Featherweight TEX and Parser Correctness

Sebastian Thore Erdweg and Klaus Ostermann

University of Marburg, Germany

Abstract. TEX (and its LATEX incarnation) is a widely used document preparation system for technical and scientific documents. At the same time, TEX is also an unusual programming language with a quite powerful macro system. Despite the wide range of TEX users (especially in the scientific community), and despite a widely perceived considerable level of "pain" in using TEX, there is almost no research on TEX. This paper is an attempt to change that.

To this end, we present Featherweight TEX, a formal model of TEX which we hope can play a similar role for TEX as Featherweight Java did for Java. The main technical problem which we study in terms of Featherweight TEX is the parsing problem. As for other dynamic languages performing syntactic analysis at runtime, the concept of "static" parsing and its correctness is unclear in TEX and shall be clarified in this paper. Moreover, it is the case that parsing TEX is impossible in general, but we present evidence that parsers for practical subsets exists.

We furthermore outline three immediate applications of our formalization of TEX and its parsing: a macro debugger, an analysis that detects syntactic inconsistencies, and a test framework for TEX parsers.

1 Introduction

Almost every user of TEX [7,8] or LATEX [10] is familiar with the technique of binary error search: Since TEX error messages often give no hints about the cause of an error ("File ended while scanning use of ...", "Something's wrong - perhaps a missing ..."), TEX users comment out one half of the code were the cause is suspected and continue by binary search. The situation gets even worse when macros are involved, since all errors in macro definitions – including simple syntactic errors – only show up when the macro is invoked. Even when the error message contains some context information it will be in terms of expanded macros with no obvious relation to the cause of the error in the original document. There is no formal grammar or parser for TEX, and consequently no syntactic analysis which could reveal such errors in a modular way – let alone more sophisticated analyses such as type checkers.

Errors also often arise since it is not clear to the user how the evaluation model of TEX works: When and in which context is a piece of code evaluated? There is no formal specification, but only rather lengthy informal descriptions in TEX books [4,7]. The TEX reference implementation [8] is much too long and complicated to serve as a substitute for a crisp specification.

We believe it is a shame that the programming community designs beautiful programming languages, analyses, and tools in thousands of papers every year, yet the language which is primarily used to produce these documents is neither very well understood, nor amenable to modern analyses and tools.

We have developed a formal model (syntax and operational semantics) of TEX, which we call Featherweight TEX, or FTEX for short. FTEX omits all typesetting related properties of TEX (which are constituted mostly by a set of a few hundred primitive commands) and concentrates on the TEX macro system. We hope that it can have the same fertilizing effect on TEX research that Featherweight Java [6] had for Java.

We use FTEX to study the parsing problem. A parser for TEX would have immediate applications such as "static" syntactic error checking, code highlighting or code folding and many other conveniences of modern programming editors, and it would enable the application of other more sophisticated analyses which typically require an abstract syntax tree as input. It would also open the door to migrating thousands of existing TEX libraries to other text preparation systems or future improved versions of TEX.

To appreciate the parsing problem it is important to understand that in many dynamic languages – general-purpose languages as well as domain-specific languages – parsing and evaluation are deeply intertwined. In this context, it is not clear how a static parser could operate or what a correct parser even is. Due to the many advantages of static syntactic analyses, the parsing problem is not only relevant to TEX but to dynamic language engineering in general.

In particular, programs of dynamic languages do not necessarily have a syntax tree at all. For example, consider the following TEX program:

$$\begin{array}{l} \mathit{\backslash def} \cdot \mathit{\backslash app} \cdot \#1 \cdot \#2 \cdot \{\#1 \cdot \#2\} \\ \mathit{\backslash def} \cdot \mathit{\backslash id} \cdot \#1 \cdot \{\#1\} \\ \mathit{\backslash app} \cdot a \cdot b \\ \mathit{\backslash app} \cdot \mathit{\backslash id} \cdot c \end{array}$$

It defines a macro $\mathit{\backslash app}$ which consumes two arguments, the "identity macro" $\mathit{\backslash id}$, and two applications of $\mathit{\backslash app}$. This program will evaluate to the text $a \cdot b \cdot c$. Now consider the question whether the body of $\mathit{\backslash app}$, $\#1 \cdot \#2$, is a macro application or a text sequence. The example shows that it can be both, depending on the arguments. If the first argument to $\mathit{\backslash app}$ is a macro consuming at least one argument, then it will be a macro application, otherwise a sequence. Since TEX is a Turing-complete language, the property whether a program has a parse tree is even undecidable.

Our work is based on the hypothesis that most TEX documents do have a parse tree – for example, TEX users will typically not define macros where an argument is sometimes a macro, and sometimes a character. Hence our ultimate goal is to solve the parsing problem for a class of documents that is large enough for most practical usages of TEX. To this end, we identify a set of TEX features that are particularly problematic from the perspective of parsing, and present evidence that many TEX documents do not use these features. As additional

evidence we define a parser for a subset of TEX in which the absence of the problematic features is syntactically guaranteed.

We will *not* present a working parser for unrestricted TEX documents, though. A careful design of a parser will require a more in-depth analysis of typical TEX libraries and documents, to come up with reasonable heuristics. But even more importantly, we believe that the first step in developing a syntactic analysis must be to understand exactly what it means for such an analysis to be correct. For TEX, this is not clear at all. Most languages are designed on top of a context-free grammar, hence the question of correctness does not arise, but since the semantics of TEX is defined on a flat, unstructured syntax it is not clear what it means for a parse tree to be correct. We will formulate correctness of a parse tree as as a correct prediction of the application of reduction rules during program evaluation.

The contributions of this work are as follows:

- We present FTEX, the first formal model of TEX. It is, compared to existing descriptions, rather simple and concise. It can help TEX users and researchers to better understand TEX evaluation, and it can be the basis for more research on TEX.
- We describe in detail the problem of parsing TEX and formalize a correctness criterion for syntactic analyses. This correctness criterion can not only be used for formal purposes, but also as a technique to test parser implementations.
- We identify those features of TEX that are particularly worrisome from the perspective of parsing. We have also evaluated how the TEX macro system is used in typical LATEX documents by instrumenting a LATEX compiler to trace macro usages. This analysis shows that worst-case scenarios for parsing (such as dynamically changing the arity of a macro) rarely occur in practice.
- We present a working parser for a subset of TEX and verify its correctness.
- We show how our formalism can be adapted to form tools relevant in practice. Amongst others, we outline how a macro debugger can be constructed using our TEX semantics.
- Since LATEX is just a library on top of TEX, all our results apply to LATEX as well.

We also believe that our formal model and parsing approaches can be applied and adopted to other languages that have similar parsing problems, such as C with the C preprocessor, or Perl. CPP is agnostic to the grammar of C, hence the C grammar cannot be used to parse such files. Obviously, it would be desirable to have a parser that identifies macro calls and their corresponding arguments. In Perl, like in TEX, parsing and evaluation are intertwined. For example, the syntactic structure of a Perl function call `(foo $arg1, $arg2)` may be parsed as either `(foo($arg1), $arg2)` or `foo($arg1, $arg2)`, depending on whether `foo` currently accepts one or two arguments. In this paper, we will concentrate on TEX, though, and leave the application of our techniques to these languages for future work.

