Towards Interactivity for TEX

Joachim Schrod
Technical University of Darmstadt, WG Systems Programming
Alexanderstraße 10, D-64283 Darmstadt, Germany
schrod@iti.informatik.th-darmstadt.de

Abstract

Much work has been done to improve the level of interactivity available to TEX users. This work is categorized, and probable reasons are discussed why it is not really widespread. A more general view of "interactivity" may also lead to other tools. A common prerequisite for all these tools is the need to know about TEX's functionality. The description of TEX should be formal, since the available informal descriptions have not given satisfactory results.

After an abstract decomposition of TEX, an approach for the formal specification of one subsystem (the macro language) is presented. This specification may be interpreted by a Common Lisp system. The resulting Executable TEX Language Specification (ETLS) can be used as the kernel of a TEX macro debugger.

Variations on A Theme

"Interactive TEX" is the oldest theme on TUG meetings: Morris (1980, p. 12) reports that D.E. Knuth started his opening remarks at the first TUG meeting with it.

> [E]arly on he thought an interactive TEX would be useful, but finds now that TEX users internalize what TEX will do to such an extent that they usually know what TEX is going to do about their input and so have no pressing need to see it displayed on a screen immediately after the input is finished.

Already here a precedent is set for most future reflections on an interactive TEX: A user interface for an author is anticipated that gives feedback on the formatting of the document. Actually, many TEX users don't agree with Knuth, they want to see their formatted document displayed. With the arrival of WYSIWYG-class desktop publishing systems, some of them even want to get it displayed while they are editing, and effectively to edit the formatted representation.

It is worth noting that early usage of the term "interactive formatter" concerns mostly immediate feedback, i. e., the ability to see the formatted representation while the document is input (Chamberlin et al. 1982). In the TEX domain this approach was presented first by Agostini et al. (1985), still on an IBM mainframe at this time. Blue Sky Research invested work in that direction, their product Textures is now advertised as an "Interactive TEX."

The most advanced approach in the connection of TEX input with formatted output was explored by the VORTEX project (Chen and Harrison 1988). In principle, it was possible to edit both the TEX source and the formatted representation as the respective entities were linked to each other. A full implementation of this principle is done in the Grif system (Roisin and Vatton 1994), but this is not related to TEX.

The need to work with the formatted document representation was and is particularly motivated by the error-proneness of creating TEX input. Simple errors (e. g., forgetting a brace) occur very often and may lead to complaints in places that are far away from the error's source. In addition, the time lag between the creation of the error and the notification about it is too large for a smooth work flow. Direct manipulation (DMP) systems are environments that couple actions with reactions of the system and provide immediate feedback (Shneiderman 1983). They encourage one to create and change documents in an ad-hoc manner, without the need for much preplanning of abstractions and structures. (One may argue that this is a disadvantage for the task of writing; but this is not an argument I want to address in this article.)

It is important to emphasize that two terms mentioned above, WYSIWYG and DMP, concern completely different abstractions. A WYSIWYG system allows one to manipulate the presentation of a document; it concentrates on the task of creating and changing this presentation, it focuses on formatting. The term WYSIWYG is domain specific, it is tied to software systems that do layout (in the broadest sense, not only of documents). The category DMP is much more general and such on a different abstraction level: It classifies a set of interfaces that enables users to directly manipulate the objects they are working with, and where immediate feedback is given to them concerning these manipulations. DMP interfaces are often realized by means of windows, icons, menus, and pointing devices (e. g., mice); a member of this subclass of DMP interfaces is also

called a WIMP interface (Chignell and Waterworth 1991).

Since DMP interfaces have been shown to ease learning (Svendsen 1991), some approaches to yield interactivity for TEX document creation use separate systems with a DMP interface for document editing. They generate TEX input, to use the power of TEX's typesetting engine. These systems are pure front-ends, it's not possible to read TEX source and edit it. In early systems like Edimath (André et al. 1985) or easyTEX (Crisanti et al. 1986) the eventual target—the TEX language—is still visible. Newer systems like VAX DOCUMENT (Wittbecker 1989) or Arbortext's SGML Publisher hide this detail from the user. (It's quite interesting that these systems don't use TEX any more in the strict sense. Arbortext and Digital have modified the program to enhance its capabilities or to be able to integrate it better into the overall environment.)

Quint et al. (1986) noted early that such systems are, in fact, not TEX specific. Something that one can really call an interactive writer's front-end to TEX must be able to read TEX source, to enable not only the creation of documents but also their change. They presented the usage of Grif in such a context, but Grif is only able to understand a very limited subset of TEX markup. Similarly, Arnon and Mamrak (1991) presented the automatic generation of an editing environment for a fixed subset of plain TEX math, by formal specification of this subset.

