MiniLatex: a Parser-Renderer for a Subset of LaTeX

JAMES CARLSON, University of Utah, USA

MiniLatex is (1) a subset of LaTeX, (2) a parse-render pipeline which transforms documents written in MiniLatex to HTML. The strategy is to construct an abstract syntax tree (AST) by parsing source text, then render the AST to HTML, with math-mode elements passed on verbatim for processing by MathJax. The parse-render pipeline is a library with a small API written using the `elm/parser` combinator package of Elm, a statically typed functional language that is designed for building front-end web apps. Both the AST and the parser are quite small: 12 and 334 lines of code, respectively. A (context-sensitive) grammar for MiniLatex is sketched. Performance of the system is quite good: fast enough for real-time editing. Three apps have been using built using the MiniLatex library. The, first MiniLatex Live, is a small app (338 lines of code) with a simple editor featuring two windows, with source text on the left, rendered text on the right, and a field for entering math-mode macro definitions.. The rendered text is updated every 250 milliseconds, so that changes are "immediately" reflected in the rendered version. The second, MiniLatex Reader, is even smaller. It is a read-only app that can be embedded in web pages to display content written in MiniLatex. The third, knode.io, is a full content-management app with a searchable document store and facilities for uploading images for inclusion in MiniLatex documents.

CCS Concepts: • **Software and its engineering → Functional languages**; **Abstract data types**; **Markup languages**;

Additional Key Words and Phrases: LaTeX, HTML, Elm, parser combinator, MathJax, parse, render, abstract syntax tree

ACM Reference Format:
James Carlson. 2666. MiniLatex: a Parser-Renderer for a Subset of LaTeX. *ACM Trans. Web* 0, 0, Article 0 (January 2666), 10 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

Thanks to the pioneering work of Donald Knuth [10] in creating TeX, the subsequent development of LaTeX by Leslie Lamport [11], and the continued contributions of an active community of developers, those who rely on LaTeX to publish in print or as PDF have superb document creation tools. For publishing documents with mathematical content on the web, there are a number of options. Most widely used is MathJax [6], which provides high-quality rendering of math-mode TeX-LaTeX for HTML pages. There are also a number of command-line tools, e.g., Pandoc [13], for converting LaTeX documents to HTML using embedded images for the mathematical text. Of particular note is Daan Leijen's Madoko [12], which is perhaps closest in spirit to this project. Madoko uses Koka, a typed functional language to parse and render a markdown-like language that handles mathematical formulas written in LaTeX and which can export documents to LaTeX. The gap that MiniLatex aims to fill is to provide a tool for live-rendering both text-mode and math-mode LaTeX.

Author's address: James Carlson, University of Utah, 155 South 1400 East, JWB 233, Salt Lake City, UT, 83105, USA, jxxcarlson@gmail.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2666 Copyright held by the owner/author(s). Publication rights licensed to ACM.
1559-1131/2666/1-ART0 $15.00
https://doi.org/0000001.0000001

