

complex expressions) when encountering limits of a given C compiler.

Since we wrote the translator itself in C, the only compiler needed to build TurboTeX from `tex.web` on a new, stand-alone machine is a C compiler. (We also apply the translator to TANGLE.) We maintain portability for all the translation tools, and not just the TeX program.

A further goal was to have a single C source for all target machines and operating systems. C is suitably adept at conditional compilation with its standard preprocessor, and we used that language feature to keep a unified C source for both Unix System V and MS-DOS. We used only features common to the various C language standards to minimize the amount of conditional code. We also combed out all operating system dependencies into small, hand-written, separately-compiled source files. We maximized the portability of the source code by carefully designing the translator to produce only portable C constructs.

Finally, our watchwords were “correctness” and “certifiability”. We designed a product that we could finish on a limited schedule and budget, but which would still be a full implementation of TeX. Thus we tended to decide design details in favor of conserving simplicity of the handwritten code *vs.* saving execution time or code size.

Creating PASCHAL, Our Pascal-to-C Translator

The bulk of the TurboTeX project involved the Pascal-to-C translator, which we named PASCHAL. We chose a name so close to “Pascal” because we wanted to emphasize the equivalency of the C output to the Pascal input. When speaking aloud, one may pronounce the name “PASCHAL” with a Mediterranean accent to distinguish it from Pascal, and thereby also emphasize the etymology of both terms in Greek (*Páscha*) and Hebrew (*Pesakh*). (Those unaccustomed to these languages may simply raise the pitch of the second syllable over the first.)

We were fortunate to have had experience writing large programs in both Pascal and C, so that we had a clear understanding of the issues involved in translating one into the other. We also have written several language-based translators for other applications.

The two languages are thoroughly similar in all but a few features, and TeX is restrained in the use of Pascal’s distinctives. In the cases of a few *hapax legomena*, we simply rewrote the difficult Pascal statements in a more common dialect.

Our development system, an AT&T 3B1 running Unix System V, provided the YACC and Lex tools to automate the production of PASCHAL’s parser. We also referred to the grammar from the Berkeley pc Pascal compiler (likewise written in YACC) to guide us in designing and writing PASCHAL’s parser.

To date, PASCHAL has required 240 hours of senior programmer effort and a total of 3200 lines (10,800 words) of source code in YACC, Lex, and plain C. The PASCHAL executable on our machine is only 62K bytes (not counting dynamic memory used for storing pending output).

The various modules of PASCHAL organize as follows: the main function to interpret command-line options and input file names, the lexical analyzer, the syntax tables and semantic actions of the parser, dynamic memory allocation, string handling, symbol table, parameter list interpreter, subrange interpreter, non-scalar type analyzer, variant record decomposition, and function-return-value replacement.

Besides the PASCHAL translator proper, we created a run-time “Pascal Compatibility Library” in C to replace the Pascal run-time library, and a header file, `paschal.h`, which contains macros used by the PASCHAL output.

Translating TeX to “Pure” C

We will now discuss translating TeX in Pascal to TeX in C. We will defer the details of how we translated operating-system dependencies, such as the run-time library and I/O, until the following sections on Unix and MS-DOS.

The easiest parts to translate include most of the control structures, such as compound statements, conditionals, loops, and so on. Using YACC and a few string-handling functions does the job. Certain cases, like Pascal **for** loops, have a more complex meaning in Pascal than in C, and PASCHAL puts in extra C statements to achieve the absolutely equivalent effect.

One aspect we handle on the lexical level is conflicts in reserved words. Some of TeX’s identifiers, like *int*, are reserved words in C, so we recognize them specially and prefix a “PCL” on them before they reach the parser. In this way we avoid any conflicts in the C output.

Also on the lexical level, we recognize the Pascal built-in functions and procedures, and change them into the names used in the “Pascal Compatibility Library.” This library eventually links with the final executable.

The lexical analyzer also handles conversion of numeric and character constants into appropriate C constants. Most operators also have a simple lexical equivalent, such as Pascal's "<>" and C's "!=".

The operators and precedence in the two languages have a few subtle differences, requiring special care in certain cases. For example, Pascal's "/" is strictly real division, while in C it is either the integer modulo or real division, depending on the operands. **PASCHAL** recognizes all such special cases and is careful to clothe them in appropriate parentheses and type casts. Built-in functions can also be very tricky. For example, the Pascal "abs" adapts implicitly to the type of the operand while C's "abs" is strictly integer. **PASCHAL** cages chameleons like this with C preprocessor macros that mimic the Pascal behavior.

Most cases of the Pascal **type** statement translate into a C **typedef**. **PASCHAL** does an optimal mapping of each Pascal subrange type into a C **char**, **short**, **int**, or **long**, based on the storage or execution speed profile of the target machine. **PASCHAL** uses the C **unsigned** modifier to optimize use of storage or execution time. **PASCHAL** translates Pascal **const** statements into C preprocessor **#define**'s, thus permitting the C compiler to reduce constant expressions at compile-time.