With regard to related work, to the best of our knowledge this is the first work investigating the macro system and parsing problem of TEX (or *any* aspect of TEX for that matter). There are some tools that try to parse LATEX code, such as pandoc[1] or syntax highlighters in LATEX IDEs such as TEXnicCenter[2], but these tools only work on a fixed quite small subset of LATEX and cannot deal with user-defined macro definitions appropriately. This is also, to our knowledge, the first work to define a correctness criterion for syntactic analyses of languages that mix parsing and evaluation. The main existing related work is in the domain of parsers for subsets of C with CPP [2,5,11,12,13,14], but none of these works is concerned with formally (or informally) defining correctness.

The rest of this paper is structured as follows. The next section describes why parsing TEX is hard. Section 3 presents our TEX formalization, FTEX. Section 4 defines parser correctness by means of a conformance relation between syntax trees and parsing constraints generated during program evaluation. In Section 5 we explain the difficulties of parsing TEX in terms of our formalization and demonstrate that provably correct parsers exist for reasonably large subsets of TEX. Our empirical study of TEX and LATEX macro usage is presented in Section 6. Section 7 discusses the applicability of our techniques to everyday TEX programming. Section 8 concludes.

2 Problem Statement

TEX has a number of properties that make parsing particularly challenging. First, TEX macros are dynamically scoped: A macro call is resolved to the last macro definition of that name which was encountered, which is not necessarily the one in the lexical scope of the macro call. This means that the target of a macro call cannot be statically determined, which is a problem for parsing since the actual macro definition determines how the call is parsed. For example, whether a is an argument to foo in the macro call $\mathit{foo} \cdot a$ depends on the current definition of foo.

Second, macros in TEX can be passed as arguments to other macros. This induces the same problem as dynamic scoping: Targets of macro calls can in general not be determined statically.

Third, TEX has a lexical [3] macro system. This means that macro bodies or arguments to macros are not necessarily syntactically correct[3] pieces of TEX code. For example, a macro body may expand to an incomplete call of another macro, and the code following the original macro invocation may then complete this macro call. Similarly, macros may expand to new (potentially partial) macro definitions.

Fourth, TEX allows a custom macro call syntax through delimited parameters. Macro invocations are then "pattern-matched" against these delimiters. For example, the TEX program

[1] `http://johnmacfarlane.net/pandoc/`

[2] `http://www.texniccenter.org`

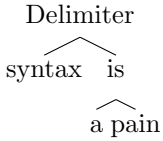
[3] We do not know yet what syntactic correctness for TEX means anyway

$\backslash def \cdot \backslash foo \cdot \#1 \cdot d \cdot \#2 \cdot \{\#1 \cdot x \cdot y \cdot \#2\}$
$\backslash foo \cdot a \cdot b \cdot c \cdot d \cdot e$

will evaluate to $a \cdot b \cdot c \cdot x \cdot y \cdot e$. Such delimiters can be used by TEX libraries to effectively define new domain-specific syntax. For example, using the `qtree`[4] library for drawing trees, the command

```
\Tree [.Delimiter syntax [.is [.a ] pain ] ]
```

results in the following tree:



There are two other parsing-related properties of TEX which, however, we will not be concerned with in this paper, namely category codes and TEX primitives. During the evaluation of a TEX program, a mapping of characters to category codes is maintained. For example, the category code `0` describes escape characters, which is usually only the backslash `\`. In principle, it is possible to change the category code mapping while evaluating a TEX program, but to our knowledge this happens very rarely outside the "bootstrapping" libraries, so we do not expect this to be a big problem in practice. It is in any case a problem that can be dealt with separately from the problem we are dealing with here.

TEX primitives are also relevant for parsing, because they can change the argument evaluation in a way that cannot be expressed using macros. Most notably, the *\expandafter* and *\futurelet* commands affect the evaluation order of programs. The former temporarily skips ahead of the expression following it, while the latter constitutes a lookahead mechanism. Furthermore, with TEX's various kinds of variables come a multitude of special-syntax assignment primitives, and alignments need their own special treatment anyway [7,4]. Still, there seems to be no conceptual hurdle for our main goal, that is the development of a correct syntactic analysis for a feature-rich subset of TEX, and we believe that the formal model presented next captures the most interesting and challenging aspects of TEX for parsing.

3 Featherweight TEX

We have formalized the core of the TEX macro system – in particular the aspects described in the previous section – in the form of a small-step operational semantics. We call this language FTEX and present its syntax in Fig. 1.

In FTEX, a program s consists of five primitive forms, namely characters c, macro identifiers m, macro parameters x, groups $\{s\}$, and the macro definition command $\backslash def$. We call these forms *expressions*. In addition, let $\diamondsuit$ represent the empty expression, which is not allowed to occur in user programs. Expressions are composed by sequentialization only, i.e. there is no syntactic distinction between

[4] `http://www.ling.upenn.edu/advice/latex/qtree/`

$$
\begin{array}{rcll}
c & \in & \mathit{character} & \text{characters} \\
m & \in & \mathcal{M} & \text{macro variables} \\
x & ::= & \#1 \mid \ldots \mid \#9 \mid \#x & \text{macro parameters} \\
e & ::= & c \mid m \mid x \mid \{s\} \mid \backslash \mathit{def} \mid \diamondsuit & \text{expressions} \\
s & ::= & \overline{e} & \text{FTEX programs}
\end{array}
$$

$$
\begin{array}{rcll}
\sigma & ::= & \{\overline{x \mapsto s}\} & \text{substitutions} \\
r & ::= & c \mid x \mid m \mid \diamondsuit & \text{parameter tokens} \\
d_M & ::= & \backslash \mathit{def} \cdot M \cdot \overline{r} \cdot \{s\} & \text{macro values} \\
v & ::= & \varepsilon \mid c \cdot v \mid d_{\mathcal{M}} \cdot v \mid \diamondsuit \cdot v & \text{values} \\
R_M & ::= & [\,] \mid c \cdot R_M \mid d_M \cdot R_M & \text{reduction contexts} \\
 & & \mid \diamondsuit \cdot R_M \mid \{R_M\} \mid R_M \cdot e &
\end{array}
$$

Fig. 1. FTEX syntax

macro calls and the concatenation of text, see definition of s. Since this syntax does not identify the structure of macro applications, we call s the *flat syntax*. We do not restrict the use of the macro definition command to syntactically valid macro definitions (such as $md ::= \backslash \mathit{def} \cdot m \cdot \overline{x} \cdot \{s\}$) since in TEX macro definitions may be computed dynamically by the expansion of other macros. For example, one could define a macro $\backslash \mathit{def}'$ which behaves exactly like $\backslash \mathit{def}$ but, say, adds an additional dummy argument to the macro.

