Cahiers GUTenberg

THE NTS PROJECT

Philip TAYLOR, Jiří ZLATUSKA, Karel SKOUPY

Cahiers GUTenberg, nº 35-36 (2000), p. 53-77.

<http://cahiers.gutenberg.eu.org/fitem?id=CG_2000___35-36_53_0>

The $\mathcal{NTS}$ project : from conception to implementation*

Philip TAYLOR [1], Jiří ZLATUŠKA [2] and Karel SKOUPÝ [3]

[1] Webmaster, RHBNC, University of London, United Kingdom ;
p.taylor@exch1.rhbnc.ac.uk
[2] Rector, Masaryk University, Brno, Czech Republic ;
zlatuska@muni.cz
[3] Independent programmer, Brno, Czech Republic ;
skoupy@informatics.muni.cz

Introduction

For today's talk, I had hoped that Karel Skoupý, the Czech implementor of $\mathcal{NTS}$, would be able to be here to present the results of his work and to answer any questions that you might have. Sadly, that will not be the case : Karel is working *desperately* hard to complete $\mathcal{NTS}$ before the commencement of the TUG 2000 conference, and I therefore have to deputize for him and attempt to answer any questions on his behalf.

Let me start by presenting an overview of today's talk and presentation ; I will attempt to cover seven separate areas, including (of course) the mandatory questions and answers at the end. The seven areas to be covered are :
– A brief history of $\mathcal{NTS}$
– TEX, ε-TEX & $\mathcal{NTS}$ compared
– The choice of Java as the language of implementation
– An overview of the classes, object and methods of $\mathcal{NTS}$
– A summary of the *status quo*
– A demonstration of $\mathcal{NTS}$, and comparison with TEX
– Questions & answers
and you will soon realize that my expertise lies very much in the earlier areas ; the *implementation details* of $\mathcal{NTS}$ are very much Karel's area, and I apologize in advance for any errors which I may make in presenting (in particular) the overview of classes, objects and methods.

*. The present authors would like to record their grateful thanks to all members of the $\mathcal{NTS}$ and ε-TEX teams, past and present, without whom neither this paper nor $\mathcal{NTS}$ itself could have ever come to fruition

1. A brief history of $\mathcal{NTS}$

The $\mathcal{NTS}$ project is the result of the foresight of just one man: Joachim Lammarsch, one of the founders – and for many years the President – of DANTE e.V. During the period leading up to the DANTE meeting in Hamburg of 1992, Joachim circulated a message to all potentially interested parties asking who would be interested in a project which was to continue where Knuth had left off. A number of us responded positively to this question, and those who did were invited by Joachim to attend the DANTE meeting in Hamburg.

During the course of the meeting, one of the scheduled sessions was devoted solely to "The Future of TEX". Those present debated at great length whether TEX should remain forever as intended by Knuth, or whether its future was too important to be determined by just one man (even one of the intellect and status of Knuth). The outcome of these deliberations was that TEX itself *must* remain solely Knuth's responsibility, but that a TEX-like system or systems should be created by an independent group, to continue developments whilst TEX itself remained frozen forever. It was also agreed that the name of this proposed new system should be $\mathcal{NTS}$ (an acronym for New Typesetting System) to indicate that this new system was *not* TEX itself, but was instead a new system which – whilst acknowledging unreservedly Knuth's rôle in its evolution – was free of the constraints which Knuth had placed on the evolution of TEX itself.

Once the main decision had been taken, other decisions followed more or less automatically. It was agreed, for example, that whilst DANTE e.V. would provide initial funding for the project, the project itself would be deemed to be transnational, transcending the artificial boundaries of any one TEX user Group and drawing its membership from TEX users and other interested parties throughout the world. Rainer Schöpf was invited to chair the group, and other members included Joachim Lammarsch, Joachim ("Johnny") Schrod, Bernd Raichle, Peter Breitenlohner, Jiří Zlatuška and myself (Philip Taylor).

Thereafter, the group met on a regular but occasional basis, almost invariably at such a time as to co-incide with a DANTE conference. The membership did not remain static, and Rainer stood down as Chairman after the first year, needing more time for other projects such as LATEX-3 and "real work" [tm]. I took over as Chairman, and we agreed that within the group two separate projects should be investigated, one evolutionary and one revolutionary. Peter Breitenlohner was to head the evolutionary ("ε-TEX") group, whilst Jiří Zlatuška would head the revolutionary group ("$\mathcal{NTS}$" proper). Joachim Lammarsch, as *fons et origo*, would remain as Managing Director, and Bernd Raichle, whose technical skills and TEX expertise were invaluable, was 2nd-in-command of both projects. Sadly we also said "farewell' to Joachim Schrod at about this time: Joachim's input and advice has been greatly respected and appreciated, but he felt unable

to agree with all of the decisions taken within the group and preferred to resign rather than be closely involved with a project with whose aims he could not entirely agree.

2. TEX, ε-TEX & $\mathcal{NTS}$ compared

2.1. TEX

There is surely little need in a talk addressed to members of GUTenberg to define what is, or is not TEX. TEX is, by definition, Knuth's program for performing typesetting of the highest quality, and this program is his and his alone. No-one other than Knuth himself may make any changes to the program (other than in the area of so-called *system dependencies*), and it is Knuth's publically stated intention that TEX should evolve no further: Don has made all the improvements to TEX that he deems necessary, and any further work which he does on TEX (at ever-increasing intervals) is restricted to eliminating any genuine bugs which have been discovered since he last updated the source. TEX is currently at V 3.14159, and Knuth wishes TEX to become *absolutely* frozen at the moment of his death, at which point it will be deemed to be V π.

2.2. ε-TEX

The ε-TEX project, which is as portable as TEX itself and which uses exactly the same tools and languages (Web, Pascal, Weave, Tangle, etc.), sought (and seeks) to extend TEX in a manner which is both conservative and innovative at the same time. It is conservative because it intentionally uses `tex.web` as the master source, and implements all changes through the medium of a change file, yet is innovative because it adds much-needed functionality to TEX and extends TEX in a way which is intended to meet the needs and demands of sophisticated TEX users who find themselves working at the very limit of TEX's abilities.

The ε-TEX project was conceived and (for some years) executed by members of the $\mathcal{NTS}$ group, under the leadership of Peter Breitenlohner and under the technical direction of myself. During this period, ε-TEX evolved from α- and β-releases via ε-TEX V 1 to ε-TEX V 2. By the time it had reached V 2.0, ε-TEX had added over thirty new primitives to the set already provided by TEX, and had extended the functionality of a number of others. Despite these extensions, ε-TEX was (and remains) 100 % TEX-compatible, and this, together with its portability, is surely ε-TEX's greatest strength.