A Pascal **record** translates into an equivalent C **struct**. In the case of variant **record**'s, C **union**'s appear. The syntax of Pascal's variant **record**'s, however, is different enough from C's **struct**'s and **union**'s to make coding of the task difficult. In some cases **PASCHAL** must generate extra synthetic identifiers to label parts of the C **struct**.

Arrays pose several problems. Pascal separates items in lists of array subscripts with commas, while C requires them to be each in square brackets. Pascal permits array bounds to start from any integer, while C always uses a zero lower bound. Thus **PASCHAL** must keep a symbol table of array names and their lower bounds, and insert an offset into each dimension of each reference to an array element.

When an array appears on the left side of an assignment, we know in the case of **TeX** that it is a string assignment. **PASCHAL** outputs C code to copy a string in that case.

In Pascal, a function or procedure call may or may not have a parenthesized actual parameter list, so the translator has to maintain a symbol table to know whether an identifier reference is to a variable or a call.

Pascal and C take different approaches to declaring formal parameters to functions and procedures. Pascal does it sensibly, whereas C expects you to recite the identifier list once in the parentheses and once again (with types this time, please) after the parentheses.

Pascal and C use radically different methods to return a value from a function. Pascal requires that the returned value be assigned to the function name and that control flow exit at the end of the function body. C simply uses a built-in "return" statement. Thus **PASCHAL** must insert a synthetic returned-value variable declaration into each function, convert all Pascal assignments of return values to assignments to that variable, and insert a **return** statement at the end of the function to return the value of the variable.

To make separate compilation possible, **PASCHAL** will gather external declarations for the whole Pascal program together and create a C "include" file from them. The functions of the C source may then be arbitrarily split into separate source files for separate compilation. The **PASCHAL** accessory program **splitp**, given an arbitrary number of lines, splits the **PASCHAL** C output into smaller source files, each containing about that number of lines.

In a few cases of Pascal language features, we just change the **TeX** source (with the changefile) to avoid a difficult Pascal feature that is rarely used by **TeX**. For example, **TeX** uses Pascal's non-local goto's for error termination, and we follow Knuth's suggested strategy by substituting a call to the C **exit()** function.

Connecting **TeX** to the Unix System V Environment

The matter of file I/O does not permit much in the way of automatic translation. We modified by hand **TeX**'s code for file opening/closing, text line input, and read and write statements. The changes convert these operations to use the C standard I/O library. While this hand-crafted modification was tedious, it amounted to only a few hundred lines and was mechanical.

To the list of handmade modifications we must add the "dirty Pascal" which Knuth cites in the index to *TeX: The Program*. C is able to perform these tricks as well as Pascal.

C provides a standard way to access command-line arguments, and we provide the customary **TeX** features in that respect. Also straightforward in C under Unix System V are: invoking a sub-shell for editing during a **TeX** run, detecting an interrupt key-press, and determining the date and time.

We use a novel method made possible by C to obtain pre-loaded versions of Turbo \TeX . We do not process core dumps into modified executable images, which has been the strategy of such programs as the Berkeley Unix `undump` utility. Instead, we created a utility program `fnt2init` in C, which adds C initializers to the global declarations of the Turbo \TeX source code, thus creating source code for pre-loaded versions of \TeX . The `fnt2init` utility determines the proper global variable initializations by executing the `initialize` function from the Turbo \TeX source, and by analyzing the `initex` format file which defines the preloaded state. This method has several important advantages over the core-dump method: (1) Complete portability of the pre-loading process, since C initializers are portable (as opposed to core-dump files which are inherently non-portable), (2) Smaller executable files and faster program loading, since C global variables which are initially undefined or zero take no space in executables compiled from C, and (3) Portability of the initialized data, since C correctly initializes items like character data for the underlying machine architecture.

Once we had debugged our first version of Turbo \TeX for Unix System V to where it would process complex documents, we were ready to try the \TeX TRIP. To our satisfaction, Turbo \TeX processed the entire \TeX TRIP without error on the first attempt.

Sugar-Coating Our Translation for MS-DOS PC's

Having finished our initial version for Unix System V, we turned our attention to porting the Turbo \TeX translation to the IBM PC and compatibles. For this effort we chose the Microsoft C Compiler version 5.0, since this compiler offered a run-time library compatible with the Unix standard I/O library, a reputation of being bug-free, and accommodating memory models for large programs like \TeX .

The main issues of porting a large program like \TeX to the PC have to do with the unfriendly architecture of the 8086 processor. Its limitations arise partly from its being a 16-bit processor with a 16-bit (64K byte) address space, and partly from an illogical design to expand the address space to 20 bits (1M byte, reduced to 640K on the PC).

Taking the verified Turbo \TeX C source to the PC and getting the code to compile and run correctly required an unpleasant amount of effort, almost as much as that to create PASCHAL. However, we were aided by PASCHAL itself in that as we discovered quirks in the way the PC executes

C, we were able to make small changes to PASCHAL to correct whole classes of problems in the PC's use of the source code.