But there are more usages of the term "interactivity". It is used often to characterize *TEX shells*, too. Developers recognized that the task of writing a document is more than editing and formatting; one has to handle bibliographies, create index, draw figures, etc. Tools are available for many of these tasks, but their existence and the respective handling (e. g., syntax of the command line options) has to be remembered. Environments that integrate these different tools into one coherent representation can release the author from that cognitive burden and can help to concentrate on the real tasks (Starks 1992). Sometimes such environments are labeled "interactive", in particular, if they have a WIMP interface. *Visible interface* is a better attribute for such systems as they do not provide a new level of interactivity—they merely make the current possibilities visible. (This terminology is due to Tognazzini (1992).)

As outlined above, the past has seen many attempts to increase the interactivity level of TEX systems for authors. Nevertheless the typical TEX user still writes the complete text with a general-purpose editor, not using any TEX-specific editing software. Even the low level of an immediate preview (or an early one, i. e., concurrently to TEXing) is not common in use.

The question must be posed why this happens. In my opinion, several reasons may be given:

- Some systems are very ambitious, actually they want to provide new publishing systems that replace TEX. Those systems that have been completed are proprietary and not freely distributable. Since they are not targeted to the mass market, they do not get the initial user base that would make them as widespread as TEX is today. The hypothesis that innovative non-mass systems will not be widespread without being freely distributable is backed up by HOPL-II (1993), the similarity between programming and authoring environments is assumed to exist in this regard.
- Those systems that restrict themselves to a certain subtask (e. g., editing of a formula) are often not prepared to communicate with other tools from the author's workbench. The developers often place unreasonable demands on authors (e. g., to place each formula in a separate file).
- Developers underestimate the inertia of users to stay with their known working environment. They are proud of their "baby", and often don't see that the benefit from their new system does not outweigh the costs of learning it. As an example, most UNIX users won't accept a TEX-specific editor that is not as powerful, flexible, and comfortable as Lucid (GNU) Emacs with AUC-TEX (Thorub 1992)—and that's hard to beat.
- Developers are unaware that there is more to interactivity than the creation of structured editing systems or full-blown WYSIWYG publishing systems. In particular, there exist more tasks in the production of a publication and there are lower levels of interactivity that are probably easier to implement.
- The TEX user interface (i. e. its markup language) is realized as a monolithic Pascal program together with a bunch of non-modular macros. It is not possible to incorporate parts of it (e. g., a hypothetical math typesetting module) into an interactive system. Each system rebuilds its needed abstractions anew, often incompatible with others and only approximating TEX's behavior.

Let's sort these issues out in the rest of this article. First, I will be more specific in the definition of "interactivity" and categorize different forms of it, thereby spotlighting interfaces that I think are needed and possible to create. Then preconditions for an easy realization of such systems will be shown, and the results of preliminary work to illustrate these preconditions will be presented.

On Interactivity

Interactivity means (1) that a user may control at run time what the system does, and (2) that feedback to the user's actions happens as soon as possible. If a user may just start a program and cannot control its progress, it is called a *batch program.* If a user may trigger an action at any time, even if another action is still running, the software system has an *asynchronous* user interface, and is regarded as highly interactive. Such user interfaces are usually WIMP-style, this is the reason why command-line oriented system are considered to have a lower level of interactivity—the user may act only at specific points in time, when asked by the system. Interactive systems are notoriously difficult to create, Myers (1994) has shown that this difficulty is inherent in the problem domain.

In the TEX area, we can identify at least the following forms of interaction that might be supported or enabled by software systems, to increase the level of interactivity available to users:

- Full-fledged publishing systems; with DMP, preferably WIMP-style, user interfaces
- Structural editing facilities for specific document parts
- Program visualization for educational purposes (e.g. training)
- Support for TEX macro development

This list is ordered by the additional abstraction level these interfaces provide and the difficulty of producing them. (Of course, the correspondence is not by chance.)

It seems that full-fledged publishing systems are the dream of many developers. But they tend to neglect two facts of life: Such systems have to be much better than existing ones, and their development will not succeed in the first attempt. Systems of the size one has to expect will never be written at once, they have to be developed incrementally. This is the case with all successful middle-sized software systems; it's worth to note that TEX is not an exception to that rule. (The current TEX is the third completely rewritten version, not counting TEX3, according to Knuth (1989).)

To improve on an existing system, one has to address at least the full production cycle. To be concerned only with the demands of authors is not enough any more; document designers, editors, supervisors, etc. work with documents as well. For instance, more appropriate help for designers can be supplied by better layout description facilities (Brüggemann-Klein and Wood 1992). Such facilities need better input methods as well, as designers are usually not trained to work with formal description methods. Myers (1991) shows convincingly that the paradigm of *programming by demonstration* may help here.