The operational semantics of FTEX, Fig. 2 is defined in a variant of the evaluation context style from Wright and Felleisen [16]. This means that, instead of introducing environments and similar entities, every relevant runtime entity is encoded within the language's syntax. The necessary additional (runtime) syntax and evaluation contexts are defined below the FTEX syntax in Fig. 1. It is important to note that the forms d and R are parametrized over the set of macro variables $M \subseteq \mathcal{M}$ that may occur in definitions.

$$
\begin{array}{l}
\mathit{dropDefs} : v \to v \\
\mathit{dropDefs}(\varepsilon) \quad = \varepsilon \\
\mathit{dropDefs}(c \cdot v) \;= c \cdot \mathit{dropDefs}(v) \\
\mathit{dropDefs}(d \cdot v) \;= \diamondsuit \cdot \mathit{dropDefs}(v) \\
\mathit{dropDefs}(\diamondsuit \cdot v) = \diamondsuit \cdot \mathit{dropDefs}(v)
\end{array}
\qquad
\begin{array}{l}
\hat{\sigma} : s \to s \\
\hat{\sigma}(x) \quad = \sigma(x) \\
\hat{\sigma}(\{s\}) \;= \{\hat{\sigma}(s)\} \\
\hat{\sigma}(e) \quad = e \\
\hat{\sigma}(\varepsilon) \quad = \varepsilon \\
\hat{\sigma}(e \cdot \overline{e}) = \hat{\sigma}(e) \cdot \hat{\sigma}(\overline{e})
\end{array}
$$

$$
\frac{s \to s'}{R_{\mathcal{M}}[s] \to R_{\mathcal{M}}[s']}\ \text{(R-RStep)}
\qquad
\frac{}{\{v\} \to \mathit{dropDefs}(v)}\ \text{(R-GVal)}
$$

$$
\frac{\mathit{match}(\overline{r},\, s') = \sigma}{\begin{array}{r}\backslash \mathit{def} \cdot m \cdot \overline{r} \cdot \{s\} \cdot R_{\mathcal{M}\setminus\{m\}}[m \cdot s'] \\ \to \backslash \mathit{def} \cdot m \cdot \overline{r} \cdot \{s\} \cdot R_{\mathcal{M}\setminus\{m\}}[\hat{\sigma}(s)]\end{array}}\ \text{(R-Macro)}
$$

Fig. 2. FTEX reduction semantics

The reduction system has only three rules: A congruence rule (R-RSTEP) which allows the reduction inside a context, a reduction rule (R-GVAL) to eliminate groups that are already fully evaluated, and a rule (R-MACRO) for evaluating macro applications.

(R-RSTEP) is standard for every evaluation-context based semantics. For (R-GVAL), it is important to understand that macro definitions inside a group are not visible outside the group, hence there is actually a mix of static and dynamic scoping. Within a fixed nesting level, scoping is dynamic, i.e., the rightmost definition of the macro wins, as long as it is on the same or a lower nesting level, but definitions in deeper nesting levels are ignored. For this reason it is safe to discard all macro definitions in fully evaluated groups and just retain the text contained in it - which is exactly what *dropDefs* does.

Not surprisingly, the (R-MACRO) rule for evaluating macro calls is the most sophisticated one. The evaluation context $R_{\mathcal{M}\setminus\{m\}}$ is used to make sure that the macro definition of m is indeed the right-most one on the same or lower nesting level by prohibiting further definitions of m. (R-MACRO) uses the parameter text $\overline{r}$ and the macro arguments s' to compute a substitution σ, which is then applied to the macro body s.[5] Substitution application $\hat{\sigma}$ is not hygienic [9], capture-avoiding or the like; rather, it just replaces every occurrence of a variable by its substitute.

The *match* function is used for matching actual macro arguments with the parameter text, i.e., the part between the macro name and the macro body, which may potentially contain delimiters of the form c or m in addition to the macro parameters; see the syntax of r. *match* expects the parameter text as its first argument and the argument text as its second one. It then generates a substitution σ, mapping macro parameters x to TEX terms s.

Due to space limitations we cannot display the full definition of *match* here, but instead present a few examples in Fig. 3. The first two calls of *match* accord to usual parameter instantiation, where in the first one the argument's group is unpacked. Delimiter matching is illustrated by the subsequent three calls. In particular, the fifth example shows that delimited variables may instantiate to the empty expression. The last example fails because only delimited parameters can consume sequences of arguments; the second character is not matched by the parameter.

$$
\begin{array}{lll}
\mathit{match}(x, & \{c_1 \cdot c_2\} &) = \{x \mapsto c_1 \cdot c_2\} \\
\mathit{match}(x_1 \cdot x_2, & c_1 \cdot c_2 &) = \{x_1 \mapsto c_1, x_2 \mapsto c_2\} \\
\mathit{match}(c_1 \cdot c_2, & c_1 \cdot c_2 &) = \{\} \\
\mathit{match}(x \cdot c_4, & c_1 \cdot c_2 \cdot c_3 \cdot c_4 &) = \{x \mapsto c_1 \cdot c_2 \cdot c_3\} \\
\mathit{match}(x \cdot c, & c &) = \{x \mapsto \diamondsuit\} \\
\mathit{match}(x, & c_1 \cdot c_2 &) = \mathit{undefined}
\end{array}
$$

Fig. 3. Examples of Matching

[5] Here and in the remainder of this paper, we use $\overline{a} = a_1 \cdots a_n$ to denote a sequence of a's and ε to denote the empty sequence.

In summary, macros are expanded by first determining their active definition using parametrized reduction contexts, then matching the parameter text against the supplied argument text, and finally applying the resulting substitution to the macro body, which replaces the macro call.

4 Correctness of Syntactic Analyses

Given an unstructured representation of a program, syntactic analyses try to infer its structured representation, that is a syntax tree. A syntax tree is a program representation in which program fragments are composed according to the syntactic forms they inhabit. In FTEX, there are three compositional syntactic forms, namely macro application, macro definition and sequentialization, see Fig. 4. Macro applications $t \mathbin{@} \bar{t}$ consist of a macro representation and the possibly empty list of arguments. The definition of a macro is represented by the sequence of program fragments forming the definition $\langle \bar{t} \rangle$. The sequentialization of two trees $t\,;t$ is used to denote sequential execution and result concatenation.

$$
\begin{array}{lll}
f & ::= c \mid m \mid x \mid \{t\} \mid \backslash def & \text{tree expressions} \\
t & ::= \emptyset \mid f \mid t\,;t \mid t \mathbin{@} \bar{t} \mid \langle\, \bar{t}\, \rangle & \text{syntax trees} \\
t_\perp & ::= \perp \mid t & \text{parse result}
\end{array}
$$

Fig. 4. Tree syntax

An FTEX parser thus is a total function $p : s \to t_\perp$ assigning either a syntax tree or $\perp$ to each program in s (see Fig. 1 for the definition of s). However, not each such function is a valid parser; the resulting syntax trees must represent the original code and its structure correctly. But what characterizes a correct structural representation? Looking at syntax trees from another angle helps answer this question.