The main problems were: getting the program to run in 640K, getting correct coercion of parameters in function calls, and getting I/O to work properly with the changed operating system environment.

Finishing with Output Drivers and Utilities

We ported Nelson Beebe's output drivers in C (see TUGboat Volume 8, No. 1) to Unix System V. Beebe's source code, at the time we received it, required only a few changes to compile and run perfectly. At our site we use DVIJEP for the HP LaserJet+ and DVIIBM for IBM-compatible dot-matrix printers.

The font metric and bitmap files require no translation, since they are in a completely portable format to begin with.

Encore!

We have a number of items on our wish-list of future upgrades.

In the course of translating \TeX we first translated TANGLE. The WEB and other Pascal-based utilities not strictly required to run \TeX (WEAVE, TftoPL, etc.) will follow soon.

Previewers must be written for specific graphics hardware, so we will first produce them for only a few machines, namely the AT&T 3B1 and the IBM PC, in order to have control over our own versions.

We would like to turn PASCHAL loose on METAFONT next. The METAFONT source provides some new challenges, but we are confident that our design will adapt well to any problems involved.

As to ports to other machines, we hope to complete them for the Apple Macintosh, IBM's OS/2, and VAX Unix and VAX/VMS. Turbo \TeX may run unchanged on the two VAX operating systems, but we have not yet attempted that port.

Since there is no object code standard for the hundreds of various Unix System V machines, those with Unix machines which we do not support should obtain the Turbo \TeX source code from us and compile it for their machine.

There are several optimizations to \TeX which we hope to hand-craft in C for Turbo \TeX , including rewriting the "inner-loop" code, optimizing near and far pointers in the PC version, and using C dynamic memory to optimize at run-time the use of memory for arrays.

We would like to improve PASCHAL, or possibly write a post-processor, to improve the looks of the

resulting C code. Like processed cheese, the food value of the original is there, but the flavor is changed and the texture is gone.

Distributing the Product

We have elected to make Turbo \TeX a “semi-commercial” product within the US, that is, we will charge a modest license fee for each copyrighted copy of the binary and/or source code. However, unlike other commercial versions of \TeX , the source code will still cost less than the other’s binaries. We will distribute a complete package including Turbo \TeX , utilities, and printer drivers.

Late-breaking News. We have completed some preliminary benchmarks on the VAX BSD version of Turbo \TeX , with encouraging results. We compared Turbo \TeX in C to the public-domain Unix \TeX distribution in Pascal on a VAX 750. We observed an execution speed-up factor of between 1.6 to 3.0 compared to the Stanford distribution \TeX (the factor varies depending on the type of document being formatted). The size of the Turbo \TeX executable code is about 60% of the distribution version.

easy \TeX

Ester Crisanti
Alberto Formigoni
Paco La Bruna

1 Introduction

1.1 easy \TeX _{1.0}

\TeX has introduced new powerful tools for scientific documents typesetting, allowing formulae to be easily built up through a linear language. As a new tool using \TeX , a project was born in 1984 at the Istituto di Cibernetica (now Dipartimento di Scienze dell’Informazione) dell’Università degli Studi di Milano, Italy.

That project has produced easy \TeX _{1.0} that we propose as a new powerful tool for \TeX documents typesetting.

easy \TeX is an interactive Formula Processor, developed from the initial idea of Prof. Gianni Degli Antoni, Dipartimento di Scienze dell’Informazione, planned and implemented by TECO GRAF

with the collaboration of Dipartimento di Scienze dell’Informazione dell’Università degli Studi di Milano.

It allows the interactive typewriting of mathematical formulae on IBM-compatible Personal Computers. The formulae produced by easy \TeX are memorized in ASCII standard files, prepared in order to be processed by \TeX , either including such files in other ones by means of the \TeX command “\input”, or using usual editor commands for file merge.

The formula being built up is displayed on the screen through the fonts created with METAFONT and it is also possible to use every symbol and mathematical font.

The use of easy \TeX is very simple, since the user is driven in his work by a pop-up menu interface, by means of which the choice of operators and mathematical symbols is easily made. It is also possible to select some virtual keyboards which, because they can be displayed on the screen, achieve a correspondence with the physical keyboard, allowing insertion of characters belonging to different alphabets, like the greek, or a wide selection of mathematical symbols.

Also, complex mathematical formulae can be typeset in an easy way, similar to the one used in writing by hand the same formula. Both for the foregoing reasons, and because the positioning of the cursor is automatically obtained through an interactive construction of the formula on the screen, easy \TeX offers to the user a good facility for the preparation of a \TeX document.

easy \TeX has been implemented using attributed grammar techniques, as developed by D.E. Knuth. Programs have been written in C language.

2 Functional characteristics

2.1 User interface

The user communicates with easy \TeX using pop-up menus making the selection of commands simple and fast. Using easy \TeX , it is not necessary to know editing languages or to learn a particular syntax for the commands, because everything is done in an interactive way.

2.1.1 The screen layout

The screen handled by easy \TeX is structurally divided into three separated areas named:

- Menu line
- Work area
- Status line