In addition, one must not restrict users too much, contrary to the belief of many software developers they are not unintelligent. That means that the straitjacket of pure structural editors, where everything *must* be done via menu and mouse, is not necessarily the right model to use. Research in programming environments (where such straitjacket interfaces did not succeed either) shows that it is possible to build hybrid editors that combine support for structured editing with free-format input, providing immediate feedback by incremental compilation (Bahlke and Snelting 1986).

Last, but not least, one should not forget to scrutinize persuasions we've grown fond of. For example, the concept of markup itself might be questioned, as shown by Raymond et al. (1993). Let's look outside the goldfish bowls we are swimming in, and build new ones.

If one does not have the facilities to produce a new publishing system, one might at least create tools that help users with specific tasks. Even on the author's task domain, one still needs editing facilities that fully understand *arbitrary* TEX math material or tables, and provide appropriate actions on them that are beyond the realm of a text-oriented editor.

Such tools must be able to communicate with other tools, preferably they should provide flexible means to adapt to different protocols. It's in the responsibility of the developer to provide the user with configurations for other tools to access this new one; the best tool will be tossed away if its advantages are too difficult to recognize.

In theory, tools for subtasks can also be used as building blocks of a complete system. In practice, this approach needs further study before one can rely on it. Good starting points for the management of such a tool integration approach are the ECMA standards PCTE and PCTE+ (Boudie et al. 1988).

Development of subtask tools is a hazard; it may be that one constructs a tool that will not be used because it does not enhance productivity enough. Therefore one should make provisions, so that the time spent for development should not be thrown away. The target should be a collection of modules that may be reused for further development projects. This must be taken into account very early, reusability is a design issue and cannot be handled on the implementation level alone (Biggerstaff and Richter 1989).

TEX is here to stay and will be used for a long time. Even the construction of a system that is ultimately better will not change this fact.[1] Experienced users must not forget the difficulties they had in

[1] It may be argued that TEX will be the FORTRAN or C of document markup languages—not the best tool available, but widely used forever.

learning TEX, even though they might by now have internalized how to prevent typical problems—as predicted by Donald Knuth. A topic of research might be the creation of systems that help to explore the functionality of TEX for novice users. For instance, the comprehension of the way TEX works may be made easier by program visualization (Böcker et al. 1986). A tool that visualizes the state of TEX's typesetting engine, allows one to trigger arbitrary actions interactively, and gives immediate feedback on state changes would enhance the understanding considerably. Similarly, advanced TEX courses will be able to make good use of a tool that visualizes the data entities of TEX's macro processor and allows the interactive, visible, manipulation of such entities. Such visualization tools may even be the kernel of a whole TEX macro programming environment (Ambler and Burnett 1989).

Let's not forget the poor souls in TEX country: those who develop macro packages and have to work in a development environment that seems to come from the stone age. This is not necessarily meant as a critique of Donald Knuth's program or language design, as it is reported that he did not anticipate the usage of TEX in the form it's done today. In fact, creating macro packages is programming; programming with a batch compiler.

But even for command-line based batch compilers (e. g., classic compilers for imperative languages) we're used to have a debugger that allows us to interact with the program while it is running. Each programming language defines an abstract machine, the state of this machine can be inspected and changed by the debugger. Execution of a program can be controlled by breakpoints, single stepping, etc. The debugging support available in the TEX macro interpreter is minimal. A first improvement would be an interpreter for the TEX macro language without the typesetting engine, since many errors already happen on the language semantics level.

Preconditions for Realizations

All presented aspects to increase interactivity have one need in common: they rely on access to information that one usually considers internal to TEX. Access to intermediate states of the typesetting process, values of the macro processor, etc., is crucial to build maintainable interactive systems. Since the production of reusable modules is also an aim, the access should not be by ad-hoc methods or heuristic inverse computations. Instead, well defined interfaces are preferred. Actually, before we may define interfaces we need a precise description what TEX "does" at all. In this context, precise explicitly means formal. Informal descriptions are not an adequate tool, after all we want to create a base of understanding, to be used as the underlying model and the terminology of module interface definitions.

The formal description must classify and categorize subsystems of TEX. It's important to take a system point of view in such a classification. Informal specifications that describe the functionality of TEX from a user's point of view exist—but they have not been of much use for the construction of further TEX tools. The target group of a formal description is different: it is not intended to be understood (or even read) by authors, software professionals will use it.

