rho \oplus (\!|\, x \rightsquigarrow \{x'\} \,|\!)\} \wedge \\
\quad\quad\quad [\![x]\!]^{\nu}(\rho \oplus (\!|\, x \rightsquigarrow \{x'\} \,|\!)) \subseteq \mathit{choice}([\![s]\!]^{\nu}(\rho)) \\
\quad\quad \} \neq \{\} \\
\quad) \\
\}
\end{array}
$$