Indeed, so important was compatibility considered when ε-TEX was being developed that – if no special action is taken when launching ε-TEX – it then

behaves *identically* to TEX itself, and with the sole exception of the banner line cannot be distinguished from TEX. It goes without saying that, in this mode, ε-TEX passes the so-called *trip test* with flying colours!

If access is required to ε-TEX's many extensions, then at the point of launch it is necessary to indicate this explicitly. This is accomplished (on command-line based systems) by launching Ini-ε-TEX with an asterisk where an ampersand would otherwise be allowed by TEX, as in

```
e-initex *e-plain \dump
```

as compared to (for example)

```
initex &plain \dump
```

The presence of the ampersand triggers ε-TEX into so-called *extended mode*, and this information is then stored in any format file which is dumped at the end of that ε-TEX instantiation. If such a format is then loaded into Vir-ε-TEX, the latter will then automatically start in extended mode, as in the following:

```
e-initex *e-plain \dump
e-virtex &e-plain <source file>
```

Once in extended mode, the user has access to all of ε-TEX's many extensions, yet – *if none of these is used* – ε-TEX continues to behave in a manner identical to that of TEX itself. Thus all legacy documents which do not, by accident, attempt to invoke one of the new ε-TEX primitives will behave and typeset *identically* under both TEX and ε-TEX.

But ε-TEX has one further *truc* up its sleeve: as well as compatibility and extended modes, ε-TEX offers a so-called *enhanced* mode in which *strict* compatibility is sacrificed in the interests of even greater functionality. As of V 2.0, ε-TEX possessed only a single enhancement: the implementation of TEX--XET, based on Knuth and MacKay's original TEX-XET but completely integrated within ε-TEX (and thus requiring no special IDV driver). Since the implementation of TEX--XET requires that maths nodes be overloaded, 100 %-compatibility *has* to be sacrificed, yet the differences are so subtle that most ε-TEX users who chose to exploit its enhanced mode would still notice no difference in output of their legacy (mono-directional) documents.

To enter enhanced mode, specific user action is required: the ε-TEX document being processed must specifically enable enhanced mode, either at the beginning of the document or at a point at which access to enhanced mode is required. For TEX--XET, this is accomplished by setting one of ε-TEX's so-called *state variables*, as in:

```
\TeXXeTstate = 1
```

In general, once ε-TEX is operating in enhanced mode, it is not possible to force it back into extended mode (enhanced mode can *only* be entered from extended

mode, never from compatibility mode). In certain circumstances, however, and in documents carefully written to localize all side-effects, it *may* be possible to cause ε-TEX to revert to extended mode. For the example above, this would be achieved by using:

```
\TeXXeTstate = 0
```

at a later point in the document, but users are cautioned that because of the asynchronous nature of (e-)TEX's page-breaking operations, there may still be some undesirable interactions if any modified maths nodes are still on one of ε-TEX's internal lists having not (yet) been flushed out. Thus for all practical purposes the user should assume that, once in enhanced mode, ε-TEX will remain in enhanced mode for the remainder of the instantiation.

One last point remains to be discussed under the heading of ε-TEX before passing onto $\mathcal{NTS}$ proper: with effect from ε-TEX V 2.1, Peter Breitenlohner assumed sole responsibility for ε-TEX. Peter has indicated that, while he still wishes to develop ε-TEX further, he no longer wishes to do so within the ægis of the $\mathcal{NTS}$ group, and with some considerable sadness we have acquiesced to his wishes. We wish Peter all the best with ε-TEX, and are confident that he will continue to maintain and support it with the same zeal and interest as he has in the past.

2.3. $\mathcal{NTS}$

For over five years, the $\mathcal{NTS}$ group were forced by circumstances to devote their attention almost exclusively to the development of the evolutionary system known as ε-TEX. This situation was brought about by the very nature of the group itself: it was composed entirely of volunteers, none of whom were in a position to expend great tracts of time on a project ($\mathcal{NTS}$) which was of little if any interest to their real employers. Recognizing that $\mathcal{NTS}$ could never become a reality if it was to be developed solely by volunteers working in their own time, the group decided that $\mathcal{NTS}$ should be put on ice until such time as funds could be found to allow a full-time programmer to be employed.

During 1997/98, that much-longed-for possibility became a reality. DANTE e.V. agreed to contribute the magnificent sum of DM 30 000 to the project, sufficient to allow a programmer to be employed full-time to work on the project. Jiří Zlatuška, as Dean of the Faculty of Informatics at Masaryk University (Brno, CZ), was in the fortunate position of not only being able to recommend to the group a highly competent programmer (Karel Skoupý) but also being able to arrange a tripartite contract to allow DANTE funds to be routed via the University and thence to Karel himself. We met with Karel, discussed the project with him, and despite the almost vertical learning curve which he foresaw would be required, Karel agreed to take on the task.

For some time, Karel did little but read. He read *The TEXbook*, *TEX-the-program*, a great deal about Java, and much else besides. Then, in the Spring of 1998, Karel and Jiří came to my home in England, and Karel outlined his proposals for $\mathcal{NTS}$. Jiří & I were much impressed with the expertise which Karel had clearly acquired, and with very few changes agreed that he should continue to develop his ideas. By the time we next met, Karel was to be in a position to demonstrate working code.

The next review took place in Brno, at the University, and on this occasion Joachim Lammarsch also took part. Joachim had greater familiarity with, and exposure to, Java than either Jiří or myself, and his presence at that review was invaluable. One of the most striking points which came out of the review was that Karel had elected to program for efficiency rather than for clarity, and there were a number of places where we felt obliged to ask him to re-think his approach (for example, we asked Karel to eschew the use of integers as general-purpose variables, and instead to use them *only* where integer arithmetic was required). Karel responded positively to our suggestions, although he clearly retained his reservations, and agreed to adopt our rather more defensive and didactic programming style.

When the contract with Karel was first discussed, all involved in the project believed that we could get from theory to a full implementation in one calendar year. As the end of the year approached, it became only too obvious that we had been (typically, many would say, in the IT/software world) very naïve in our analysis and far too confident in Karel's ability to complete the project on schedule. Indeed, by the end of the first year, although TEX's "mouth" had been re-programmed in Java, $\mathcal{NTS}$ was still unable to perform even the simplest typesetting, and an enormous amount of work clearly remained to be carried out.

Despite the obvious disappointment with which members of DANTE received the news that $\mathcal{NTS}$ would not be delivered on schedule, their confidence in the project remained on the whole unshaken and they generously voted to continue funding the project for a further period. During 1999, an anonymous benefactor pledged a further DM 7 500 to support the project (this benefactor, to whom the group are deeply indebted, is a private individual, not a user group or other corporate body), and at the 1999 EuroTeX meeting other TEX user groups also undertook to support the project financially. It is particularly pleasant to be able to thank the members of GUTenberg personally, since GUTenberg have pledged EU 3 000 for three years in support of the project. Thank you GUTenberg!