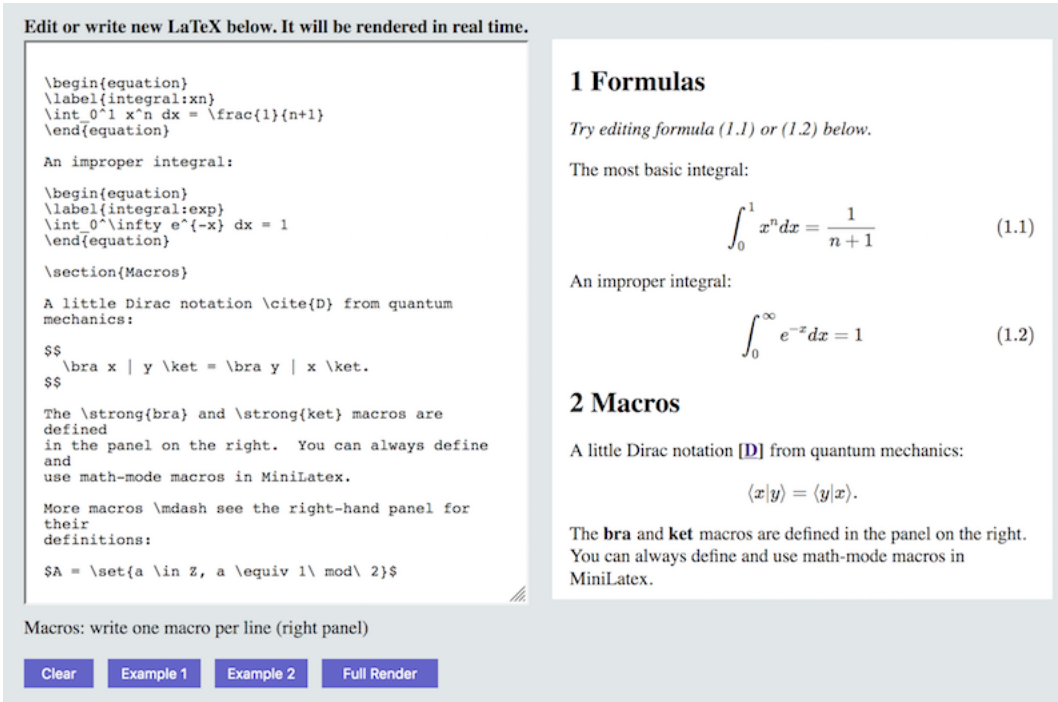

Fig. 1. MiniLatex Live.

The overall strategy for MiniLatex is to parse LaTeX source text, producing an abstract syntax tree (AST), then render the AST into HTML, with the math-mode elements passed on verbatim for rendering by MathJax. This strategy works well enough so that one can do live editing and rendering, as illustrated in Figure 1 below and as demonstrated by the MiniLatex Live app [3]. Changes in the source text window on the left are rendered in window on the right, essentially instantaneously. For small documents, the raw speed of the system is sufficient. For large documents, a diffing strategy is used to parse, render, and update only the parts of the document that have changed. Thus even large documents may be edited with ease, while still offering live rendering. We note that there is also a very light-weight "read-only" app, MiniLatex Reader app [4], for displaying LaTeX source text which can be embedded in web pages.

MiniLatex is still a research project, with many improvements and extensions of scope yet to come. Nevertheless, it is sufficiently mature to be used for real work, e.g., the production of lecture notes. The document in Figure 2 is an example of this. It is one section in a set of lectures on quantum mechanics, available at knode.io/427 [1]. Documents on `knode.io` are referenced by numerical ID, a feature which makes them easy to share.

The parse-render pipeline, the demo app, and the content management system front end are written in Elm [8], the statically typed functional programming language for developing front-end web apps created by Evan Czaplicki. Elm grew out of Czaplicki's undergraduate thesis [7]. As of this writing, it is at verson 0.19. All components of the project are open-source and available on-line [2], [3].

In the body of this article, we discuss the core technology: parser, renderer, and some optimizations.