Syntax trees can also be understood to predict a program's run in that the evaluation needs to follow the structure specified by the tree. More precisely, if we assume the language's semantics to be syntax-directed, the syntax tree's inner nodes restrict the set of applicable rules to those matching the respective syntactic form. By inversion, the reduction rules used to evaluate a program constrain the set of valid syntax trees. For instance, in a program which is reduced by macro expansion, the macro and its arguments must be related by a macro application node in the syntax tree. A value, on the other hand, must correspond to a sequence of macro definitions and characters.

Accordingly, a correct FTEX parser is a total function $p : s \to t_\perp$ such that (i) each syntax tree represents the original code and (ii) each tree is compatible with the reduction rules chosen in the program's run.

In the present section, we formalize these requirements, i.e. we give a formal specification of correct syntax trees and parsers. Often, however, one wants a syntax tree to be compatible not only with the specific program run generated

by evaluating the represented program; rather, syntax trees should be modular, i.e. reusable in all contexts the represented programs may be used in. We refer the specification of modular syntax trees and analyses to future work, though.

4.1 Structure Constraints

Essentially, FTEX programs are syntactically underspecified. The operator $\cdot$ is used to compose expressions, but its syntactic meaning is ambiguous; it may correspond to any of the following structural forms of the stronger tree syntax:

- Expression sequences: for example, $c_1 \cdot c_1$ corresponds to the character sequence $c_1\,;c_2$.
- Macro application: for example, $\mathit{\backslash foo}\cdot c$ corresponds to the application $\mathit{\backslash foo}\mathbin{@}c$ if $\mathit{\backslash foo}$ represents a unary macro.
- Macro argument sequence: for example, in $\mathit{\backslash bar}\cdot c_1\cdot c_2$ the second use of $\cdot$ forms a sequence of arguments to $\mathit{\backslash bar}$ if the macro takes two arguments. The call then corresponds to the syntax tree $\mathit{\backslash bar}\mathbin{@}c_1\cdot c_2$.
- Macro definition constituent sequence: for example, all occurrences of $\cdot$ in $\mathit{\backslash def}\cdot\mathit{\backslash id}\cdot\#1\cdot\{\#1\}$ contribute to composing the constituents of the macro definition $\langle\mathit{\backslash def}\cdot\mathit{\backslash id}\cdot\#1\cdot\{\#1\}\rangle$.

While evaluating an FTEX program, the syntactic meaning of uses of $\cdot$ become apparent successively, as illustrated in the above examples. In order to reason about a program's syntactic runtime behavior, we introduce a constraint system which relates the syntactic meaning of occurrences of $\cdot$ to their use during reduction. A syntax tree then has to satisfy all generated constraints, i.e. it has to predict the syntactic meaning of each occurrence of $\cdot$ correctly.

In order to distinguish different uses of $\cdot$ and relate them to their incarnations in syntax trees, we introduce labels $\ell \in \mathcal{L}$ for expression concatenation and tree composition as shown in Fig. 5.[6] Similar to sequences $\overline{a}$, we write $\widetilde{a} = a_1 \mathbin{.^{\ell_1}} \ldots \mathbin{.^{\ell_{n-1}}} a_n$ to denote sequences of a's where the composing operator $\cdot$ is labeled by labels $\ell_i \in \mathcal{L}$. All labels in non-reduced FTEX programs and syntax trees are required to be unique. The reduction semantics from the previous section is refined such that labels are preserved through reduction. This, however, violates label-uniqueness as the following example shows.

$$
\begin{array}{l}
\ell \;\in\; \mathcal{L} \\
s \;::=\; \widetilde{e} \\
t \;::=\; \emptyset \mid f \mid t \mathbin{;^{\ell}} t \mid t \mathbin{@^{\ell}} \widetilde{t} \mid \langle \widetilde{t} \rangle \\
\\
k \;::=\; \mathit{SEQ}(\ell) \mid \mathit{APP}(\overline{\ell}) \mid \mathit{DEF}(\overline{\ell})
\end{array}
$$

Fig. 5. Label-extended Syntax and Structure Constraints

[6] Though the set of labels $\mathcal{L}$ is left abstract, we will use natural numbers as labels in examples.

Example 1.

$$\begin{aligned}&\backslash def\ {\cdot^1}\ \backslash seq\ {\cdot^2}\ \#1\ {\cdot^3}\ \#2\ {\cdot^4}\ \{\#1\ {\cdot^5}\ \#2\}\ {\cdot^6}\ \backslash seq\ {\cdot^7}\ c_1\ {\cdot^8}\ c_2\\ \rightarrow\ &\backslash def\ {\cdot^1}\ \backslash seq\ {\cdot^2}\ \#1\ {\cdot^3}\ \#2\ {\cdot^4}\ \{\#1\ {\cdot^5}\ \#2\}\ {\cdot^6}\ c_1\ {\cdot^5}\ c_2\end{aligned}$$

In this example, a macro $\backslash seq$ taking two arguments is defined. In the macro body the two arguments are composed by the operator labeled 5. Since labels are preserved through reduction, label 5 occurs outside the macro body after expanding $\backslash seq$, namely in the body's instantiation $c_1 \cdot^5 c_2$.

When only regarding the uninstantiated macro body of $\backslash seq$ there is no way of telling how the macro arguments are combined, i.e. which syntactic meaning the operator labeled 5 inhabits. Depending on the actual first argument the operator could, for instance, denote a macro application or the construction of a macro definition. In the above example, however, the macro expands into a composition of characters, hence the operator has to represent the sequentialization of expressions.

More generally, labels fulfill two purposes. First, they enable the identification of conflicts: if, for instance, a macro is called several times and expands into conflicting syntactical forms, such as sequentialization and application, there will be conflicting requirements on the involved labels. The respective operators thus have to have several syntactic meanings during runtime, which rules out a static syntactic model. Second, by using the same labels in FTEX programs and syntax trees, the structural prediction made by a parser can be easily related to the program's structure during runtime.

Our formulation of syntax tree correctness relies on the notion of structure constraints k, Fig. 5, which restrict the syntactic meaning of labeled composition operators. $SEQ(\ell)$ requires that $\cdot^\ell$ represents an expression sequentialization, $APP(\ell_1 \cdots \ell_n)$ denotes that $\cdot^{\ell_1}$ is a macro application composition and $\cdot^{\ell_2} \ldots \cdot^{\ell_n}$ forms a sequence of macro argument. Since, macro argument sequentialization always is accompanied by a macro application, only one form of constraint is needed. Lastly, constraints of the form $DEF(\ell_1 \cdots \ell_n)$ demand that the labeled operators $\cdot^{\ell_1} \ldots \cdot^{\ell_n}$ represent the composition of macro definition constituents.

To see these constraints in action, reconsider Ex. 1. As we will show in the subsequent subsection, all of the following constraints can be derived by evaluation:

$$DEF(1 \cdot 2 \cdot 3 \cdot 4),\ SEQ(5),\ SEQ(6)\ APP(7 \cdot 8)$$

4.2 Constraint Generation

The syntactic relations between different parts of a program emerge during evaluation. To this end, we instrument the previously presented reduction semantics to generate structure constraints as the syntactic meanings of compositions $\cdot$ become apparent.