A System View on TEX

The classification of our formal description will be guided by a general model of TEX: It may be considered as an abstract machine. The v. Neumann model—processing unit, data storage, and control unit—is suited also as a model for TEX.

Here the *processing unit* is the typesetter engine, the parts of TEX that break paragraphs and pages, hyphenate, do box arrangements, transform math materials into boxes, etc.

The *data storage unit* is a set of registers that can save values (glues, boxes, etc.) for later usage. The storage and the processing unit work both with a set of abstract classes. These classes are basic elements of a "TEX base machine" abstraction, their instances are the things that are passed to the processing unit to parameterize its actions. We can see them as the primitive, assembler-level data types of the TEX computer.

The *control unit* allows us to access the registers and to trigger operations of the typesetter engine. In TEX, this control unit is hidden beneath a macro language. It is important to be aware of the fact that the macro language is *not* identical with the control unit, it is even not on the same semantic level. In particular, primitive types like boxes or even numbers don't exist in the macro language. (Numbers may be represented by token lists; i. e., the macro language handles only representations—sequences of digits—not number *entities*.) Furthermore, often the macro language only permits us to trigger many typesetting operations at once. These are typical signs of a high-level language.

In terms of our demand for a system-view specification of TEX this is important: *We don't have access to the assembler level of the TEX computer.* That implies that this level is not described informally in available documents; it must be deduced from fragmentary remarks in the TEXbook and there might even be more than one "correct" model of that level.

Eventually, we have a coarse categorization, a decomposition of TEX that is presented in figure 1. The typesetting engine and the storage unit are considered as a component, together with basic object classes. This *TEX base machine* is the basis of the system; in other environments such a component is called a *toolkit*. It is accessible through the control

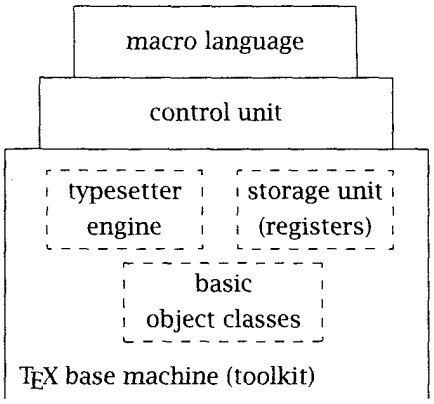

Figure 1: Subsystems of TEX

unit, the subsystem that allows the definition and evaluation of macro language primitives.

Actually, it might be of interest to compare the result of this data-driven analysis with the TEX modularization. (At least for the SAIL version a system structure is reported by Morris (1980).) Since Knuth used the method of structured programming, his modularization is algorithm-driven. The data-driven approach is preferred here since it will allow an easier isolation of subsystems.

Further work will have to analyze the subsystems, to identify modules thereof. The collection of module specifications will provide us with the formal description of the respective subsystem. The rest of this article will face only one subsystem: the macro language. A formal definition of it is useful if we want to export and import TEX documents into other tools.

Basic Terminology

Before the approach used to specify the macro language is presented, we must settle on a precise terminology that is needed for this presentation. While TEXnical jargon and anthropomorphic terms like "mouth" and "stomach" might make for some light and enjoying reading hour, I would like to use the dull terminology of computer science and introduce a few definitions:

Characters are read by TEX from a file. They are transformed to TEX-chars. With the transformation, a character disappears from the input stream and cannot be accessed further on.

Characters are not accessible at the macro language level.

A **TEX-char** is a pair (*category*, *code*). The code of a TEX-char is the `xchr` code of the read character (wlog. ASCII). The category is determined by the *catcode* mapping on codes. Sequences of TEX-chars are transformed to tokens; most often such a sequence is of length 1. If a TEX-char is transformed, it disappears and cannot be accessed further on.

TEX-chars are not accessible at the macro language level.

A **token** is a pair (*type*, *name*). A name is either an ASCII string or a character; strings of length 1 and characters are distinguished. A token is immutable, neither its type nor its name can be changed.

Token types are *not* TEX-char categories, even if they are often presented as such. The type of a token constructed from exactly one TEX-char is analogous to the category of this TEX-char. But there are categories that have no corresponding types and there is also one type that has no corresponding category. (This token type is symbol, a canonical term for the entities usually called control sequences or active characters.) Since we need to distinguish these two entities, we cannot use the same term for both (as done, for example, in the TEXbook).

In this document, we use the typographic convention (`type.name`) for token types.

An **action** is a tuple of the form (*semantic function*, param-spec *list*, *primitive*, *expandable*, *value*). It is a basic operation of the TEX macro language, the computational unit a programmer may use, the smallest syntactical unit of a program.

An action may be evaluated to trigger the respective semantic function. The evaluation of an expandable action results in a list of tokens.