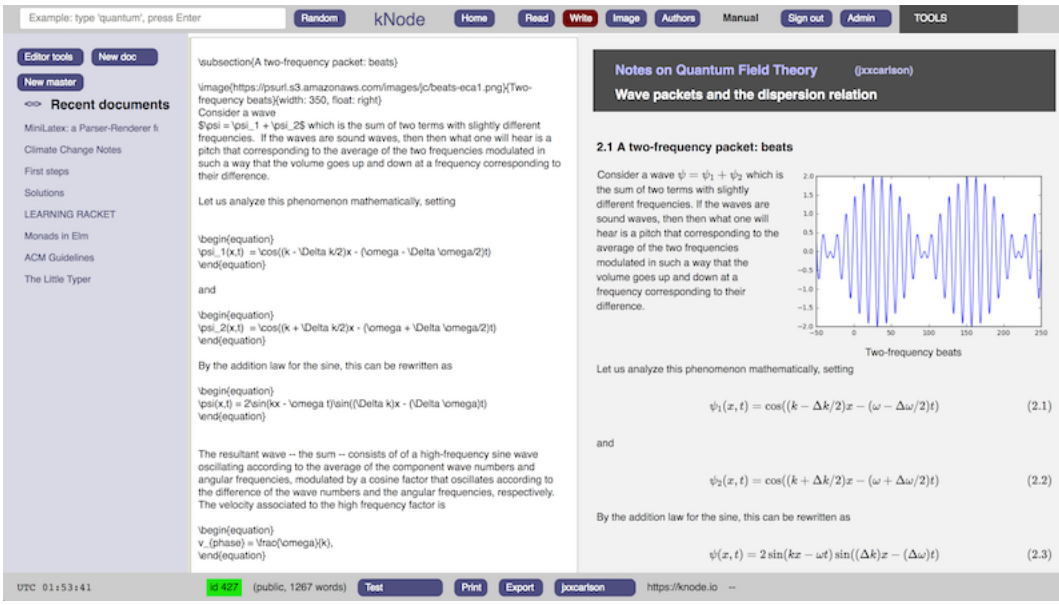

Fig. 2. knode.io/427.

2 AST

The heart of the MiniLatex system is the type definition of its abstract syntax tree. It is a twelve-line recursive algebraic data type:

```
type LatexExpr
    = LXString String
    | Comment String
    | Item LatexExpr
    | InlineMath String
    | DisplayMath String
    | Macro String (List LatexExpr) (List LatexExpr)
    | SMacro String (List LatexExpr) (List LatexExpr) LatexExpr
    | Environment String (List LatexExpr) LatexExpr
    | LatexList (List LatexExpr)
    | NewCommand String Int LatexExpr
    | LXError (List DeadEnd)
```

A value of type `LXString String` is used to represent prose, with the string "Hello there!" parsed as `LXString ("Hello there!")`. `Comment String` represents strings with a leading percent sign. The `Item LatexExpr` type is used for items in the enumerable and itemize environments, while `InlineMath String` and `DisplayMath String` are for constructs of the form $... $ and $$... $$. The `Macro ...` type is for macros, e.g. `\emph{Yes!}`. A macro has name, a list of optional arguments, and then a list of normal arguments. The first are enclosed in brackets, the second in curly braces. The `SMacro` type is similar to the `Macro` type, except that there is an additional, final argument which is terminated by two or more newlines. The `Environment` type captures the name of the environment, a possibly empty list of optional arguments, and the body of the environment, which is an arbitrary `LatexExpr`. The `LatexList` type is used to parse paragraphs, which typically

yield a sequence of `LatexExprs`. The `NewCommand String Int LatexExpr` type is used to hold user-defined macro definitions. Finally, there is an error type, which is used to capture and display information on parse errors. The error messages are already reasonably good. For example, if one omits the final curly brace in a macro, a red "Expecting symbol: }" message is displayed *in situ*. We plan to make the error messages much, much, better using the Advanced mode of the `elm/parser` library. The feasibility of this has already been demonstrated by the high-quality error messages emitted by the Elm compiler.

3 GRAMMAR

The grammar for MiniLatex consists of a small set of productions. We give some examples here. First is the top-level production, which can be read straight from the definition of the AST:

$$\text{LatexExpr} \rightarrow \text{LXString} \mid \text{Comment} \mid \text{Item} \mid \text{InlineMath} \mid \text{DisplayMath}$$

$$\text{Macro} \mid \text{SMacro} \mid \text{NewCommand} \mid \text{Environment} \mid \text{LatexList} \mid \text{LXError}$$

The production for inline math elements is

$$\text{InLineMath} \rightarrow \$\ mathSymbols^{+}\ \$$$

and the production for macros is equally straightforward:

$$\text{Macro} \rightarrow \text{MacroName OptionalArg}^{*}\ \text{Arg}^{*}\ \text{WS}$$

Here `MacroName` is an identifier, and `WS` is whitespace. The production for environments is somewhat more complex:

$$\text{Env envName} \rightarrow \texttt{\textbackslash begin\{envName\}}\ \text{LatexExpr}\ \texttt{\textbackslash end\{envName\}}$$

This last production is different from all the others in that the left-hand side contains both a nonterminal symbol `Env` and a terminal symbol, `envName`, e.g., "theorem", "corollary", "verbatim", etc. For this reason the MiniLatex grammar is context-sensitive but not context-free. Such grammars cannot be recognized by a pushdown automaton, but can be recognized by a linear bounded Turing machine: one for which the length of the working tape is bounded by a constant times the length of the input tape.