The adapted reduction semantics is shown in Fig. 6. For a reduction step $s \xrightarrow{K} s'$, K denotes the set of generated constraints. Accordingly, the reduction rule (R-RSTEP) simply forwards the constraints generated for the reduction of the

subexpression s. When applying (R-GVal), it is exploited that the group contains a value only, i.e. that it contains a sequence of macro definitions and characters. The corresponding structure constraints are generated by applying *valcons* to that value.

$$
\begin{array}{lll}
\multicolumn{3}{l}{valcons : v \to \{k\}} \\
valcons(\varepsilon) & = \emptyset & \\
valcons(c) & = \emptyset & \\
valcons(d) & = \{DEF(\ell_1 \cdots \ell_n)\}, & \\
 & \quad \text{where } d = e_1 \cdot^{\ell_1} \ldots \cdot^{\ell_n} e_{n+1} & \\
valcons(c \cdot^{\ell} v) & = \{SEQ(\ell)\} \cup valcons(v) & \\
valcons(d \cdot^{\ell} v) & = valcons(d) \cup \{SEQ(\ell)\} \cup valcons(v) & \\
valcons(\diamondsuit \cdot^{\ell} v) & = \{SEQ(\ell)\} \cup valcons(v) &
\end{array}
$$

$$
\frac{s \xrightarrow{K} s'}{R_{\mathcal{M}}[s] \xrightarrow{K} R_{\mathcal{M}}[s']}\ \text{(R-RStep)} \qquad \frac{}{\{v\} \xrightarrow{valcons(v)} dropDefs(v)}\ \text{(R-GVal)}
$$

$$
\frac{}{\begin{array}{c} \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \{s\} \cdot^{\ell_3} R_{\mathcal{M}\setminus\{m\}}[m] \\ \xrightarrow{\emptyset} \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \{s\} \cdot^{\ell_3} R_{\mathcal{M}\setminus\{m\}}[\hat{\sigma}(s)] \end{array}}\ \text{(R-MacroEps)}
$$

$$
\frac{\begin{array}{c} match(\widetilde{r}, \widetilde{e}) = \sigma \\ \widetilde{e} = e_1 \cdot^{\ell'_1} \cdots \cdot^{\ell'_n} e_{n+1} \\ k = APP(\ell' \cdot \ell'_1 \cdots \ell'_n) \end{array}}{\begin{array}{c} \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \widetilde{r} \cdot^{\ell_3} \{s\} \cdot^{\ell_4} R_{\mathcal{M}\setminus\{m\}}[m \cdot^{\ell'} \widetilde{e}] \\ \xrightarrow{\{k\}} \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \widetilde{r} \cdot^{\ell_3} \{s\} \cdot^{\ell_4} R_{\mathcal{M}\setminus\{m\}}[\hat{\sigma}(s)] \end{array}}\ \text{(R-Macro)}
$$

Fig. 6. Constraint Generation

Macro expansion is split into two rules. The first one, (R-MacroEps), covers the expansion of macros that have no parameters (note the lack of $\widetilde{r}$ in the macro definition). In this case no syntactical structure is exploited and thus no constraints are generated. The second macro expansion rule, (R-Macro), generates a single constraint which denotes the applicative structure of the expanded macro call.

Constraint generation now can be summarized as follows.

Definition 1. *The set of constraints $\Omega(s \to^* s')$ associated to the reduction sequence $s \to^* s'$ of the program s is defined by:*

$$
\begin{array}{ll}
\Omega(v) & = valcons(v) \\
\Omega(s) & = \emptyset, \ \textit{if } s \neq v \textit{ for all } v \\
\Omega(s \xrightarrow{K} s' \to^* s'') & = K \cup \Omega(s' \to^* s'')
\end{array}
$$

In the definition of Ω, the first two cases cover reduction sequences of length zero, i.e. the constraints for the final state of the particular run of the program is defined. The third case collects all constraint sets generated during reduction and is recursively defined on the given reduction sequence.

Let us once again consider Ex. 1, where s is the initial program and s' its reduct. Then $\Omega(s \to^* s') = \{DEF(1 \cdot 2 \cdot 3 \cdot 4), SEQ(5), SEQ(6)APP(7 \cdot 8)\}$. The constraint $APP(7, 8)$ is generated because of the expansion of the macro *\seq* with arguments c_1 and c_2. The remaining constraints follow from applying *valcons* to s', which is a value and therefore is handled by the first case of Ω.

4.3 Parser Correctness

We are finally back to formulating a correctness criterion for FTEX parsers. First of all, a correct parser needs to produce a syntax tree which represents the analyzed program. In order to be able to relate syntax trees to flat program representations and structure constraints, we introduced labeled trees in Fig. 5. Labeled trees can be easily flattened and compared to weakly structured programs since composition operators are uniquely identified by their label. Accordingly, dropping all of a syntax tree's structural information while retaining the labels reveals the underlying FTEX program:

$$
\begin{array}{ll}
\pi : t \to s & \\
\pi(\emptyset) & = \varepsilon \\
\pi(t_1 \,;^{\ell}\, t_2) & = \pi(t_1) \cdot^{\ell} \pi(t_2) \\
\pi(t \,@^{\ell}\, t_1 \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} t_n) & = \pi(t) \cdot^{\ell} \pi(t_1) \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} \pi(t_n) \\
\pi(\langle t_1 \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} t_n \rangle) & = \pi(t_1) \cdot^{\ell_1} \ldots \cdot^{\ell_{n-1}} \pi(t_n) \\
\pi(\{t\}) & = \{\pi(t)\} \\
\pi(f) & = f, \qquad \text{if } f \neq \{t\} \text{ for all } t
\end{array}
$$

Nevertheless, not only need syntax trees to represent the code correctly, but also its structure. In the previous subsection, we were able to derive the set of constraints representing all syntactic structure a program exhibits during and after evaluation. According to our viewpoint of syntax trees as structural predictions, correct parsers must foresee a program's dynamic structures, i.e. syntax trees have to satisfy all constraints associated with the evaluation of the program.

A constraint is satisfied by a syntax tree if the constrained composition operators are represented within the tree as required. To this end, we first define what it means for a syntax tree to match a constraint, in symbols $\vdash t : k$, Fig. 7. Note that we require the labels in the constraints to equal the ones in the syntax tree, thus assuring that appropriate composition operators are matched only. Constraint satisfaction then is defined as follows.