An action has an associated parameters specification, the param-spec list. Each param-spec denotes a token list that conforms to some pattern. If an action is evaluated, an argument is constructed for each param-spec, in general by reading tokens from the input stream. These arguments are passed to the primitive.

In addition, an action may yield a value. The computation of the value may need arguments as well, the corresponding param-spec is considered part of the value tuple element.

Users can create new actions by means of macro definitions, the primitive tuple element is used to distinguish builtin (aka primitive) actions and user-defined ones.

In this document, we use the typographic convention action for actions.

A **binding** is a mapping *token* → *action*; every token has an associated action. The action bound to a token that is not of type symbol is fixed, it cannot be changed by the programmer. The macro language defines a set of bindings for symbol tokens, each token not in this set is bound to the action undefined.

These definitions allow a precise description how TEX processes its input:

1. A token is taken from the input stream.
2. The action bound to this token is determined.
3. The arguments for the action are constructed, as defined in the action's param-spec list. More tokens might be read from the input stream for this purpose.
4. The action is evaluated.
5. If the action was expandable, the result of the evaluation is pushed on front of the input stream. I. e., the next token taken from the input stream will be the first token of the result's list.

These steps are repeated until the action end is evaluated. The semantic function of this action will terminate the process.

A very good, and longer, explanation of the way TEX processes its input, may be found in a tutorial by Eijkhout (1991). In contrast to the explanations above, this tutorial takes a process-oriented view, whereas my analysis is data-centered.

Formal Language Specification

The TEX macro language (TML) has neither a common syntactic structure nor a "standard semantics", like those found in imperative or functional programming languages. The formal specification of such a language is not to be taken as an easy task; we are warned by Knuth (1990, p. 9):

> In 1977 I began to work on a language for computer typesetting called TEX, and you might ask why I didn't use an attribute grammar to define the semantics of TEX. Good question. The truth is, I haven't been able to figure out *any* good way to define TEX precisely, except by exhibiting its lengthy implementation in Pascal. I think that the program for TEX is as readable as any program of its size, yet a computer program is surely an unsatisfying way to define semantics.

Of course, one is well advised to take his statement seriously and to be specifically cautious in applying the attribute grammar framework. This difficulty is primarily caused by the inadequacy of context free grammars to describe the TML syntax in an elegant way, see below. Besides attribute grammars (Knuth 1968), other methods for formal language specification are the operational approach, axiomatic specification, and denotational semantics.

In the operational approach (Ollongren 1974), a transformation of language constructs to a prototypical computer model is done, i. e., the language semantics are explicated by construction. That is the earliest approach to define formal language semantics, it was used in the definition of PL/I. The method is particularly suited for languages that are to be compiled.

Axiomatic specifications, usually used for correctness proofs of algorithms, are also applicable to formal language definition; Hoare (1969) mentioned that already in his seminal paper. This approach has not been used often, due to the very complicated descriptions that result. Even Hoare and Wirth (1973) ignored hairy parts when they specified Pascal.

The denotational semantics method specifies a language by defining mappings of its syntactic constructs into their abstract "meaning" in an appropriate mathematical model (Stoy 1977). (Typically, that model is based on the lambda calculus.) The mapping is called the syntax construct's *semantic function.*

Since TML is not compiled, we will use a specification method that belongs to the denotational semantics category. First, we have to identify the syntactic elements of TML. The previous section explained that the computational model of TML is that of evaluation of actions, expanding macros as a side-effect. That implies that we can regard actions as top-level syntactic elements, there is no element that is created by combining several actions. Therefore we have to supply exact syntactic definitions for each action, supply the appropriate semantic function, and will get a full TML definition this way.

In previous work, the TML syntax was formulated partially by a context free grammar (in particular, in BNF format). Of course, the first approach is the incomplete specification given in the summary chapters of the TEXbook (Knuth 1986). Later, Appelt (1988, in German) tried to complete this grammar. Both show the same problems:

- The construction of a token may be configured by the programmer, via the catcode mapping. This is neglected in both grammars, they use exemplary notations for tokens. While this is described exactly by Knuth, Appelt somewhat vaguely introduces the notion of a *concrete reference syntax* for plain TEX (actually, the SGML term is meant) that he uses in his grammar.
- A rather large set of terminal syntactic categories is described by prose (36 in Appelt's grammar, even more in the TEXbook). Sometimes it's even not clear why these syntactical categories are terminal at all, e. g., a BNF rule for ⟨balanced text⟩ is easy to define and not more complicated to read than other definitions that are given.
- The difference between tokens and actions is not explained. Many syntactic structures don't look at specific tokens at all, they care only for the action that is bound to a token.