4 PARSER

In this section, we discuss the combinators in the `elm/parser` library [9]. Notable are the pipeline combinators `(|.)` and `(|=)` discussed below.

The parser is built from elementary parsers that recognize fixed strings, e.g., `\begin{theorem}`, whitespace, words, etc. Words are defined as strings not containing whitespace and not beginning with a reserved character such as a backslash or a dollar sign. There are also special parsers such as

$$\text{succeed} : a \rightarrow \textit{Parser}\ a$$

which always succeeds and returns a value of type a. From given parsers one may construct others using combinators. Thus the top level parser is constructed using the `oneOf` parser function, which takes a list of parsers as input, applies each in turn, returning on the first parser to succeed, otherwise finishing with an error.

```
LatexExpr : Parser LatexExpr
LatexExpr =
    oneOf
        [ texComment
        , displayMathDollar
```

```
        , displayMathBrackets
        , inlineMath ws
        , newcommand
        , macro ws
        , smacro
        , words
        , lazy (\_ -> environment)
        ]
```

Notice the close correspondence between the production for the nonterminal symbol `LatexExpr` and the construction of the parser: reading the right-hand side of the first is like reading the argument to `oneOf` from top to bottom.

The parser for macros is listed below. Again, the right-hand side of the production read left to right corresponds to the "body" of the parser read from top to bottom. In this case, instead of using `oneOf` for alternation, one uses a parser pipeline for sequencing. The combinator `|.` means "ignore what is parsed by the parser on its right-hand side," while the combinator `|=` means "keep what is parsed by the parser on its right-hand side." In this parser pipeline, `macroName` recognizes strings of the form "backslash followed by an identifier" and returns the identifier. The next two parsers recognize (possibly empty) lists of arguments , and `wsParser` parses whitespace.

```
macro : Parser () -> Parser LatexExpr
macro wsParser =
    succeed Macro
        |= macroName
        |= itemList optionalArg
        |= itemList arg
        |. wsParser
```

Here are the formal definitions of the pipeline combinators:

```
(|.) : Parser keep -> Parser ignore -> Parser keep
(|=) : Parser (a -> b) -> Parser a -> Parser b
```

With these in hand, one can verify that `macro` has type `Parser Macro`.

Finally, consider the parser for environments, for which we give a simplified treatment.

```
environment : Parser LatexExpr
environment =
    envName |> andThen environmentOfType
```

The `envName` parser recognizes a string like `\begin{theorem}` and in this case returns the string "theorem". The `environmentOfType` parser takes the string "theorem" as input, parses the body of the environment, and then parses the phrase `\end{theorem}`.

```
environmentOfType : String -> Parser LatexExpr
environmentOfType envType =
  let
    theEndWord = "\\end{" ++ envType ++ "}"
  in
    environmentParse theEndWord envType
```

To glue the `envName` and `environmentOfType` parsers together, one uses another sequencing combinator:

```
andThen : (a -> Parser b) -> Parser a -> Parser b
```

This combinator is, up to a permutation of arguments, the monadic bind operator for the `Parser` functor:

```
(>>=) : Parser a -> (a -> Parser b) ->  Parser b
```

The rest of the parser is constructed in a like fashion.

5 RENDERER

Construction of the renderer is straightforward. One defines a rendering function for each nonterminal symbol in the grammar; this set of mutually recursive functions teams up to render the AST. As an example, the top level rendering function is indicated below. Its form is governed by the production for `LatexExpr`, with one clause of the `case` statement for each term on the right-hand side of the production.

```
render : LatexState -> LatexExpr -> Html msg
render latexState LatexExpr =
    case LatexExpr of
          Macro name optArgs args ->
            renderMacro latexState name optArgs args

        InlineMath str ->
            Html.span [] [ oneSpace, inlineMathText str ]

        Environment name args body ->
            renderEnvironment latexState name args body
      ...
```