So what *is* $\mathcal{NTS}$, and why is it taking so long to reach completion? Unlike ε-TEX, which is conservative and evolutionary, $\mathcal{NTS}$ is truly *revolutionary* in

that it attempts (for the first time, as far as we are aware) to re-implement the algorithms and functionality of TEX-the-typesetting-system without in any way copying the coding (or even the data structures, though to a far lesser extent) of TEX-the-program. Whilst TEX is written in Pascal-Web, $\mathcal{NTS}$ is written in Java. And whilst TEX-the-program is a deeply entangled (though carefully structured) and highly daunting monolithic[1] program, $\mathcal{NTS}$ is intended to consist of a series of loosely coupled modules, any or all of which can be replaced by functionally equivalent module(s) with the same interface semantics.

The success of this latter approach was borne out fairly early on, since Karel wisely decided to cut his $\mathcal{NTS}$ teeth on the far less daunting task of re-engineering TFtoPL and PLtoTF in Java. The module which interprets the TFM file for the purposes of TFtoPL is *exactly* the same module as performs that function for $\mathcal{NTS}$, and thus "*software re-usability*" – that much-vaunted modern *desideratum* – has been achieved in practice.

Whilst the ultimate goal of $\mathcal{NTS}$ is to provide a complete and integrated, yet functionally distinct, set of typesetting tools, the short-term aim is to provide a complete re-implementation of TEX in a flexible and extensible manner. Early experience suggests that we are well on our way to achieving that aim, and (despite certain caveats that will appear later on) this has, in part, been achieved by the careful choice – and use – of programming language.

3. The choice of Java as the language of implementation

Right from its conception, the $\mathcal{NTS}$ project was not only a project which would concentrate on providing a new, more powerful, successor to TEX, but was also an effort to re-program TEX-the-program as such.

The reason for this was simple : TEX-the-program forms an example of a monolithic Pascal program based heavily on optimized data structures which allowed Knuth to cope with the memory limitations of computers in existence more than twenty years ago. At that time, the reasons for choosing both the lan-

1. Knuth would almost certainly take great exception to the use of the word *monolithic*, since he evidently took enormous care to divide the program into small and quasi-independent modules. Unfortunately, whilst the logic, structure and orthogonality of those modules is undeniable, the program as a whole is a masterpiece of efficiency, re-using code and/or data structures whenever possible, and as a result the program in totality is *very* difficult for others to modify or extend. Indeed, Peter Breitenlohner's ability to add functionality to TEX via the medium of a change-file is, as far as we know, the only significant attempt to extend TEX-the-program in any non-trivial way other than the equally significant but in many ways far more restricted changes made by Hàn Thế Thành in his PDF-generating variant pdfTEX.

guage and the programming techniques were understandable. Pascal developed from the family of procedural languages based on a formal syntax and well-defined semantics starting with Algol 58, Algol 60, and Algol 68, the latter providing one of the brightest peaks of the development of programing languages but which was unfortunately too complex to survive. Pascal appeared as a branch of development which combined the basic programming tools of structured programming as a methodology for programming, motivated by program correctness proof techniques and an approach to building large programs from manageably smaller components, and the tools for using abstract data structures instead of just the data types provided by the underlying computer hardware. Using a deliberately restricted set of constructions, Pascal could be seen as an abstract "machine code" for a general set of computers which also allowed for its portability and universal adoption both as far as implementations for various types of computers were concerned, and its general use as a language of choice for teaching programming.

For Knuth, Pascal was a language especially well-suited for expressing general-purpose algorithms in a way suitable for publishing. In the decades of the 60's and 70's, the issues associated with particular ways of expressing algorithms and developing proper programming style using high-level programming languages were among the the topics which formed core problems for research in Computer Science. Knuth added the concept of "literate programming" allowing the expression of his algorithms as a stream of constructions which follow the logic of the ideas needed for understanding a particular program construction, rather than the logic of the syntax of the programming language used – as Knuth stated his goal, to write programs in such a way that "instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do". Pascal as such did not suit Knuth's needs sufficiently, but literate programming tools based on macro generation allowed him to introduce an extension to Pascal's programming constructs (the otherwise clause in the case statement), to break the structure of declaration and code sections (using the "tangling" feature of the WEB system), and also to provide tools for allowing very careful memory optimization and implementation independence (using only integer data types and using macro constructions to generate code fragments necessary for the packing/unpacking of data used).

Within the two decades which followed the birth of TEX, the premises on which these decisions were based have become questionable, and indeed formed an obstacle to efforts to provide a successor to TEX which could extend the latter's capabilities sufficiently far. Pascal has been succeeded by a line of languages or systems which gained much smaller practical acceptance, and similarly to the fate of the Algol family, C and C++ have succeeded Pascal in its general

usage and/or implementation (and Knuth himself has moved to CWEB literate programming based on C which ultimately yielded a language that provided him with "indescribable joy" in programming). The need for careful packing of data into as little space as possible was removed by the emergence of computer architectures supporting much larger memory sizes and the practical availability and affordability of installed memory sizes unthinkable twenty years ago in any context other than perhaps secondary disk storage. Data structure optimization has become an obstacle preventing program modification and placing a kind of time bomb into the code which explodes when seemingly small and straightforward modifications need to be made. Similar problems with modifiability are caused by the monolithic structure of Pascal code as used by Knuth within TEX-the-program. Particular algorithms used within its body are extremely hard to modify or extend because of the lack of narrow and clearly defined interfaces between individual parts of the program code and because of the general accessibility of shared global data structures from within any part of the code.

The attempt to start $\mathcal{NTS}$ development has therefore been linked with a deliberate decision to re-create TEX-the -program so that the programming language and programming methodology allow for the removal of unnecessary data optimization, the removal of explicit storage allocation and the use of mechanisms already present in modern programming languages, and also to allow for a modular code structure with clearly defined interfaces and data paths which will allow for easier modification and for experiments with the resulting code. The idea of re-creating TEX using a more modern programming technique first came from Joachim Schrod, one of the inaugural members of the $\mathcal{NTS}$ group who also came with a prototype example of what he meant by such a re-implementation effort, showing how to extract the macro-generation language of TEX as a LISP program. The general work plan for $\mathcal{NTS}$ development effort has since then consisted of assuming that $\mathcal{NTS}$ version 0 would be created as a faithful 100 % (or for any practical purpose as close as possible to that) TEX-compatible program, and only from such code would further development activities continue by modifying and/or extending this code.