Most prominently, that happens with the definition of actions (*commands* in TEXbook

terminology) itself. If a terminal token like \parshape appears in the grammar, that does not denote the token (*symbol*."parshape")—an arbitrary token with the bound action parshape is meant instead.

Knuth start the presentation of his grammar fragments with a general explanation of this fact. In addition, every exception—when really a token was meant this time—is mentioned explicitly in the accompanying explanation. He even introduces the notion of implicit characters only for that explanation. (An *implicit character* is a token with type symbol where the bound action is an element of the set of actions bound to non-symbol tokens. By the way, the incompleteness of Knuth's prose definition clearly shows the advantage of formal definitions.)

Appelt even ignores that distinction: He uses token notations like '{' both for the description of a token of type begin-group and of an arbitrary token with the bound action start-group.

- If arguments for an action are constructed, they may be either *expanded* or *unexpanded* (the tokens that are collected will have been expanded or not). In fact, that is an attribute of a param-spec.

  Knuth notes only in the prose explanation which param-spec category is used for an argument; in addition, this explanation is scattered over the whole TEXbook. Appelt doesn't note this difference at all, e.g., in his grammar def and edef have the same syntax.

These examples should also show the value of a full formal language specification; discussions about the "structure" of a TML construct should not be necessary any more.

ETLS: The Executable TEX Language Specification

ETLS is a denotational semantics style language specification of TML. The mathematical model to which actions (the TML syntactic constructs) are mapped, is a subset of Common Lisp (CL). A set of appropriate class definitions for object classes from the TEX base machine is used as well. The computational aspect of the used CL subset (no continuation semantics or other imperative-style features) is well described and close enough to the lambda calculus to be used as a target model even in the usual sense of denotational semantics.

An action syntax is specified by a param-spec description for each argument. A param-spec is a pair (*expanded*, *pattern*). If a param-spec has the attribute expanded, all tokens that are used to construct an argument are fully expanded first. A pattern is either an identifier from a fixed set, or an alternative of a set of patterns, or an optional pattern. Pattern identifiers either denote a predicate function or an expression on token lists. The actual token list used as the argument will be checked by the predicate or matched by the expression.

Patterns defined by expressions on token lists (e.g. numbers) are specified by context free grammars. Of course, the specification of these parts must not ignore the problematic issues outlined in the previous section: Tokens are explicated as pairs, thereby providing a clear definition for grammar terminals. A special notation for "arbitrary token with a specific action bound to" is introduced. It can be ignored whether the token lists for the argument shall be expanded or unexpanded, though; this is mentioned already in the param-spec description.

A CLOS-style syntax is used for a full action specification. The param-spec list is given like a class slot list. The semantic function is the definition body. The additional attributes (primitive, expandable, and the value function) are put in between, like class attributes.

As an example, consider the specification of the action expandafter:

```
(define-action expandafter
    (:expanded-args
        (skip :token)
        (to-expand :token))
    (:primitive t
     :expandable-action t
     :value nil)
  "Expands the next-after-next token
     in the input stream."
  (cons skip (expand to-expand)))
```

Since this action is expandable, it has to return a list of tokens. That list is the replacement for the token this action was bound to and for the two argument tokens. We create it by prepending the first argument to the top-level expansion of the second argument. A value element of nil specifies that this action does not have any value semantics. (E.g., it is of no use as an argument to the action the.)

Action definitions like above may be embedded in a Common Lisp interpreter. That way we can interpret them directly and test if they have the same semantics as in the TEX processor. But it should be noted that these definitions do not trigger the same error handling as TEX—in case of an error condition they just signal an exception and the surrounding system must supply appropriate handlers.

Application of ETLS

Many people regard the formal definition of a programming language as an exotic goal pursued only by ivory-towered academics. But such work *is* practical and can even lead to immediate results.

As an example, consider the need for a TEX macro debugger. I.e., a tool that provides breakpoints with associated actions, stepwise execution,

tracing of particular macros and argument gathering, full access (read & write) to the state of the macro processor, etc. Everybody who has developed TEX macros at some time will have missed it.

The ETLS already realizes a large part of such a debugger. Since it is embedded in a Common Lisp system, the CL debugger can be fully applied to TEX macro processing. (In some of the better CL systems, even a GUI for the debugger is available.) If the result of an operation from the TEX base machine is needed, the ETLS limits are reached, though. But the implementation of some modules from this level (most prominently, the data storage unit with the basic object classes) allows us already to debug many typical error-prone macros. Of course, if typesetting problems have to be checked, one needs modules that do not yet exist.