There are a number of observations to be made. First is that the rendering function takes an as-yet-unexplained argument, `LatexState`. This is a record that contains information about section numbers, cross-references, etc., that is needed to render LaTeX source text in the manner to which we are accustomed. Its computation will be discussed later. Second, observe that parsed text is not rendered to HTML, but rather to `Html msg`, which is Elm's native data type for representing HTML. Values of this type are efficiently inserted into the web browser's domain object model (DOM) by the Elm runtime. Third, for a complete explanation, one must follow each of the rendering functions down the rabbit hole of function calls until one arrives at a result of type `Html msg`. Let us take the `renderMacro` function, listed below, as an example. If the macro were

```
\emph{Awesome!}
```

then `name` would be `emph`, the first list of `LatexExprs`, representing optional arguments would be empty, and the last list would consist of a single element with payload `Awesome!`. The `renderMacro` function looks up the name `emph` in a dictionary whose keys are macro names and whose values are rendering functions. In this case, the lookup succeeds and the resulting function `f` is applied to its arguments, computing thereby the needed value of type `Html msg`. If the dictionary lookup fails, as it would for

```
\foo{bar},
```

the macro is rendered verbatim, but in red, thus providing a reasonable *in situ* error message.

```
renderMacro : LatexState -> String -> List LatexExpr
              -> List LatexExpr -> Html msg
renderMacro latexState name optArgs args =
    case Dict.get name renderMacroDict of
        Just f ->
```

```
            f latexState optArgs args

        Nothing ->
            reproduceMacro name latexState optArgs args
```

A similar strategy is used for rendering environments.

6 COMPUTING THE LATEXSTATE

To compute the `LatexState`, we use the notion of an Accumulator, which has the following type:

```
Accumulator state a b : state -> List a -> (state, List b)
```

An accumulator takes an initial `state` and and a list of a's as inputs and produces an updated state and a list of b's as output. An accumulator is constructed from a fold and a special kind of reducer, a function of type

```
Reducer state a b = a -> (state, List b) -> (state, List b)
```

The reducer takes as input a value of type `a` and a pair consisting of a `state` and a list of `b's`. It then produces an updated `state` and a new list of `b's`. Here is the accumulator which computes the `LatexState`:

```
Accumulator.parse :
    LatexState
    -> List String
    -> ( LatexState, List (List LatexExpr) )
Accumulator.parse latexState paragraphs =
    paragraphs
        |> List.foldl parseReducer ( latexState, [] )
```

The reducer on which it depends is defined below. It operates by first parsing the `inputString`, which is a paragraph of LaTeX source text. The result, `parsedInput`, is a list of `LatexExpr`. Next, a second reducer, `LatexStateReducer`, is applied to the parsed input and the current `LatexState` to compute a new `LatexState`. Finally, a tuple is returned. Its first element is the new `LatexState`. Its second element is obtained by appending the parsed input to the old list of lists of `LatexExpr` .

```
parseReducer :
    String
    -> ( LatexState, List (List LatexExpr) )
    -> ( LatexState, List (List LatexExpr) )
parseReducer inputString ( latexState, listListLatexExpr ) =
    let
        parsedInput =
            Parser.parse inputString
        newLatexState =
            latexStateReducer parsedInput latexState
    in
    ( newLatexState, listListLatexExpr ++ [ parsedInput ] )
```

It remains to explain the the `latexStateReducer`, which takes a list of `LatexExprs` and a `LatexState` as input and produces a new `LatexState` as output. This is accomplished using a fold and yet another reducer, as listed below.

```
latexStateReducer : List LatexExpr -> LatexState -> LatexState
latexStateReducer list state =
  List.foldr latexStateReducerAux state list
```

The real work is done by `latexStateReducerAux`, which has signature

```
LatexExpr -> LatexState -> LatexState
```

It functions very much as does the render function, operating via a case statement with one clause for each relevant type in the AST to dispatch to a reducer for that type:

```
latexStateReducerAux : LatexExpr -> LatexState -> LatexState
latexStateReducerAux lexpr state =
  case lexpr of
    Macro name optionalArgs args ->
       macroReducer name optionalArgs args state
    SMacro name optionalArgs args LatexExpr ->
       smacroReducer name optionalArgs args LatexExpr state
    NewCommand name nArgs body ->
       setMacroDefinition name body state
    Environment name optionalArgs body ->
       envReducer name optionalArgs body state
    LatexList list -> List.foldr latexStateReducerAux state list
     _ -> state
```

7 CHUNKING BEFORE PARSING

MiniLatex does not parse the entire source text in one go. Rather, it first chunks the source into a list of *logical paragraphs*. These are either ordinary paragraphs or an outer begin-end pair. Each chunk is parsed separately. This decision has two beneficial consequences. The first is that errors are localized — they cannot extend beyond one logical paragraph. It also means that the parser is of type $LL(N)$ rather than $LL(\infty)$, where N is the maximum number of characters in a paragraph.