The choice of a suitable programming language for the re-implementation efforts had been a crucial unresolved problem until early 1998. Discussions oscillated around three different programming methodologies, each of which would provide a different set of advantages concerning the programming methodology, the availability of compatible implementations across a wide spectrum of hardware platforms and operating systems, and the existence of a sufficiently large base of programmers who would form the "brains trust" for future $\mathcal{NTS}$ extensions and experiments with different typesetting paradigms and user interfaces.

Functional programming as represented by LISP or CLOS (the Common LISP Object System) had been the language for Joachim Schrod's early attempts. The principal motivation for using this language consisted in the fact that lists of lists are the basic data structures manipulated within TEX, and hence the basic internal programming paradigm should be easy to represent. Symbolic data structures as used within LISP allow for easy meta-programming and prototyping, two techniques very handy for experimental development. Within university environment, LISP has traditionally been the language for implementation of experimental student projects with high programming productivity, and hence satisfying the essential requirements for viability of its use. Even though CLOS systems do provide a sufficiently stable and compatible implementation of this paradigm, the cross-platform compatibility is less than ideal.

Logic programming as represented by PROLOG or constraint-satisfaction programming languages based on expressing programs within a subset of symbolic logic has been another prototype language family with a high level of abstraction and a high productivity rate. Symbolic data structures (terms) allow for high flexibility in writing very easily modifiable code, and backtracking mechanisms used for underlying implementation of state-space search could provide interesting possibilities for searching for semi-optimal solutions of sets of constraints through which very complex conditions on the resulting typeset material could be expressed. Even though the programming structures are as far from the underlying hardware data structures as possible, the actual implementations of this paradigm vary significantly to such an extent that compatibility problems among the dialects make logic programming a very problematic choice if eventual cross-platform and cross-system compatibility is sought.

Procedural languages, C and C++ in particular, present a group of languages with considerable lower productivity in writing the program code and its resulting size. Even though the languages as such can be well-defined, practical differences as far as libraries included or operating systems interfaces make it a real mess to produce universally usable code which would run across different platforms with compatibility comparable to that of TEX itself. Another potential problem was seen in notorious problems with non-trivial modifications of programs employing access to general common shared structures, a common programming technique used in connection with these languages.

Eventually, Java has emerged as a compromise satisfying many of the essential requirements, offering interesting future opportunities, and being complicated by relatively few drawbacks. As far as programming methodology is concerned, Java combines C-based procedural programming with object based programming style. Objects serve as the basic program components allowing for structuring clean interfaces between separate components of the resulting program

and the development of future modifications by the substitution of certain objects of which the programs consist by other components. Objects also serve as a consistent replacement of traditional data structures and thus remove the traditional drawback of global shared data structures. Sun came up with Java as a company-based standard providing uniform system interfaces and a high level of security of Java applications. These claims remain to be demonstrated in reality and not just as wishful thinking and bold P.R. statements, yet the development of tested and certified Java interpreters incorporated into ubiquitous WWW browsers made it highly probable that the requirement of general compatibility could be achieved and a significant base of Java programmers formed. Last but not least, Java as a WWW-based and Internet-aware language make it possible to think of $\mathcal{NTS}$ as a network-based program which will eventually allow the combination of elements downloaded from the network and the standardization of interfaces and techniques used across huge groups of geographically dispersed users.

The trend associated with the network as an important element of the future computing environment has contributed to the choice of Java as the $\mathcal{NTS}$ implementation language. Karel Skoupý joined the $\mathcal{NTS}$ team in early 1998 as the programmer whose initial task has been to deconstruct TEX-the-program into an object-based programming code preserving the essential functional features of TEX as such but providing the grounds for future modifications and extensions.

After some 18 months of (re-)design and programming, a functional prototype of core components of TEX has been developed and made accessible for initial experimentation; early results were presented at previous conferences (e.g. EuroTeX'99) and it is hoped that the final code for $\mathcal{NTS}$ V0 will be demonstrated at TUG 2000. What you will see today is very close to that code!

4. An overview of the classes, objects and methods of $\mathcal{NTS}$

The implementation language of $\mathcal{NTS}$ is Java. It is strictly object oriented, and all of the program code is encapsulated in object methods. The objects are instances of certain classes, and cluster together to form packages, discussion of which forms the majority of what follows. At the time of writing, not all packages were complete; in particular, mathematics remains to be implemented, although the majority of the design work for this remaining task is virtually complete.

4.1. Package base

The main purpose of this package is to define the elementary data types used in the rest of the system. It is a minimal element of the $\mathcal{NTS}$ package hierarchy, meaning that no class here is dependent on any classes from other $\mathcal{NTS}$ packages.

The most important classes are as follows:

Dimen represents a dimension measured in printers' points (or mu units). The fact that the internal representation is the same as that of dimensions in TEX ensures the strict compatibility needed. The public interface of this class tries to be completely independent of its internal representation. To the outside world, a Dimen looks like a fraction of points. There are methods for conversion from integer, from fraction (given by its integral numerator and denominator), from floating point number, and *vice versa*. It also provides basic arithmetic operations. In the case of binary operations, versions for combination with other convertible numeric types are supported too. The general public interface allows for a complete change of the precision or even the internal unit of representation without affecting the other code.

Glue reflects another type familiar from TEX. It has its natural dimension and the amount and order of stretchability and shrinkability. It provides arithmetic methods such as adding two Glues, multiplying by a scalar number and also versions for other convertible types.

Num represents an integral number. It is just an integer, but wrapped into an object so it can be stored directly in the table of equivalents and can be distinguished from ordinary integers in the code. It serves mainly as the representation of the value of numeric registers (`\count`) (and is somewhat symmetric to Dimen and Glue for `\dimen` and `\skip` registers).

All the basic classes above provide methods for obtaining character string representations which can be displayed on screen, in the log file or used by the `\the` primitive.

LevelEqTable is the last important and relatively complex class. It is used to implement TEX's table of equivalents and the hash table. Whilst TEX uses an associative hash table only for the meanings of control sequences, $\mathcal{NTS}$ stores many other kinds of equivalents in an associative manner. Any object can be associated with a particular combination of kind and key. Different kinds are defined for different types of equivalents: one is for control sequence meanings, another is for each class of register, still another for catcodes, etc. The key is (according to the kind of equivalent) either an object (e.g. a control sequence name) or a number (most others). This associative approach for storing register values naturally avoids the limitation on any particular number of registers.

Although $\mathcal{NTS}$ is compatible with TEX in providing only 256 registers of each sort, this limitation is artificially added and can be easily removed in the future.

As the name suggests, besides storing equivalents, the LevelEqTable also maintains pushing and popping of levels which are result from grouping in the input language and the corresponding saving and restoring of associated values.