The handling of syntax or static semantic errors is a further point where work is to be done. In case of an error, one is not greeted by the well-known "gentle" error messages of TEX, but is confronted with the Lisp fallback handler for a signaled exception. Then one can issue all kinds of Lisp commands (including the continuation of one's macro code). Of course, a better error handling, on the semantic level of a macro writer, can be easily imagined. (Traditionalists may want to have the TEX error loop available as well.)

Conclusion

Often the wish for interactive tools for TEX is mentioned. This covers author tools that can be used with arbitrary TEX documents, or developer tools that help to program in TEX and to understand the way TEX works. A precise description of TEX is a prerequisite for building such tools.

I have presented an abstract decomposition of TEX that sets an agenda for the specification of subsystems. In particular, one subsystem (the macro language) was further analyzed and an approach for its formal specification was presented. The resulting Executable TEX Language Specification (ETLS) is embedded in a Common Lisp interpreter and may be used to parse and partially interpret TEX source code. The immediate applicability of such an executable specification has been described as well, minimal effort is needed to enhance it to a TEX macro debugger.

Further work has to be done to add the (preferably formal) description of more subsystems. A first aim would be an analysis of the respective subsystems and the documentation of a modularization resulting from that analysis.

In addition, the utility of ETLS should be explored further. The TEX debugger needs the addition of error handlers to be of pragmatic use; a better user interface would be valuable as well. The semantic recognition of some substructures (e.g., the contents of haligns and formulas) is minimal and should be improved.

The work presented here is only a first step, but it may be used as the starting point to enhance interactivity for TEX users; though much remains to be done.

Technical Details & Administrivia. CLISP, a freely distributable Common Lisp implementation from the Karlsruhe University, was used for the actual realization of ETLS. CLISP has been ported to many platforms, Unix workstations, and PC-class microcomputers. No other Lisp system has been used until now.

Both systems are available by anonymous ftp from `ftp.th-darmstadt.de`. You find CLISP in the directory `/pub/programming/languages/lisp/clisp` (executables are there as well). ETLS is placed in the directory `/pub/tex/src/etls`.

Acknowledgments. CHRISTINE DETIG provided invaluable comments and helped to improve the document's structure.

References

Agostini, M., Matano, V., Schaerf, M., and Vascotto, M. "An Interactive User-Friendly TEX in VM/CMS Environment". In (EuroTEX85 1985), pages 117–132.

Ambler, Allen L. and Burnett, Margaret M. "Influence of Visual Technology on the Evolution of Language Environments". *IEEE Computer* **22**(10), 9–22, 1989.

André, Jacques, Grundt, Yann, and Quint, Vincent. "Towards an Interactive Math Mode in TEX". In (EuroTEX85 1985), pages 79–92.

Appelt, Wolfgang. *TEX für Fortgeschrittene*, Anhang "TEX-Syntax", pages 149–171. Addison Wesley, 1988.

Arnon, Dennis S. and Mamrak, Sandra A. "On the Logical Structure of Mathematical Notation". In *Proceedings of the TUG 12th Annual Meeting*, pages 479–484, Dedham, MA. TEX Users Group, Providence, RI, 1991. Published as TUGboat 12(3) and 12(4).

Bahlke, Robert and Snelting, Gregor. "The PSG System: From Formal Language Definitions to Interactive Programming Environments". *ACM Transactions on Programming Languages and Systems* **8**(4), 547–576, 1986.

Biggerstaff, Ted J. and Richter, Charles. "Reusability Framework, Assessment, and Directions". In *Software Reusability*, edited by T. Biggerstaff and A. Perlis, volume I (Concepts and Models), pages 1–18. ACM Press, 1989.

Böcker, Heinz-Dieter, Fischer, Gerhard, and Nieper, Helga. "The Enhancement of Understanding through Visual Representations". In *Proceedings*

of CHI '86 Human Factors in Computing Systems, pages 44–50, Boston, MA. ACM SIG on Computer & Human Interaction, 1986.

Boudie, G., Gallo, F., Minot, R., and Thomas, I. "An Overview of PCTE and PCTE+". In *Proceedings of the 3rd Software Engineering Symposium on Practical Software Development Environments*, pages 248–257, Boston, MA. ACM SIG on Software Engineering, 1988. Published as Software Engineering Notes 13(5), 1988.

Brüggemann-Klein, Anne and Wood, Derick. "Electronic Style Sheets". Bericht 45, Universität Freiburg, Institut für Informatik, 1992. Also published as technical report 350 at University of Western Ontario, Computer Science Department.

Chamberlin, D. D., Betrand, O. P., Goodfellow, M. J., King, J. C., Slutz, D. R., Todd, S. J. P., and Wade, B. W. "JANUS: An interactive document formatter based on declarative tags". *IBM Systems Journal* **21**(3), 250–271, 1982.