The second advantage is that one can employ a differential rendering strategy. Let u be the list of logical paragraphs corresponding to some source text, and let v be the list for an edited version of the source text. Write these lists as $u = a\ x\ b$ and $v = a\ y\ b$, where a is the greatest common prefix and b is the greatest common suffix. The diffing algorithm computes (a, x, y, b) from u and v, and it does this quite efficiently using a finite-state machine that operates line by line. If u', the parsed and rendered version of u, is stored, then to compute v' it suffices to render y, which is known. The lists a' and b' are easily computed from u'. If m is the length of the list a, which is known, then a' is the list consisting of the first m elements of u'. If n is the length of b, then b' is the list consisting of the last n elements of u'. All of these operations except for parsing y are quite efficient. But y is usually small in comparison with v, so there is a ordinarily a large speed-up. This speed-up is very much noticeable in practice. The downside is that "differential parse-rendering" can leave parts of the document, e.g. section numbers out-of-sync. To bring them back into sync, one must do a full render.

The diffing strategy just described is far from the theoretical optimum for random changes to a document. If a letter is changed in the first and last paragraphs, or if there are scattered changes throughout the document, then the entire document would be re-parsed and re-rendered. However, a typical human editor tends to make highly localized edits most of the time, so that the diffing strategy is close to optimal in practice. This is especially so since the document is parsed and rendered every 250 milliseconds if there is a change to it. (Documents are saved to the back end every eight seonds.)

8 SCOPE AND SPECIFICATION

The scope of MiniLatex is determined, among other things, by the set of macros and environments which are implemented. At present there are 24 macros that have the same meaning in both LaTeX and MiniLatex; in addition, there are 19 MiniLatex macros that are translated into LaTeX on export, and there are 2 macros which do not have counterparts in LaTeX. Finally, there are 12 environments common to both LaTeX and MiniLatex. These macros and environments are reported in [5]; they constitute part of a draft specification of MiniLatex.

Because of the architecture of the renderer — the use of dictionary-driven accumulators — it is an easy matter to add new macros or new environments: just add a name to a dictionary and define the function that it points to. More sophisticated changes in the current architecture require additions or changes to the record defining the `LatexState` and changes to the dictionary and functions which is references.

A recent addition to the MiniLatex system is a the ability to expand text-mode macros defined in the usual way by `\newcommand`. For example, one could say `\newcommand{\hello}[1]{Hello \strong{#1}!}`, then say `\hello{John}` to produce the rendered text: Hello **John**!. The macro expansion feature is implemented via a function

```
expandMacro : LatexExpr -> LatexExpr -> LatexExpr
```

which manipulates the AST. It weighs in at 63 lines of code andt greatly extends the scope and flexibility of MiniLatex. We plan to investigate the related topic of defining new environments at runtime.

9 DIFFERENCES AND LIMITATIONS

The statement that MiniLatex is a subset of LaTeX needs qualification. Let $\mathcal{L}$ denote LaTeX, and let $\mathcal{M}$ denote MiniLatex. Then the relevant sets on which to comment are (1) $\mathcal{L} \cap \mathcal{M}$, (2) $\mathcal{L} - \mathcal{M}$, and (3) $\mathcal{M} - \mathcal{L}$. The first set is the "compatibility" set and is largely described by the set of macros and environments currently implemented in MiniLatex. The second set represents work to be done, e.g., macros and environments to be implemented. The third set represents features of MiniLatex not present in LaTeX. This a small set, e.g., a three-argument macro for placing images. Most of these differences are resolved when a MiniLatex document is exported. Macros are transformed if need be, and if need be a macro definition is supplied in the document preamble so that it can be rendered using `pdflatex`. There is a much smaller set of specialized macros with no counterpart in LaTeX, e.g.,macros for placing scrollable PDF documents.