Although the registers were moved from a static array to an associative table, there is still another type of value which is not associative but which is subject to saving and restoring. These are parameters (such as `\tolerance`, `\hsize`, ...) – the current value of a parameter is stored in one concrete place. The LevelEqTable provides an interface for these external equivalents too, and maintains saving and restoring for them.

4.2. Package io

This package contains classes and interfaces for reading characters from an input file and writing to the log file. Either or both of those files may equally well represent the user's console. The package is independent of the other $\mathcal{NTS}$ packages as well as of the package base.

CharCode is an interface and is very interesting (at least, we think so!). There had been considerable discussion as to whether or not to represent character codes by some Java primitive type or by a class. It was decided that a class should be used do as to clearly distinguish it from other usages of primitive types. Eventually (during development), it turned out that an even more abstract representation (as an interface) best matches its purpose. It declares methods for getting the corresponding character or numeric value, comparing with another CharCode, character or number for matching, making the corresponding uppercase or lowercase CharCode, writing on a character-oriented output file and several predicates. Most of the methods are there because the TEX language uses characters heavily not only for typesetting but also as numeric values and as parts of keywords; in addition, certain characters have an influence on scanning and on log output (`\endlinechar`, `\escapechar`, `\newlinechar`).

Currently the implementation of CharCode used in $\mathcal{NTS}$ is just a class containing an ordinary character. But there exists the possibility to use very different representations (e.g. named characters) without changing any $\mathcal{NTS}$ code. Such objects can pass through the whole system provided that at the end there exists an output object which recognizes them and handles them properly. Even several independent implementations of CharCode could co-exist in some future application.

Name has the same relation to CharCode as String does to char. It is used to represent the names of control sequences, `\jobname`, fontnames and file-names scanned from the input.

InputLine represents one line from the input file or from the user's console. There are methods for getting the next CharCode or just peeking to see what the next code is without altering the current reading position. It interprets extended character codes (such as `^^M`), ignores trailing blanks and appends `\endlinechar` if needed. Another class, LineInput, serves as an input sequence of InputLines from a file or console.

Log is an important interface for printing information on a log file or on the user's console. It declares methods for printing values of primitive types, Strings, CharCodes and Loggables (see below). Several methods are declared to control output line breaking. Class StandardLog implements the Log interface in the standard TEX way.

Loggable is a very simple interface which declares one method for printing on a Log. It is very handy because most of the important classes in $\mathcal{NTS}$ implement this interface and so their logging is conveniently handled.

4.3. Package command

Classes in the package command form the interpreter for the TEX input language. Although it is a large package, it has nothing to do with typesetting *per se*. In fact, at least one third of the TEX source is not about typesetting at all. It is responsible for the process of scanning input tokens, expanding them and for most of the mode-independent processing such as macro definitions and register assignments.

Token is an abstract class. It declares methods for getting the meaning of a Token, assigning a new meaning (if allowed), matching another Token, and a number of predicates which tell if it is redefinable, is a brace, a letter, and so on. There are several kinds of Tokens, and they form a small hierarchy of subclasses. Typical examples include CtrlSeqToken, ActiveCharToken, SpaceToken, LetterToken, LeftBraceToken, ...

Tokenizer is able to provide a sequence of Tokens. There are various subclasses of Tokenizer such as : LineInputTokenizer for tokenization of the input file, MacroExpansion for macro bodies with supplied parameters, InsertedTokenList for a token list from a token register inserted into the input stream, or BackedToken for just one backed-up token. Tokenizers are pushed onto a TokenizerStack – the analogue of TEX's input stack.

Command is an abstract class which represents each TEX command. Mostly the commands are primitives, each of which is registered under its name in the

table of equivalents, but there are important exceptions such as Macro or the meaning of a character. In TEX, tokens and command codes are represented by the same type and they are often interpreted in both ways which may lead to confusion. $\mathcal{NTS}$ strictly separates the concept of token and command. As outlined above, a Token is a piece of input which can have some meaning. The type of this meaning is the Command discussed here.

A Command has methods for execution and expansion. Only some Commands can be expanded, and this property is indicated by another predicate method. The heart of $\mathcal{NTS}$ is a cycle very similar to TEX's `main_control`. In one step, a token is fetched from the input and its meaning is examined. If it is expandable, the appropriate method for expansion is called. If there is some result of expansion, the method is responsible for pushing it onto the input stack. If it is not expandable, the method for executing the command is called.

There is one curious fact about expandable commands: they are executed if their expansion is suppressed by `\noexpand`. In this case they behave exactly like `\relax` (they do not do anything apart from re-setting TEX's internal state and terminating any active look-ahead) and they even pretend that they are `\relax` when examined by `\show`. For that reason, the whole subtree of expandable commands is derived from the Relax command.

Another important part of the Commands interface are the methods used for indicating availability and getting some value of a certain type. It is useful when the command occurs on the right-hand side of an assignment, for example, and therefore the registers, parameters and a few others can provide numeric, dimension, glue or token list values.

CommandBase is a superclass of Command. It defines only static methods which are related to scanning various elements of input (such as numbers, dimensions, file names, keywords, ...), maintaining the table of equivalents, input stack and several instances of Log output. As we will see later, there are more objects than just Commands which require such services and so are derived from this abstract class for convenience.

4.4. Package node

Now at last we are getting to typesetting! The classes in this package represent material to be typeset. There are also general interfaces to font metrics and output generators. The package is relatively low in the hierarchy, the classes are dependent only on the base and io packages. That gives them a good chance to be re-used in a different typesetting system which may provide a completely different input language or user interface.

Node is an interface which defines the elementary building block of typesetting material. It has methods to get its sizes (even when affected by some stretching or shrinking), to describe itself on a logfile and to be typeset. There is a hierarchy of classes which implement the Node interface. Some of these are elementary, for example: RuleNode, HKernNode, VKernNode, HSkipNode, VSkipNode, PenaltyNode; other objects are complex and can contain lists of subsidiary nodes: HBoxNode, VBoxNode.

Packer is used when we need to compute the sizes of complex boxes which are built out of lists of nodes. This process is called packaging in TEX. The algorithm is essentially the same for horizontal and vertical lists of boxes, only the horizontal and vertical dimensions are flipped for different cases. The abstract class Packer defines an abstract algorithm and provides a placeholder to get the appropriate box dimensions. There are then special subclasses for horizontal and vertical boxes which are in turn sub-classed outside this package to give the right kind of warnings if something is not decent.