Definition 2. *A syntax tree t satisfies a constraint k, in symbols $t \models k$, if t contains a subtree t' such that $\vdash t' : k$. A parse tree t satisfies a set of constraints K, in symbols $t \models K$, iff t satisfies all constraints $k \in K$.*

We are now able to specify correctness of FTEX parsers.

$$\frac{}{\vdash t_1 \;;^{\ell}\; t_2 \;:\; SEQ(\ell)}\ \text{(K-Seq)} \qquad \frac{}{\vdash t \,@^{\ell}\, t_1 \,.^{\ell_1} \cdots .^{\ell_n}\, t_{n+1} \;:\; APP(\ell \cdot \ell_1 \cdots \ell_n)}\ \text{(K-App)}$$

$$\frac{}{\vdash \langle t_1 \,.^{\ell_1} \cdots .^{\ell_n}\, t_{n+1} \rangle \;:\; DEF(\ell_1 \cdots \ell_n)}\ \text{(K-Def)}$$

Fig. 7. Constraint matching

Definition 3. *A total function $p : s \to t_\perp$ is a correct FTEX parser if and only if for all FTEX programs s with $p(s) \neq \perp$*

1. *$\pi(p(s)) = s$, and*
2. *for all programs s' with $s \to^* s'$, $p(s) \models \Omega(s \to^* s')$.*

A syntactic analysis for FTEX thus is correct if the resulting syntax trees are proper representations and structural predictions of original programs.

5 Towards Parsing TEX

In the previous section, we presented a formal specification of correct FTEX parsers. Accordingly, syntax trees computed by correct parsers must satisfy all structure constraints emerging during the evaluation of the analyzed program. For some programs, however, the set of generated structure constraints is inherently unsatisfiable, that is, inconsistent: the constraints place contradicting requirements on syntax trees.

In the present section we show that these inconsistencies arise from the problematic language features of TEX we discussed in Section 2. We furthermore demonstrate that by restricting the language such that the problematic features are excluded, we are able to define a provably correct parser for that subset.

5.1 Parsing-contrary Language Features

When language features allow syntactic ambiguities they actually hinder syntactic analyses. For TEX, these ambiguities translate into the generation of inconsistent structure constraints, because ambiguous expression can be used inconsistently. Here we present an example for each such language feature and show that it entails inconsistent constraints. For brevity, we only denote those labels in the examples that are used for discussion.

Dynamic scoping. In TEX, parsing a macro application depends on the applied macro's actual definition. It matters whether the macro expects one or two arguments, for example, because then either the following one or two characters, say, are matched. With dynamic scoping the meaning of a macro variable depends

on the dynamic scope it is expanded in. Therefore, one and the same macro variable can exhibit different syntactic properties:

$\backslash def \cdot \backslash foo \cdot \{c\}$
$\backslash def \cdot \backslash bar \cdot \{\backslash foo \cdot^1 c\}$
$\backslash bar$
$\backslash def \cdot \backslash foo \cdot \#1 \cdot \{c\}$
$\backslash bar$

Here, each of the calls to $\backslash bar$ reduces to $\backslash foo \cdot^1 c$. In the former call, however, $\backslash foo$ is defined as a constant, i.e. it does not expect any argument. Therefore, the former expansion of $\backslash bar$ implies the structure constraint $SEQ(1)$. Contrarily, when expanding the second call to $\backslash bar$, $\backslash foo$ is bound to expect one argument. Therefore, the expansion of $\backslash bar$ corresponds to a macro call of $\backslash foo$ with argument c, and hence the constraint $APP(1)$ is generated.

In a lexically scoped language, the expansion of $\backslash bar$ would consist of a closure that binds all free variables in the macro body. The macro identifier $\backslash foo$ thus would refer to the definition $\backslash def \cdot \backslash foo \cdot \{c\}$ in both expansions of $\backslash bar$.

Higher-order arguments. Similarly to macro identifiers in dynamic scoping, higher-order arguments lead to syntactically ambiguous interpretations of macro parameters. Depending on the actual argument, a parameter may represent a constant expression or in turn a macro.

In the following we define two macros, both of which take two arguments. The former one builds the sequence of its parameters while the second applies the first parameter to the second parameter.

$\backslash def \cdot \backslash seq \cdot \#1 \cdot \#2 \cdot \{\#1 \cdot^1 \#2\}$
$\backslash def \cdot \backslash app \cdot \#1 \cdot \#2 \cdot \{\#1 \cdot \#2\}$

Evidently, the only difference between the definitions of $\backslash seq$ and $\backslash app$ are their names. The macro body's syntactic structure thus only depends on whether or not the first argument is a constant or expects further input. When calling $\backslash seq$ with two characters, say, the call expands to a sequence of these characters and the constraint $SEQ(1)$ is generated. In contrast, when calling $\backslash seq$ with a unary macro and a character, it will expand into yet another macro call. In this case the conflicting constraint $APP(1)$ is generated.

Lexical macro system. In contrast to syntactical macro systems [15], macros in TEX are lexical, that is, macro arguments and bodies do not necessarily correspond to complete syntax trees.

$\backslash def \cdot \backslash foo \cdot \{\backslash def \cdot^1 \backslash baz\}$
$\backslash def \cdot \backslash bar \cdot \#1 \cdot \{\backslash foo \cdot^2 \#1 \cdot^3 \{c\}\}$

In this example neither dynamic scoping nor higher-order macros is relevant. Still, the structure of $\backslash bar$'s body is ambiguous and depends on the argument $\#1$. The call $\backslash bar \cdot c'$, for example, expands to $\backslash def \cdot^1 \backslash baz \cdot^2 c' \cdot^3 \{c\}$ in two steps. Correspondingly, the structure constraint $DEF(1, 2, 3)$ is generated. On the other hand, if $\backslash bar$ is called as in $\backslash bar \cdot \{\{c'\}\}$, it expands to $\backslash def \cdot^1 \backslash baz \cdot^2 \{c'\} \cdot^3 \{c\}$, thus the constraints $DEF(1, 2)$ and $SEQ(3)$ are generated.

Furthermore, the body of $\backslash foo$ cannot be correctly represented by any syntax tree, because it does not inhibit a valid syntactical form. This is the intrinsic difficulty in performing syntax analyses on top of a lexical transformation system such as TEX.

Custom macro call syntax. TEX users are allowed to define their own macro call syntax as desired. For instance, the following macro needs to be called with parentheses.

$\backslash def \cdot \backslash foo \cdot (\cdot \#1 \cdot) \cdot \{c\}$

This, however, easily leads to ambiguities when the call syntax depends on macro parameters, i.e. when a macro argument is matched against the call pattern.

$\backslash def \cdot \backslash bar \cdot \#1 \cdot \{\backslash foo \cdot^1 (\cdot^2 \#1 \cdot^3)\}$
$\backslash bar \cdot c$
$\backslash bar \cdot)$

Here, the call to $\backslash foo$ in the body of $\backslash bar$ depends on the macro parameter $\#1$. While the first call to $\backslash bar$ entails the constraint $APP(1, 2, 3)$ as expected, the second call expands to $\backslash foo \cdot^1 (\cdot^2) \cdot^3)$. Consequently, the constraints $APP(1 \cdot 2)$ and $SEQ(3)$ are generated, and establish an inconsistency with the first expansion of $\backslash bar$.

5.2 Parsing FTEX Correctly

We just identified some sources of syntactic ambiguities in FTEX, and can now focus on finding an actually parsable subset of the language. To this end, we present a non-trivial FTEX parser and prove it correct with respect to Def. 3.