9.1 Math-mode macro definitions

MiniLatex supports math-mode macros because MathJax supports them. One can, for example, enclose these definitions in double dollar signs to make them available in a document. In knode.io there are facilities for making this operation more convenient: one may define a "macro document" where macros are defined, then ask the the given document use it when the text is rendered.

9.2 Inputs, style sheets, etc.

Except for composite documents which are made of many subdocuments, there is no notion of a document with input files. There is, however, the notion of a master document for knode.io: basically a list of references to other documents. Master documents are used, for example, in creating sets lectures notes. See knode.io/427 and knode.io/424.

There are no style sheets in MiniLatex, though these may be added to exported MiniLatex documents. That is how this document was created: it was first written on knode.io, then exported and inserted with a few minor changes in the template for Communications of the ACM. There is

also no \begin{document} — \end{document} pair. These are added on export so that the file can be run through pdlatex without further changes. There are no regions, as in

```
{\bf This is bold text}
```

This decision was made in part to keep the parser as small as possible. For the same reason, there is also no use of dashes to produce m- and n-dashes. Use \mdash and \ndash or unicode characters instead. MiniLatex prefers space above and below environments so that the chunking is as fine-grained as possible.

10 TESTING

MiniLatex has a test suite for core components, e.g., the parser, the differ, and the accumulator. The test suite needs to be expanded. Another test runs as follows. There is a render-to-latex function which is used for export. Let f be the function "parse and render to latex." If the system is functioning correctly, then for all source text x, $f(f(x)) = f(x)$ — provided that $x \in \mathcal{M} \cap \mathcal{L}$.

11 FUTURE WORK

As noted in the introduction, MiniLatex is still a research project, albeit one that has been successfully used for writing some quite substantial lecture notes, e.g., knode.io/424. A major goal of the project is to make MiniLatex a tool that its natural communities can rely on to produce web content. For that reason we intend to expand its scope (macros, environments, possibly other features) on the basis of our own perceived needs and also on the basis of feedback from the community. A specification is also being drawn up.

ACKNOWLEDGEMENTS

I would like to express my thanks to the many people on the Elm Slack who have helped me with this project. I am especially grateful to Evan Czapliciki, Luke Westby, and Ilias van Peer for their help and advice. I would also like to thank Davide Cervone for crucial help with MathJax.

REFERENCES

[1] James Carlson. 2018. knode.io (App). https://knode.io
[2] James Carlson. 2018. MiniLatex (Elm package). https://package.elm-lang.org/packages/jxxcarlson/meenylatex/latest/
[3] James Carlson. 2018. MiniLatex Live (App). https://jxxcarlson.github.io/app/miniLatexLive/index.html
[4] James Carlson. 2018. MiniLatex Reader (App). https://knode.io/reader3/
[5] James Carlson. 2018. MiniLatex Technical Report. https://minilatex.gitbook.io/
[6] David Cervone, Volker Sorge, Christian Perfect, and Peter Krautzberger. 2009. MathJax: A Javascript Library for Rendering Mathematics. https://mathjax.org
[7] Evan Czaplicki. 2012. Elm: Concurrent FRP for Functional GUIs. https://www.seas.harvard.edu/sites/default/files/files/archived/Czaplicki.pdf
[8] Evan Czaplicki. 2018. elm-lang.org. https://elm-lang.org/
[9] Evan Czaplicki. 2018. elm/parser. https://package.elm-lang.org/packages/elm/parser/latest/
[10] Donald E. Knuth. 1970. *The TeXbook*. Addison-Wesley, New York, NY.
[11] Leslie Lamport. 1994. *LaTeX: A Document Preparation System*. Addison-Wesley, New York, NY.
[12] Daan Leijen. 2016. Rendering Mathematics for the Web using Madoko. In *Proceedings of the ACM Symposium on Document Engineering*. ACM Press, New York, NY.
[13] John MacFarlane. 2017. Pandoc a universal document converter. https://pandoc.org/

Received November 2018