FontMetric is an abstract interface for font metric information objects. At the moment, there are only the familiar tfm files but that is not a restriction of $\mathcal{NTS}$ – it is prepared for any kind of font metric which can be adapted to this interface. There are methods for getting an identification and various numeric or dimension parameters for TEX compatibility. But first of all there are methods to get a Node for a particular CharCode, to get a normal inter-word space and to get a special object which is able to produce the representation of characters, ligatures and kerns for a given sequence of CharCodes.

TypeSetter has similar characteristics to FontMetric. It defines an interface for general typesetting output. There are methods for typesetting a character or a rule at the current position and adjusting this position.

4.5. Package builder

This package takes care of the areas concerned with TEX's horizontal, vertical and maths modes. Whilst in TEX there is just one global integer variable which indicates one of the seven possible modes (the three mentioned above are each internal or external, and there is one "no mode") on the top of the semantic stack, $\mathcal{NTS}$ uses objects which build typesetting material for different modes.

The package is more dependent on the TEX paradigm than is node but is still independent of the TEX language. It is relatively simple and small. Some amount of complexity must be solved when typesetting commands and different modes interact but that issue is addressed in another package.

Builder is the root of the hierarchy of classes for different modes. It declares some predicate methods for getting certain characteristics of a given mode and

methods for adding a node, kern or skip to the list of nodes which is currently being built. It makes the appropriate versions (horizontal or vertical) of kerns and skips and performs other adjustments if needed. Currently only the modes known from TEX are supported but there is provision for other types of mode (chemical, picture, ...).

4.6. Package typo

The package typo is a superstructure of the package command. It contains all the Command subclasses which deal with typesetting currently developed (there will be a package maths for mathematical typesetting commands but it did not exist when this text was written). It utilizes the packages builder and node as well.

TypoCommand is similar to CommandBase but is intended for typesetting commands. It is an intermediate abstract Command class which defines several useful static methods. It maintains a stack of Builders and the current font metric. There are methods for scanning a font metric or box specification from the input, and adding a character or space to the current Builder.

Many classes in this package are derived directly from classes in the command package because they can inherit some useful behaviour from them. They cannot be included in the command package because they need some information which is available only in the typo package (usually by calling some static method of TypoCommand). There are basically two kinds of these: one is `\if` primitives such as `\ifhmode` or `\ifvbox` which just need some information about the current Builder or a certain box register; another is the commands which are mode independent but typographic such as `\setbox`, `\wd` and `\chardef`.

BuilderCommand is an abstract superclass for commands which are mode dependent. In an open system such as $\mathcal{NTS}$ we want new features to be capable of being easily added. There is, for example, a superclass Command which defines a particular set of methods which can be implemented by the new commands in any sensible way. This kind of polymorphism is directly supported by the chosen programming language.

But what to do if in future we want to add a new mode by developing a Builder which offers some new functionality not declared in the Builder interface and some specialized commands which can utilize this new functionality? If we do not want to extend the basic interface (at least until a new version) or even perhaps cannot do it (we are developing a plug-in), the only chance is to examine the type of the current Builder and to use the infamous cast operator if it is our new Builder.

But there is another problem with existing mode-dependent commands. How should they behave in the new mode? For this purpose, the BuilderCommand maintains a hash table which associates an Action with each combination of Builder class and Command instance. The association is defined at the level of the $\mathcal{NTS}$ configuration and it automatically follows the class hierarchy of Builders. Thanks to this versatility, it is very easy to specify the behaviour of commands in different modes for modified systems.

Action is a subclass of CommandBase so it inherits many methods for scanning the input, dealing with log files and error messages. Actions are usually implemented as inner classes of the corresponding BuilderCommand.

A fragment of the $\mathcal{NTS}$ configuration data looks like:

```
RulePrim hrule = new RulePrim
    ("hrule", default_rule, Dimen.NULL, Dimen.ZERO,
                                            Dimen.ZERO);
RulePrim vrule = new RulePrim
    ("vrule", Dimen.NULL, default_rule, Dimen.NULL,
                                            Dimen.ZERO);
hrule.defineAction(VertBuilder.class, hrule.NORMAL);
hrule.defineAction(ParBuilder.class, hrule.FINISH_PAR);
hrule.defineAction(HBoxBuilder.class, hrule.BAD_HRULE);
vrule.defineAction(HorizBuilder.class, vrule.NORMAL);
vrule.defineAction(VertBuilder.class, vrule.START_PAR);
```

The BuilderCommand corresponding to the TEX primitive `\hrule` defines three actions: it performs the normal operation in vertical mode, finishes the current paragraph (if any) in horizontal mode and complains inside an `\hbox`. The `\vrule` performs normally in any horizontal mode and enters a new paragraph in vertical mode. There is in fact only one class (RulePrim) which has two instances with names hrule and vrule and different parameters; they are assigned different Actions for the same modes. All the Actions NORMAL, START_PAR, FINISH_PAR and BAD_HRULE are instances of inner classes inside RulePrim or its superclass.

Other examples of BuilderCommand are: HBoxPrim, VBoxPrim, VTopPrim, LowerPrim, MoveLeftPrim, BoxPrim, KernPrim, CharPrim, ExSpacePrim, AccentPrim, AnySkipPrim.

Group is another subclass of CommandBase. Its subclasses cover the various types of group in TEX. There are groups such as SimpleGroup for a pair of braces, SemiSimpleGroup for the `\begingroup` and `\endgroup`, HBoxGroup, VBoxGroup or VTopGroup. Group itself is defined and the stack of Groups is maintained in CommandBase but most of the subclasses belong to the package typo.

Groups have one problem in common with Builder. Their closing commands behave differently in combination with certain type of Group. The right brace cannot match `\begingroup` and `\endgroup` cannot match the left brace. The problem is solved in exactly the same way as for combinations of commands and Builders.

4.7. Package tfm

The package tfm implements a particular type of font metric information – the TEX font metric file – for use in $\mathcal{NTS}$. It can be used as an example for implementing other types of font metric.

TeXFm is a class which represents the low-level – almost raw – format of a TEX font metric file. Some complications are hidden but its public interface reflects just the information which is available in the file. It uses several auxiliary classes because the `tfm` format is too complex to be captured by only one program file. As an example, the whole process of reading a `tfm` file is done by the class TeXFmLoader which creates an instance of TeXFm. TeXFm itself has methods for getting information concerning the characters, ligatures and kernings for pairs of characters, extensible recipes and sequences of enlarging characters. Another method is provided for printing its representation as a property list. This is used by a small Java application `tftopl` which – thanks to TeXFm – shares most of the code with $\mathcal{NTS}$.

TeXFontMetric is an adaptation of TeXFm which implements the FontMetric interface from the node package. It is a wrapper which uses the natural methods of TeXFm and provides the methods required by the rest of $\mathcal{NTS}$. This approach is probably useful for future implementations of other types of font metric. We cannot expect that some third party will provide the exact interface even if a Java class is supplied for access.