First, let us fix the set of programs our parser p will be able to parse, i.e. for which p results in an actual syntax tree. These programs are subject to the following restrictions:

1. All macro definitions are complete, unary, top-level and prohibit custom call syntax, that is, they strictly follow the syntactic description $\backslash def \cdot^\ell m \cdot^\ell \#1 \cdot^\ell \{s\}$ and occur non-nestedly.
2. All uses of macro variables (except in macro definitions) are directly followed by a grouped expression, as in $m \cdot^\ell \{s\}$. Since all macros are unary, the grouped expression will correspond to the macro's argument which thus can be statically identified.
3. All macros are first-order, i.e. their argument is not a macro itself. To this end, we require all occurrences of macro parameters in macro bodies to be wrapped in groups. A higher-order argument would thus always be captured by a group, as in $\{\backslash foo\}$. This would lead to a runtime error since the content of a group must normalize to a value.

The complete parser p_i is defined in Fig. 8. p_0 accounts for the top-level programs which may contain definitions. Nested program fragments are parsed by p_1 where definitions are prohibited but wrapped macro parameters are allowed.

$$
\begin{array}{ll}
p_i(\varepsilon) & = \emptyset \\
p_i(c \cdot^{\ell} s) & = c \,;^{\ell}\, p_i(s) \\
p_i(m \cdot^{\ell_1} \{s\} \cdot^{\ell_2} s') & = (m \,@^{\ell_1} \{p_1(s)\}) \,;^{\ell_2}\, p_i(s') \\
p_i(\{s\} \cdot^{\ell} s') & = \{p_1(s)\} \,;^{\ell}\, p_i(s') \\
p_i(\diamondsuit \cdot^{\ell} s) & = \emptyset \,;^{\ell}\, p_i(s) \\
p_0(\backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \#1 \cdot^{\ell_3} \{s\} \cdot^{\ell_4} s') & = \langle \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} x \cdot^{\ell_3} \{p_1(s)\} \rangle \,;^{\ell_4}\, p_0(s') \\
p_1(\{\#1\} \cdot^{\ell} s) & = \{x\} \,;^{\ell}\, p_1(s) \\
p_i(s) & = \bot
\end{array}
$$

Fig. 8. A provably correct FTEX parser

The definition of p_i is to be read such that in each case the parser also returns $\bot$ if any nested call returns $\bot$.

In order to verify the correctness of p_0 with respect to Def. 3, we need to show that the resulting syntax trees represent the input programs and satisfy all structure constraints associated to runs of the programs.

Theorem 1. *For all programs s with $p_i(s) \neq \bot$, $\pi(p_i(s)) = s$ for $i \in \{0, 1\}$.*

Lemma 1. *For all values v, $p_i(v) \models \mathit{valcons}(v)$ for $i \in \{0, 1\}$.*

Lemma 2. *For $i \in \{0, 1\}$ and all programs s and s' with $s \xrightarrow{K} s'$ and $p_i(s) \neq \bot$*

(i) $p_i(s) \models K$,
(ii) $p_i(s') \neq \bot$, *and*
(iii) $\{k \mid p_i(s) \models k\} \supseteq \{k \mid p_i(s') \models k\}$.

Proof. By case distinction on the applied reduction rule.

(i) By Lem. 1 and inversion on p_0 and p_1.
(ii) For (R-Macro) this holds since macro bodies are p_1-parsable and p_1-parsability is closed under substitution. The other cases are simple.
(iii) For (R-Macro) let $R = \backslash def \cdot^{\ell_1} m \cdot^{\ell_2} \#1 \cdot^{\ell_3} \{s_b\} \cdot^{\ell_4} R_{\mathcal{M} \setminus \{m\}}[]$ with $R[m \cdot^{\ell'} \{s_a\}] = s$, $R[\sigma(s_b)] = s'$ and $\sigma = \{\#1 \mapsto s_a\}$. Then $\{k \mid p(R[m \cdot^{\ell'} \{s_a\}]) \models k\} \supseteq \{k \mid p(R[\,]) \models k \ \vee \ p(s_a) \models k\} \supseteq \{k \mid p(R[\sigma(s_b)]) \models k\}$. The other cases are simple.

Theorem 2. *For $i \in \{0, 1\}$ and all FTEX programs s and s' with $s \to^* s'$, if $p_i(s) \neq \bot$ then $p_i(s) \models \Omega(s \to^* s')$.*

Proof. By induction on the reduction sequence $s \to^* s'$. For $s = v$ by Lem. 1. For s in normal form and $s \neq v$, $\Omega(s) = \emptyset$. For $s \xrightarrow{K} s' \to^* s''$ by Lem. 2 and the induction hypothesis.

Corollary 1. *p_0 is a correct FTEX parser.*

6 Macro Usage in TEX and LATEX

In the previous section, we demonstrated how our formal specification of FTEX parsers can be instantiated to give a provably correct parser. The subset of FTEX the parser supports was designed to avoid all of the pitfalls described in Section 5.1. Parsers which support larger subsets of FTEX do exist, though, and in the present section we set out to justify their relevance in practice.

Identifying a sublanguage of TEX which is broad enough to include a wide range of programs used in practice and, at the same time, is precise enough to include a considerably large parsable subset, is a task involving a detailed study of existing TEX and LATEX libraries as well as conflict-resolving heuristics, which exclude some programs in favor of others. Nonetheless, we believe that significantly large portions of the existing TEX and LATEX libraries in fact are parsable, i.e. not exposed to the issues associated with dynamic scoping, delimiter syntax and macro arguments consuming further input. This claim is supported by a study of macro usage in existing TEX and LATEX libraries, Fig. 9, which we have been carrying out.

Macro definitions		
total	345823	∼ 100.0%
constant	320357	∼ 92.6%
recursive	56553	∼ 16.4%
delimiter syntax	12427	∼ 3.6%
redefinitions	160697	∼ 46.5%
redef. arity changes	16289	∼ 4.7%
– ignoring macros "*temp*"	7175	∼ 2.1%
redef. delimiter syntax changes	16927	∼ 4.9%
– ignoring macros "*temp*"	5827	∼ 1.7%

Macro expansions		
total	2649553	∼ 100.0%
constant	746889	∼ 28.2%
recursive	524320	∼ 19.8%
delimiter syntax	95512	∼ 3.6%
higher-order arguments	70579	∼ 2.7%

Fig. 9. Macro usage in TEX and LATEX libraries

To collect various statistics about macro definitions and applications, we adapted a Java implementation of TEX, called the New Typesetting System [1].[7] Our adaption is available at `http://www.informatik.uni-marburg.de/~seba/texstats`. We analyzed 15 LATEX documents originating from our research group

[7] This implementation is incomplete with respect to the typesetting of documents, but complete regarding TEX's macro facilities.

and ranging from research papers to master's theses. These documents amount to 301 pages of scientific writing and contain more than 300,000 macro definitions and 2,500,000 macro expansions. The high number of macro definitions, which might be surprising, is mainly caused by the use of libraries, e.g. amsmath.