Chen, Pehong and Harrison, Michael A. "Multiple Representation Document Development". *IEEE Computer* **21**(1), 15–31, 1988.

Chignell, Mark H. and Waterworth, John A. "WIMPs and NERDs: An Extended View of the User Interface". *SIGCHI Bulletin* **23**(2), 15–21, 1991.

Crisanti, Ester, Formigoni, Alberto, and La Bruna, Paco. "EasyTEX: Towards Interactive Formulae Input for Scientific Documents Input with TEX". In (EuroTEX86 1986), pages 55–64.

Eijkhout, Victor. "The structure of the TEX processor". *TUGboat* **12**(2), 253–256, 1991.

EuroTEX85. *TEX for Scientific Documentation. Proceedings of the 1st European TEX Conference*, Como, Italy. Addison Wesley, 1985.

EuroTEX86. *TEX for Scientific Documentation. Proceedings of the 2nd European TEX Conference*, number 236 in Lecture Notes in Computer Science, Strasbourg, FRA. Springer, 1986.

Hoare, C. A. R. "An Axiomatic Basis for Computer Programming". *Communications of the ACM* **12**(10), 576–580, 1969.

Hoare, C. A. R. and Wirth, Niklaus. "An Axiomatic Definition of the Programming Language PASCAL". *Acta Informatica* **2**, 335–355, 1973.

HOPL2. *Proceedings of the 2nd History of Programming Languages Conference (HOPL-II)*, Cambridge, MA. ACM SIG on Programming Languages, 1993. Preprint published as SIGPLAN Notices 28(3).

Knuth, Donald E. "Semantics of Context-Free Languages". *Mathematical Systems Theory* **2**(2), 127–145, 1968.

Knuth, Donald E. *The TEXbook*, volume A of *Computers and Typesetting*. Addison Wesley, 1986.

Knuth, Donald E. "The Errors of TEX". *Software: Practice and Experience* **19**(7), 607–685, 1989.

Knuth, Donald E. "The Genesis of Attribute Grammars". In *Attribute Grammars and Their Applications*, number 461 in Lecture Notes in Computer Science, pages 1–12, Paris, FRA. INRIA, Springer, 1990.

Morris, Robert. "Minutes of the First TUG Meeting". *TUGboat* **1**(1), 12–15, 1980.

Myers, Brad A. "Text Formatting by Demonstration". In *Proceedings of CHI '91 Human Factors in Computing Systems*, pages 251–256, New Orleans. ACM SIG on Computer & Human Interaction, 1991.

Myers, Brad A. "Challenges of HCI Design and Implementation". *interactions* **1**(1), 73–83, 1994.

Ollongren, Alexander. *Definition of Programming Languages by Interpreting Automata*. Academic Press, 1974.

Quint, Vincent, Vatton, Irène, and Bedor, Hassan. "Grif: An Interactive Environment for TEX". In (EuroTEX86 1986), pages 145–158.

Raymond, Darrell R., Tompa, Frank Wm., and Wood, Derick. "Markup Reconsidered". Technical Report 356, University of Western Ontario, Computer Science Department, London, Canada, 1993. Submitted for publication.

Roisin, Cécile and Vatton, Irène. "Merging logical and physical structures in documents". In *Proceedings of the 5th International Conference on Electronic Publishing, Document Manipulation and Typography*, pages 327–337, Darmstadt, FRG. John Wiley, 1994.

Shneiderman, Ben. "Direct Manipulation: A Step Beyond Programming Languages". *IEEE Computer* **16**(8), 57–69, 1983.

Starks, Anthony J. "Dotex—Integrating TEX into the X Window System". In (TUG92 1992), pages 295–303. Published as TUGboat 13(3).

Stoy, Joseph E. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

Svendsen, Gunnvald B. "The Influence of Interface Style on Problem Solving". *International Journal of Man Machine Studies* **35**(3), 379–397, 1991.

Thorub, Kresten Krab. "GNU Emacs as a Front End to LATEX". In (TUG92 1992), pages 304–308. Published as TUGboat 13(3).

Tognazzini, Bruce. *Tog On Interface*. Addison Wesley, 1992.

TUG92. *Proceedings of the TEX Users Group Thirteenth Annual Meeting*, Portland, OR. TEX Users Group, Providence, RI, 1992. Published as TUGboat 13(3).

Wittbecker, Alan E. "TEX Enslaved". In *Proceedings of the TUG 10th Annual Meeting*, pages 603–606, Stanford, CA. TEX Users Group, Providence, RI, 1989. Published as TUGboat 10(4).