4.8. Package dvi

This package implements the `dvi` format as one of the possible output formats for $\mathcal{NTS}$. In many aspects it will be similar to the package `tfm` but it was too early to say more as development of this part had just started when this text was written.

4.9. Package tex

This package is an umbrella for the other $\mathcal{NTS}$ packages, and it is by far the messiest part of the system. All the classes and packages so far are designed to provide a clear and elegant interface and to be as independent of other

classes and packages as possible. But in TEX itself, there are so many unclear dependencies. That was one reason for starting the whole $\mathcal{NTS}$ project in the first place. The classes in this package join all the independent units together, and in addition all the weird cases were exported from the clean design of other packages to here if this was possible. That is the main reason that the code sometimes looks rather messy here.

Besides this, there are classes for maintaining the error pool so that commands are not dependent on the way in which the error messages are given.

The most interesting part of this package is the class Primitives which contains the configuration of the whole system. There was already an example in package typo.

4.10. Modularity and configurability

To develop a system which is as modular as possible was one of the main desiderata. In the current TEX implementation, there are a lot of dependencies. Experience shows that it is very difficult and dangerous to make some non-trivial changes since these can lead to a number of possibly unclear side-effects. The approach taken in developing $\mathcal{NTS}$ has been to make all dependencies explicit and clear. All classes have a well-defined interface of public methods which is used for all communication. There are no uncontrolled changes of global variables. This manner of programming is greatly supported by the Java object-oriented language.

Another motivation for making code units independent is to allow substitutions of some modules by other modules with the same interface but a different underlying implementation. Independent classes or packages can also be used as building blocks for another system. The $\mathcal{NTS}$ packages are therefore designed rather as class libraries with a strict hierarchy.

An interesting problem concerned with the decomposition of TEX into independent units is the problem of cyclic dependencies. There are many of them. A simple example is the relation between TEX's "eyes" and "stomach". The stomach is fed by commands which originate at the eyes, but the action of the eyes depends on `\catcode` settings which originate from the stomach.

This makes it particularly difficult to maintain a non-cyclic hierarchy of packages. On the other hand, it is very desirable if we want to use only some of them in another application. The method that $\mathcal{NTS}$ uses to avoid such cyclic dependencies is via abstract interfaces. If some class needs information or an action which is not available at the current level of hierarchy, it defines an interface and accepts an object which implements it as a parameter (of its

constructor or of some method). The parameterisation is then made at some higher level, usually in the umbrella (or maybe $\mathcal{NTS}$'s brain) – the package `tex`.

5. A summary of the *status quo*

$\mathcal{NTS}$ was envisaged (more than a little naïvely, as has already been suggested) as taking one year from commencement to full implementation. It is now two years since formal commencement, and work is not yet complete. How far have we got, and what were the reasons for the delays?

The good news is that work is very nearly complete: Karel has tackled the task in a very logical order, starting with TEX's "eyes" and "mouth" (the scanner and tokeniser), then macro expansion, then command execution where this did *not* involve typesetting, through to list creation, and page-building. $\mathcal{NTS}$ is now able to process and typeset (that is, generate `DVI`) for any document which does not involve mathematics or alignments, although it cannot (at the time of writing) yet hyphenate words. In fact, only three real challenges remain: mathematics (mathematical typesetting, of course, rather than mathematics *per se*), alignments and hyphenation. Karel has already completed a large part of the research and design phase for these.

However, there is another area in which some work remains to be carried out, and that is the area of system interactions. Of course, TEX itself does not interact with the system in any potentially dangerous way (with the notable exception of being able to open an arbitrary file for writing, provided that the user running TEX has appropriate permissions). But TEX *does* interact with the environment in rather more subtle ways, for example to ascertain the path or paths which it will search for each class of file (`\input` files, `.tfm` files, and so forth).

Most implementations of TEX perform this interaction through the medium of so-called *environment variables* (e.g. `TeX_Inputs`, `TeX_Fonts` and so forth). These environment variables are typically set by the installer of TEX for a given system, and can usually be modified by individual users to suit their particular needs. Whether these environment variables are actually variables, or logical names, or part of (e.g.) a Windows NT environment settings is irrelevant to the user: all that matters is that there is a standard way (standard, that is, for each platform and implementation of TEX) of informing TEX where the relevant files are to be found.

The problem is that Java is a *portable* language. And truly portable languages must behave identically no matter on which platform they are installed. And so the designers of Java have decreed that, since environment variables are

not standardised across platforms, Java shall have no access to environment variables. Disaster!

It therefore looks at the moment as if $\mathcal{NTS}$'s environment will have to be configured independently to that of TEX, using a Java-specific configuration system, and there will be no way of allowing $\mathcal{NTS}$ to inherit TEX's run-time environment settings. But this area is still under review, and it is still possible that some satisfactory compromise will be found. Recent improvements to the Java system have acknowledged the need for a so-called *policy file*, which by default is ignored but which – if permitted by the security settings – can be read by the Java run-time system during initialisation. Such a file could be generated in a very straightforward way from existing environment variables, although (for obvious, bootstrapping, reasons) the program to generate it could *not* be written in pure Java!

So mathematics, alignments, hyphenation and environmental enquiries remain to be implemented, virtually all else is complete; how satisfied are we with the work done so far?

In general, we are extremely satisfied; Karel has done an excellent job of re-engineering and re-implementing a TEX-compatible system in a modular and open way. Compatibility remains uncompromised: the `DVI` files and log files (and even the console output) of $\mathcal{NTS}$ and TEX are *identical* (obviously modulo such necessary differences as the NTS banner reading "This is NTS" rather than "This is TeX").

But there is also one area about which we are deeply concerned, and it is only fair that we should reveal our concerns to the sponsors of the project (such as GUTenberg). That area is *performance*. And the performance is *abysmal*.

When we first went to Knuth with our plans for $\mathcal{NTS}$, we said that we intended to perform the re-implementation in two phases: phase-1 would use a modern, rapid-prototyping, language to validate the design; the second phase would involve a further re-implementation using a language selected for efficiency. Don reassured us that this second phase would never prove necessary: "by the time you are ready to perform the second re-implementation, technology will have advanced so much that a second re-implementation will not be needed. Computer performance continues to rocket, year after year, and shews no signs of starting to reach a plateau" is a paraphrase (from memory) of Don's words.