Our analysis observes facts about the sample documents' uses of macro definitions and macro applications. To this end, a macro is considered constant if its definition has no declared parameter. A macro is called recursive if the transitive hull of macro identifiers used in its body contains the macro's own identifier.[8] Since the assumed recursive call not necessarily is executed, a macro may be considered recursive without actually being so. A macro has a delimited syntax if the parameter text in its definition contains characters or macro identifiers. A macro is redefined if a definition for the same macro identifier has previously been executed. A redefinition changes the arity of a macro, if the number of parameters in the previously executed definition differs. Similarly, the delimiter syntax of a macro is changed by its redefinition if the previously executed definition's parameter text contains different characters or macro identifiers, or a different order thereof. Additionally, arity and delimiter syntax changes have been recorded for macro identifiers not containing the string "temp". Lastly, a macro is said to be applied with higher-order arguments if at least on argument is a non-constant macro.

In the tables of Fig. 9, we state the absolute number of occurrences of the observed effects and their percentage relative to the respective category's total number of effects.

Our study shows that only 3.6% of all macro definitions and expansions use delimiter syntax. Furthermore, only 4.7% of all macro definitions correspond to redefinitions changing the redefined macro's arity, and 4.9% correspond to redefinitions changing its delimiter syntax. Nonetheless, most of these definitions redefine a macro with the string "temp" in its name. This gives reason to believe that the different versions of these macros are used in unrelated parts of the program, and do not entail conflicting constraints. Moreover, higher-order arguments are used only in 2.7% of all analyzed expansions.

The high number of redefinitions and constant macro definitions, 46.5% and 92.6%, is also interesting. This is in contrast to the number of expansions of parameterless macros, which is only 28.2%. We believe that this effect can be explained by the frequent usage of constant macros as variables, as opposed to behavioral abstractions. Accordingly, a redefinition of a constant macro occurs whenever the stored value has to change. This technique is often used for configuring libraries. For example, the LLNCS document class for this conference contains the macro redefinition

```
\renewcommand\labelitemi{\normalfont\bfseries --}
```

which is used in the environment *itemize* and controls the label of items in top-level lists. Any change of this parameter is a macro redefinition. Such macro re-

[8] For parsing, it is unimportant whether a macro is recursive. We included this property anyway because we found it an interesting figure nonetheless.

definitions cause no problems for parsing, since the protocol and parsing-related behavior of the macro does not change.

The presented data suggests that despite dynamic scoping, delimiter syntax and higher-order arguments, the syntactical behavior of macros often can be statically determined, because these language features, which are particularly troublesome for parsing, are rarely used. Therefore, we believe that the development of a practical syntactic analysis for TEX and LATEX is possible.

7 Applications

Besides giving a precise formulation of parser correctness, our formal model has valuable, immediate applications in practice. In the following we propose three tools easing the everyday development with TEX and LATEX.

Macro debugging. Our formal FTEX semantics is defined in form of a small-step operational reduction semantics. Consequently, it allows single stepping of macro expansions and FTEX processing. This support is of high potential benefit for TEX users since it enables comprehending and debugging even complicated macro libraries.

In order to apply macro stepping meaningfully to only part of a TEX document, the context in which the code is evaluated has to be fixed. One promising possibility is to scan the document for top-level library imports and macro definitions, and reduce the code in the context of those definitions. Similar techniques are already applied in some TEX editors, however not for the purpose of debugging.

Syntactic inconsistency detection. In Section 4.2, we introduced constraint generation which denotes the dynamic syntactical structure of FTEX code. Constraint generation cannot be used in static analyses, though, because it implies evaluating the program. In a programming tool, however, gathering structural information by running the program is a valid approach. Therefore, the instrumented FTEX semantics can be used to generate and identify conflicting structure constraints, which indicate accidental syntactic inconsistencies in the analyzed program and should be reported to the programmer.

Parser testing. Almost all TEX editors contain a rudimentary parser for TEX documents, so that syntax highlighting, for example, is possible. Often enough, however, these simplistic parsers produce erroneous syntax trees, essentially disabling all tool support. By taking advantage of constraint generation again, the parser can actually be tested. As stated in the definition of parser correctness, all generated constraints need to be satisfied by resulting syntax trees. The TEX editor can thus check whether its parser is correctly predicting the document's structure, and deliberately handle cases where it is not.

More generally, all generated constraints can be understood as test cases for user defined FTEX parsers. To this end, the constraint generator and the constraint satisfaction relation comprise a framework for testing FTEX parsers.

8 Conclusion

We have defined a formal model of TEX and have clarified formally what it means for a static parser to be correct. We have identified language features that are hindering syntactic analyses and have shown that provably correct parsers exist for subsets of TEX that exclude such features. Furthermore, we have given empirical evidence that this class of practical interest. We have also demonstrated how TEX programmers may benefit from our formal model in their everyday work, for instance by using a macro debugger.

We hope that this work will trigger a new line of research that will result in a broad range of tools for TEX users and, eventually, in a new design of TEX and LATEX according to modern programming language design principles which will remove the idiosyncrasies that today's TEX users have to suffer.

Acknowledgments. This work was supported in part by the European Research Council, grant #203099.

References

1. NTS: A New Typesetting System. `http://nts.tug.org/`. Visited on 20.03.2010.
2. G. J. Badros and D. Notkin. A Framework for Preprocessor-Aware C Source Code Analyses. *Software: Practice and Experience*, 30(8):907–924, 2000.
3. C. Brabrand and M. I. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. In *Partial Evaluation and Semantics-Based Program Manipulation*, pages 31–40. ACM, 2002.
4. V. Eijkhout. *TEX by Topic, A TEXnicans Reference*. Addison-Wesley, 1992.
5. A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *Automated Software Engineering*, pages 323–326. IEEE Computer Society, 2003.
6. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java - A Minimal Core Calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.
7. D. E. Knuth. *The TEXbook*. Addison-Wesley, 1984.
8. D. E. Knuth. *TEX: The Program*. Addison-Wesley, 1986.
9. E. E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. F. Duba. Hygienic Macro Expansion. In *LISP and Functional Programming*, pages 151–161. ACM, 1986.
10. L. Lamport. *LATEX: A Document Preparation System*. Addison-Wesley, 1986.
11. M. Latendresse. Rewrite Systems for Symbolic Evaluation of C-like Preprocessing. In *European Conference on Software Maintenance and Reengineering*, pages 165–173. IEEE Computer Society, 2004.
12. P. E. Livadas and D. T. Small. Understanding Code Containing Preprocessor Constructs. In *Program Comprehension*, pages 89–97. IEEE Comp. Society, 1994.
13. Y. Padioleau. Parsing C/C++ Code without Pre-processing. In O. de Moor and M. I. Schwartzbach, editors, *Compiler Construction*, LNCS, pages 109–125. Springer, 2009.
14. A. Saebjoernsen, L. Jiang, D. J. Quinlan, and Z. Su. Static Validation of C Preprocessor Macros. In *Automated Software Engineering*, pages 149–160. IEEE Computer Society, 2009.
15. D. Weise and R. Crew. Programmable Syntax Macros. In *Programming Language Design and Implementation*, pages 156–165. ACM, 1993.
16. A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.