Well, in one sense, Don was right: computer performance *does* continues to rocket, and still shews no signs of starting to reach a plateau. Yet, despite this, $\mathcal{NTS}$ is, on large benchmarks, over 100 times slower than TEX, even using the much-vaunted "just-in-time" compiler. And so, we are faced with a crucial decision: do we continue to use Java, and just wait for the hardware to speed up by a further factor of 100? (Remember, the first IBM PC ran at 4.7MHz; the

first 1GHz pentium-class machine should ship this year – a factor of 200: 1). Or do we use the Java implementation just for test purposes, but re-implement Karel's design in a radically more efficient language? Or do we simply admit defeat, say "we tried", and leave it to others to see if they can be more successful than we?

These are hard questions, and there will be considerable soul-searching before we can decide on the answer; all I can say at the moment is that GUTenberg, as one of our major sponsors, will also be one of the first to know.

6. Epilogue

Although my talk has ended on a rather downbeat note, I'd like to try to lift your spirits by asking (and answering) one vital question: what lesson(s) can be learned from our experience(s)?

The first mistake was surely to under-estimate the time necessary for the initial re-implementation. Had we followed Knuth's (? apocryphal ?) algorithm for estimating the time needed to develop a major software system, we would have added 1 and then gone up to the next order of magnitude. Thus Knuth's algorithm would have suggested (had we heeded it) that we would need not one year but two decades!

In fact, we probably need about three years to complete fully what we thought could be completed in one. Is it possible to explain why?

Rather interestingly, I think the answer is "yes" (which may suggest that I am still as naïve as I was when I started the project!). According to Karel, almost all of the extra time has been spent making $\mathcal{NTS}$ 100 % TEX-compatible. Note, 100 %, not 99.9 %. It was this last 0.1 % that ate up so much of the lost time. Little things, like making sure that the console output was *identical*, even if console output is ephemeral and can never be compared other than by memory. Little things, like making sure that $\mathcal{NTS}$'s behaviour at boundary conditions is *identical* to that of TEX, even if TEX's behaviour in such conditions is sometimes flawed and at worst completely insane. Little things, like making sure that `DVI` files produced by $\mathcal{NTS}$ are *binary*-identical with those produced by TEX, not just syntactic- and semantic-compatible.

What made this situation worse was that Karel's brief was *not* to write a TEX-simulator; had that been his task, he could probably have completed the work in eight months or less. His brief was, in fact, to write *a flexible, extensible, modular* TEX simulator, which meant that every time he discovered somewhere that TEX behaved less than ideally, he had to implement two routines: (1) the base routine, which behaved exactly as a logical person would expect in the

circumstances, and (2) a TEX-compatible routine, that introduced whatever anomalous behaviour TEX itself would exhibit in those circumstances. Thus someone taking the $\mathcal{NTS}$ source in the future will find that all the necessary logical, predictable, behaviour has already been implemented; it has simply been "sub-classed" out of sight in the interests of TEX-compatibility.

What other lessons can be learned? Well, it is certainly worth re-visiting the question of implementation language. Was Java the right choice? In hindsight, the answer appears to be "no", much as it hurts to admit it. There are three primary reasons for this. (1) Java is not as type-safe as we had thought, at least if one wants both type-safety *and* efficiency at the same time. Whereas in Pascal one can write:

```
type group = (simple_group, semi_simple_group, ...)
```

and thereafter use the identifiers `simple_group` (etc.) in the *absolute* certainty that they can never be used in a context where (e.g.) an integer is expected, this is not the case for Java. There *are* no enumerated types, and thus if one wants type-safety to be checked and enforced at the compiler level, one is virtually *forced* to use objects to represent even the simplest enumerated type.[2] And objects, of course, carry considerable baggage with them, and their use (in excess) has a heavy performance impact. (2) Java lacks generic types, and thus in a situation in which one wants to manipulate (say) lists of different *types* of object, one is forced either to write type-specific code for each type of object or to use the only truly generic object (`Object` itself), and then to use casts. In the latter case, type-checking is deferred from compile-time to run-time, with an accompanying lack of (a) compile-time type-safety, and (b) efficiency. (3) Java imposes considerable performance overheads. If $\mathcal{NTS}$ were ten times slower than TEX, I might be prepared to argue that (a) Java performance will continue to improve, and therefore we should be within touching distance of TEX's performance before too long; and (b) a performance degradation is acceptable if both maintainability and extensibility are considerably enhanced. But I cannot, in all honesty, offer these defences in the present situation: if $\mathcal{NTS}$ remains 100 times slower than TEX, its chances of ever being used in earnest are vanishingly remote.

Java's strengths, on the other hand, remain virtually unchallenged; it *is* portable (and obviates any need for system dependencies and/or local modifications), it *has* attracted a large user (=programmer) base, and it *does* offer seamless network connectivity. At the moment, we are uncertain which (if any) other language could offer these advantages while avoiding Java's limitations. "Generic Java", particularly if supported by Sun, would make great sense; Eiffel

2. the `java.util` package does recognise the need for enumerated types, but unless and until they are included in the base language, efficiency will clearly be compromised

looks interesting, too. All I can say at the moment is that Karel will finish $\mathcal{NTS}$ V 0 using Sun's Java; if, after that, there is general consensus that the project should continue, we will investigate the option of translating (probably automatically) $\mathcal{NTS}$ from Java to another, more efficient, language. And beyond that is too far to see!

And one final problem, which has dogged this project, and which (sadly) doesn't seem likely to disappear. That is a problem of communication. The team are geographically diverse, with representatives from at least five nations (UK, CZ, DE, PL, NL); our programmer is based in CZ, where the only other team member is more than fully occupied running a major university (Jiří is now Rector of Masaryk University). Thus Karel lacks the day-to-day support of others with whom to discuss progress and problems other than by e-mail and at occasional group meetings. Almost certainly, communication problems have also led to various misunderstandings within the group, which are frequently seen as being politically motivated. Politics *have* cast a shadow over this project, of that there is no doubt; yet equally without doubt every member wants the project to succeed. I believe that the goodwill which exists outweighs the difficulties which can occur, and that we will be able to bring this project to a state where $\mathcal{NTS}$ is complete and usable.

But Don advised us that we should be prepared at some point to do what he has done – to say "enough is enough" and to allow others to carry the torch forwards. I'm sure we aren't ready to do that yet – there are far too many exciting challenges to be met – yet the time will undoubtedly come when $\mathcal{NTS}$ will itself be regarded as *passé*, and others will be keen to take on the challenge of carrying computer typesetting (in the finest TEX tradition) forward in as-yet unforeseen ways. I hope that amongst those who take up this challenge, members of GUTenberg will figure prominently: you have amongst you many who have contributed enormously to the furtherance of TEX, some of whom I have had the pleasure of knowing as friends as well as colleagues. On behalf of the $\mathcal{NTS}$ project, I thank you most sincerely for your support; I hope that you enjoy the demonstration of $\mathcal{NTS}$ which follows, after which I will try to answer any questions which you may have.